iakovlev.org

Эльбрус

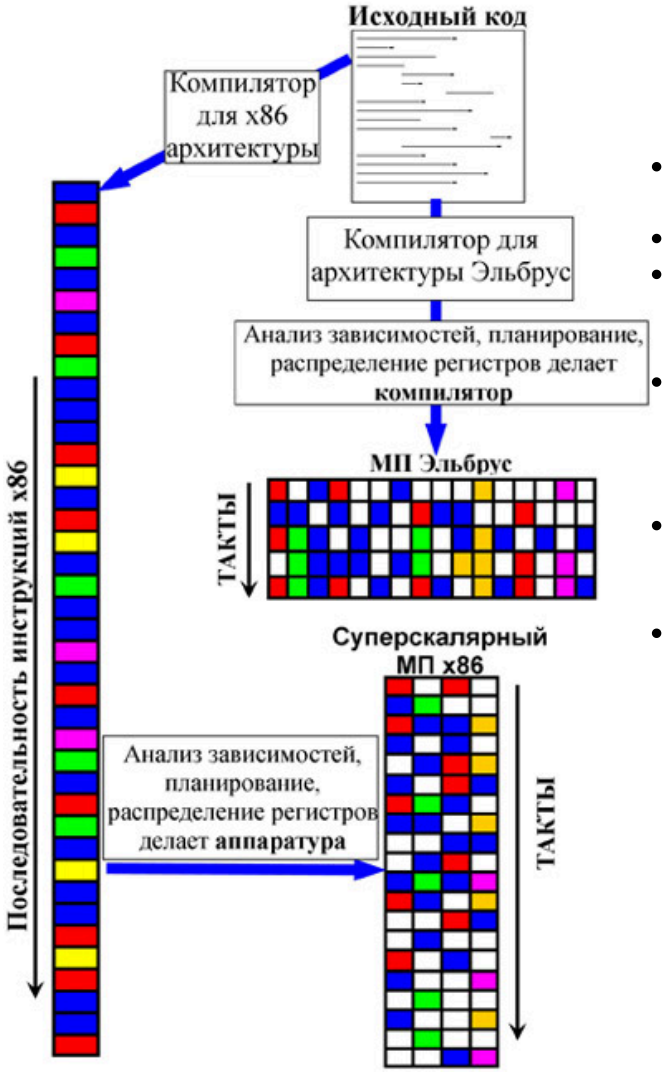
Архитектура «Эльбрус» - оригинальная российская разработка. Ключевые черты архитектуры «Эльбрус» - энергоэффективность и высокая производительность, достигаемые при помощи задания явного параллелизма операций.

Ключевые особенности архитектуры Эльбрус

В традиционных архитектурах типа RISC или CISC (x86, PowerPC, SPARC, MIPS, ARM), на вход процессора поступает поток инструкций, которые рассчитаны на последовательное исполнение. Процессор может детектировать независимые операции и запускать их параллельно (суперскалярность) и даже менять их порядок (внеочередное исполнение). Однако динамический анализ зависимостей и поддержка внеочередного исполнения имеет свои ограничения: лучшие современные процессоры способны анализировать и запускать до 4-х команд за такт. Кроме того, соответствующие блоки внутри процессора потребляют заметное количество энергии.

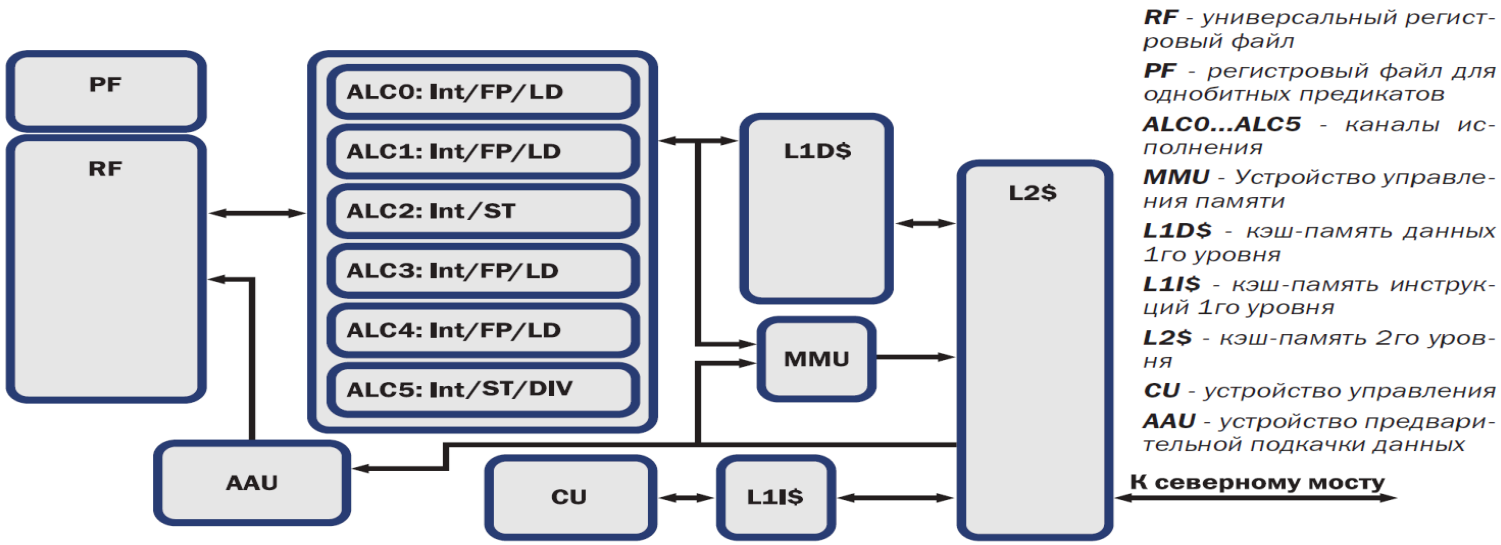
В архитектуре «Эльбрус» основную работу по анализу зависимостей и оптимизации порядка операций берет на себя компилятор. Процессору на вход поступают т.н. «широкие команды», в каждой из которых закодированы инструкции для всех исполнительных устройств процессора, которые должны быть запущены на данном такте. От процессора не требуется анализировать зависимости между операндами или переставлять операции между широкими командами: все это делает компилятор, исходя из анализа исходного кода и планирования ресурсов процессора. В результате аппаратура процессора может быть проще и экономичнее.

Компилятор способен анализировать исходный код гораздо тщательнее, чем аппаратура RISC/CISC процессора, и находить больше независимых операций. Поэтому в архитектуре Эльбрус больше параллельно работающих исполнительных устройств, чем в традиционных архитектурах, и на многих алгоритмах она демонстрирует непревзойденную архитектурную скорость.



Возможности архитектуры Эльбрус:

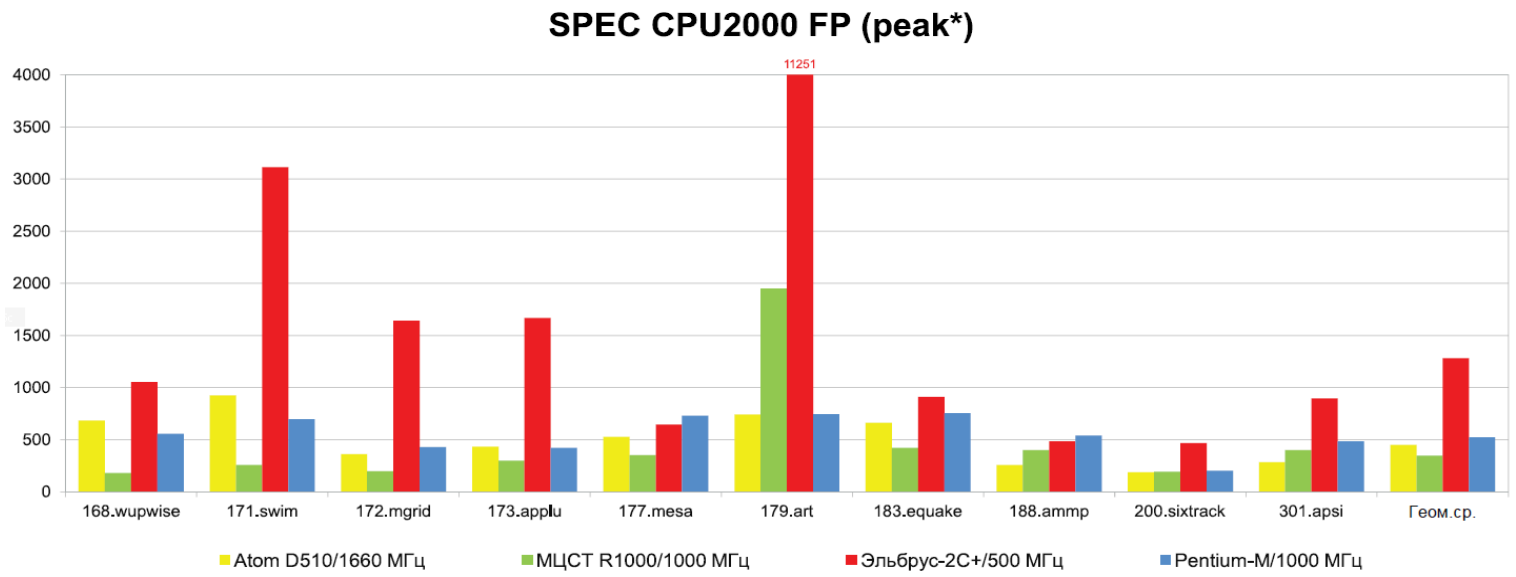
- 6 каналов арифметико-логических устройств (АЛУ), работающих параллельно.
- Регистровый файл из 256 84-разрядных регистров.
- Аппаратная поддержка циклов, в том числе с конвейеризацией. Повышает эффективность использования ресурсов процессора.
- Программируемое асинхронное устройство предварительной подкачки данных с отдельными каналами считывания. Позволяет скрыть задержки от доступа к памяти и полнее использовать АЛУ.
- Поддержка спекулятивных вычислений и однобитовых предикатов. Позволяет уменьшить число переходов и параллельно исполнять несколько ветвей программы.
- Широкая команда, способная при максимальном заполнении задать в одном такте до 23 операций (более 33 операций при упаковке операндов в векторные команды).

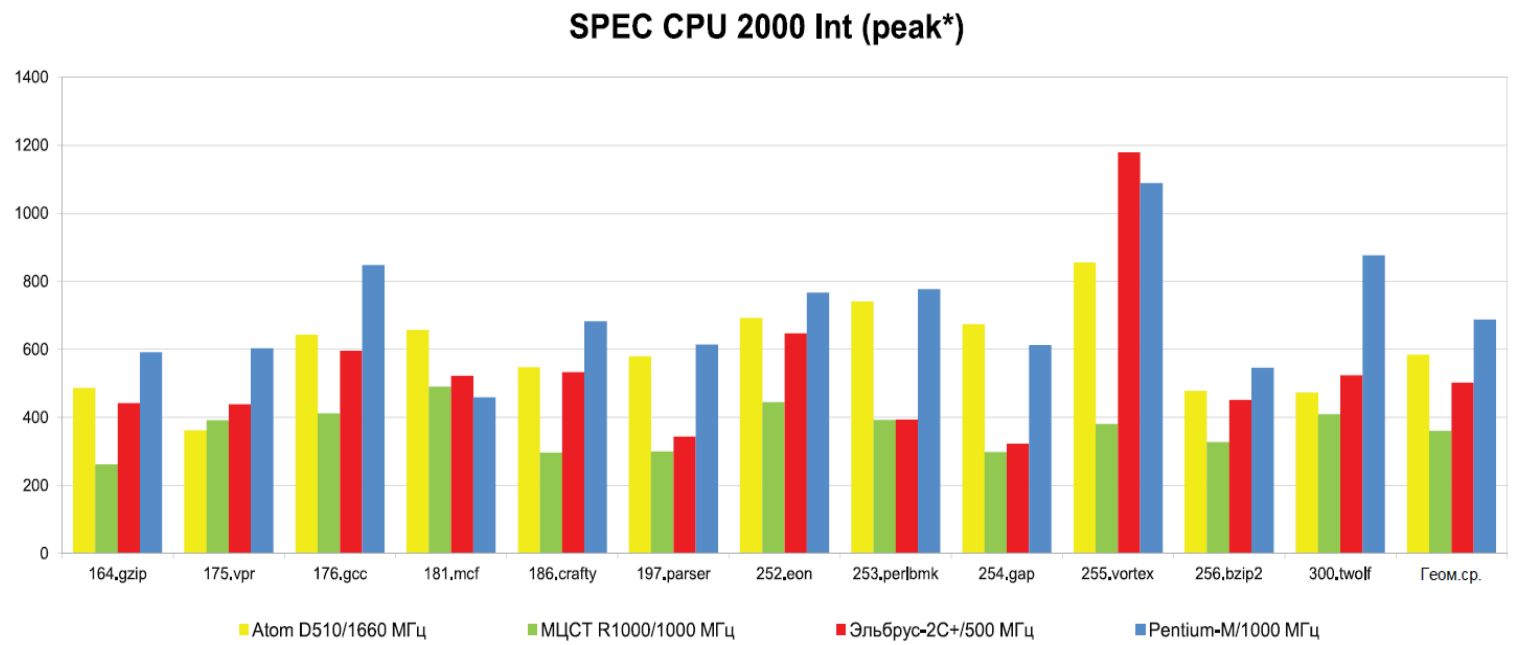


Производительность на реальных задачах:

Ниже приведена производительность процессора Эльбрус-2С+ на задачах из пакета SPEC2000 по сравнению с процессорами Intel Pentium-M ULV (1ГГц, кэш-память 1М, 2xDDR-266) и Intel Atom D510 (1,66 ГГц, кэш-память 1М, DDR2-800).

Производительность микропроцессоров Эльбрус-2С+ и МЦСТ R1000 на задачах





** Согласно требованиям SPEC Run and Reporting Rules, результаты тестирования Эльбрус-2С+ и МЦСТ R1000 публикуются как оценочные (estimates).*

Данные для Intel Pentium-M ULV получены с сайта spec.org, компилятор ICC 9.1. Для замера производительности процессора Intel Atom D510 использовалась собственная сборка тестов SPEC силами сотрудников МЦСТ.

Важно отметить, что правила комитета SPEC запрещают осуществлять модификацию исходных кодов тестов. Практика показала, что архитектура Эльбрус обладает значительным резервом производительности, который можно задействовать путём модификаций исходного кода в критических участках.

Эмуляция архитектуры x86

Еще на этапе проектирования МП Эльбрус у разработчиков было понимание важности поддержки программного обеспечения, написанного для архитектуры Intel x86. Для этого была реализована система динамической (т.е. в процессе исполнения программы, или «на лету») трансляции двоичных кодов x86 в коды процессора Эльбрус. Фактически, система двоичной трансляции создает виртуальную машину, в которой работает гостевая ОС для архитектуры x86. Благодаря нескольким уровням оптимизации удается достичь высокой скорости работы оттранслированного кода (см. диаграммы выше). Качество эмуляции архитектуры x86 подтверждается успешным запуском на

Обеспечивает исполнение ОС: MS DOS, Windows (95, NT, 2000, XP), нескольких вариантов Linux, FreeBSD, QNX

платформе Эльбрус более 20 операционных систем (в том числе несколько версий Windows) и сотен приложений.

Защищенный режим исполнения программ

Одна из самых интересных идей, унаследованных от архитектур Эльбрус-1 и Эльбрус-2 – это так называемое защищенное исполнение программ. Его суть заключается в том, чтобы гарантировать работу программы только с инициализированными данными, проверять все обращения в память на принадлежность к допустимому диапазону адресов, обеспечивать межмодульную защиту (например, защищать вызывающую программу от ошибки в библиотеке). Все эти проверки осуществляются аппаратно. Для защищенного режима имеется полноценный компилятор C/C++ и библиотека run-time поддержки.

Даже в обычном, «незащищенном» режиме работы МП Эльбрус имеются особенности, повышающие надежность системы. Так, стек связующей информации (цепочка адресов возврата при процедурных вызовах) отделен от стека пользовательских данных и недоступен для таких вирусных атак, как подмена адреса возврата. Стоит отдельно отметить, что в настоящее время вирусов для платформы «Эльбрус» просто не существует.

Сфера применения микропроцессоров архитектуры Эльбрус	
Расширенный температурный диапазон, возможность локализации производства	Государственный заказ, промышленные компьютеры, автомобильная электроника
Повышенная защищенность от вирусных атак	Платежные терминалы, сетевые экраны, взломоустойчивые серверы
Высокая производительность на криптографических алгоритмах	Модули шифрования, защищенные тонкие клиенты, прочие системы безопасности
Высокая производительность на вычислениях с действительными числами (float, double)	Робототехника, авионика, промышленные контроллеры, системы обработки изображений, суперкомпьютеры
Работа под управлением бинарного компилятора в режиме	Интернет-терминалы, маломощные рабочие станции, малогабаритные

совместимости с архитектурой x86	настольные и встраиваемые компьютеры
Защищенный режим	Особо ответственные системы, отладочные стенды

Ниже представлены характеристики актуальных моделей микропроцессоров «Эльбрус».

Характеристики микропроцессора «Эльбрус-4С»

¶

Характеристика	Значение
Тактовая частота	800 МГц
Пиковая производительность микросхемы, Gflops (64 разряда, двойная точность)	25
Пиковая производительность микросхемы, Gflops (32 разряда, одинарная точность)	50
Кэш-память данных 1-го уровня, на ядро	64 КБ
Кэш-память команд 1-го уровня, на ядро	128 КБ
Кэш-память 2-го уровня, универсальная, на ядро	2 МБ
Организация оперативной памяти	DDR3-1600 ECC, 3 канала, до 38,4 ГБ/с
Возможность объединения в многопроцессорную систему с когерентной общей памятью	До 4 процессоров
Каналы межпроцессорного обмена	3, дуплексные
Пропускная способность каждого канала межпроцессорного обмена	12 ГБ/с
Площадь кристалла	380 мм^2
Число транзисторов	986 млн.
Энергопотребление	45-60 Вт

Характеристики микропроцессора «Эльбрус-8С»

¶

Характеристика	Значение
Тактовая частота	1300 МГц
Число ядер	8
Пиковая производительность микросхемы, Gflops (64 разряда, двойная точность)	125
Пиковая производительность микросхемы, Gflops (32 разряда, одинарная точность)	250
Кэш-память данных 1-го уровня, на ядро	64 КБ
Кэш-память команд 1-го уровня, на ядро	128 КБ
Кэш-память 2-го уровня, универсальная, на ядро	512 КБ
Кэш-память 3-го уровня	16 МБ
Организация оперативной памяти	DDR3-1600 ECC
Количество контроллеров памяти	4
Возможность объединения в многопроцессорную систему с когерентной общей памятью	До 4 процессоров
Каналы межпроцессорного обмена	3, дуплексные
Пропускная способность каждого канала межпроцессорного обмена	12 ГБ/с
Площадь кристалла	321 мм^2
Число транзисторов	2.73 миллиарда
Энергопотребление	75—90 Вт

Характеристики микропроцессора «Эльбрус-8СВ»

¶

Характеристика	Значение
Тактовая частота	1500 МГц
Число ядер	8
Пиковая производительность микросхемы, Gflops (64 разряда, двойная точность)	288

Характеристика	Значение
Пиковая производительность микросхемы, Gflops (32 разряда, одинарная точность)	576
Кэш-память данных 1-го уровня, на ядро	64 КБ
Кэш-память команд 1-го уровня, на ядро	128 КБ
Кэш-память 2-го уровня, универсальная, на ядро	512 КБ
Кэш-память 3-го уровня	16 МБ
Организация оперативной памяти	DDR4-2400 ECC, 4 канала, до 68,3 Гбайт/с
Количество контроллеров памяти	4
Возможность объединения в многопроцессорную систему с когерентной общей памятью	До 4 процессоров
Каналы межпроцессорного обмена	3, дуплексные
Пропускная способность каждого канала межпроцессорного обмена	12 ГБ/с
Площадь кристалла	333 мм^2
Число транзисторов	2.78 миллиарда

Переход от скалярных процессоров к конвейеризированным и суперскалярным

Простейший скалярный процессор выполняет одну машинную команду за такт: пока не заканчивается предыдущая, следующая команда не начинает выполняться. При этом команды выстроены в цепочку, порядок которой задан в машинном коде. Но при этом совершенно необязательно, чтобы следующая команда пользовалась результатом предыдущей. Это позволило еще в середине 20 века выработать два пути ускорения исполнения команд.

Первый подход обозначен термином *конвейеризация*.

> В нём используется принцип разбиения команды на несколько последовательных стадий: прочитать команду из памяти, декодировать ее, прочитать параметры этой команды, отправить на исполнительное устройство, результат команды записать в регистр назначения. Разбитие команд на разные стадии позволяет запускать следующую команду до того, как закончилась

предыдущая. Термин «конвейеризация» пришел из промышленного производства, где этот принцип позволяет многократно увеличить темп изготовления продукции.

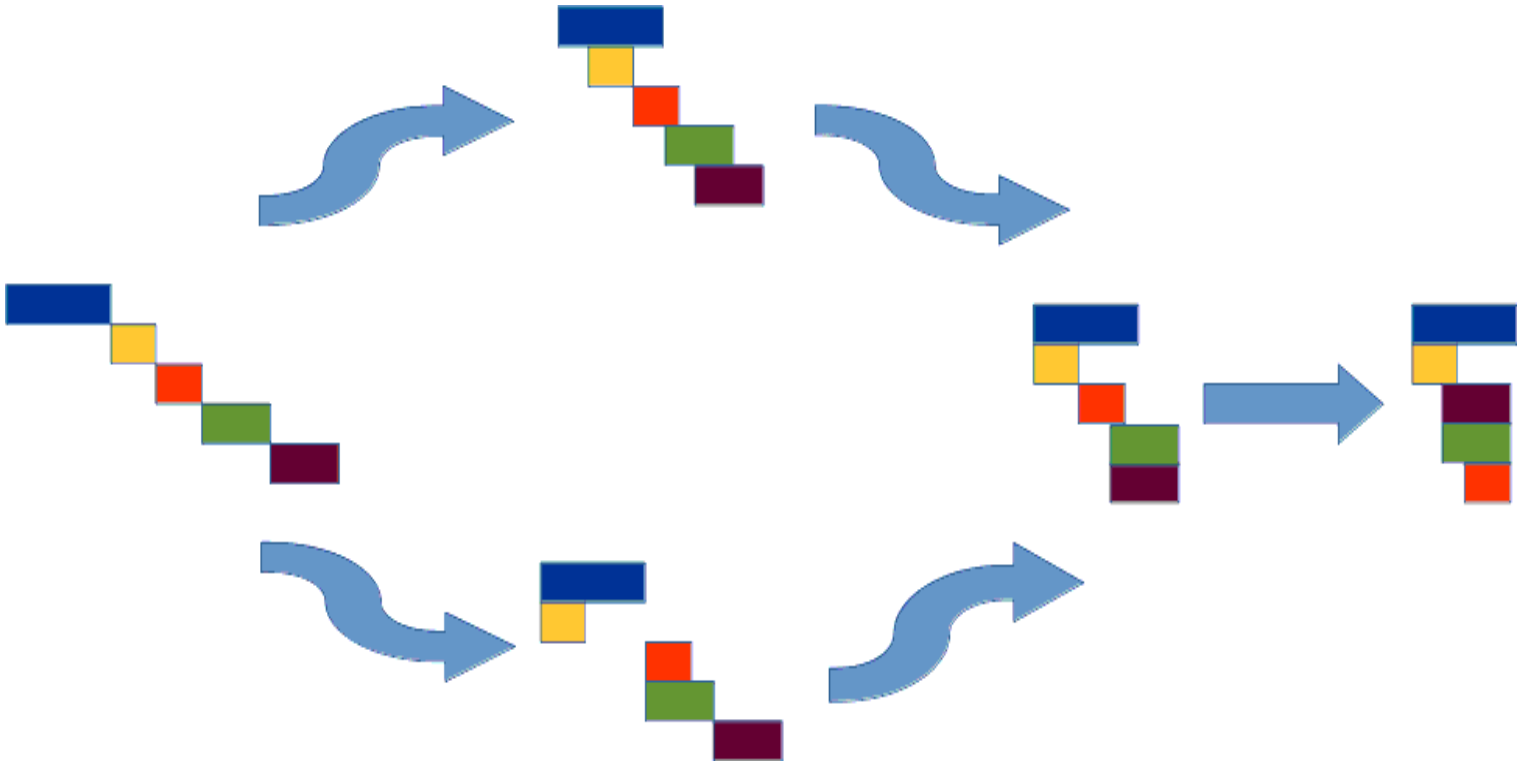
Второй подход стремится к *параллельной группировке* команд.

Для этого в последовательности команд нужно обнаруживать наборы, не пересекающиеся по используемым (аргументы) и определяемым (результаты) ресурсам. Такую группу команд можно запустить на исполнение одновременно.

Можно заметить, что описанные подходы не противоречат друг другу. Семейство архитектур, объединяющих эти принципы, известно как *in-order superscalar*.

В 60-е годы впервые был реализован следующий шаг - изменение последовательности команд относительно друг друга прямо в процессе исполнения. Такой подход назвали *out-of-order superscalar* (в дальнейшем OOOSS). Вся эволюция подходов к параллельному исполнению множества инструкций представлена на рисунке

Переход от скалярных процессоров к суперскалярным с возможностью перестановки инструкций:



В 80-е годы начала развиваться альтернативная архитектура процессоров. Их отличительной особенностью является переупорядочивание команд не во время исполнения, а во время

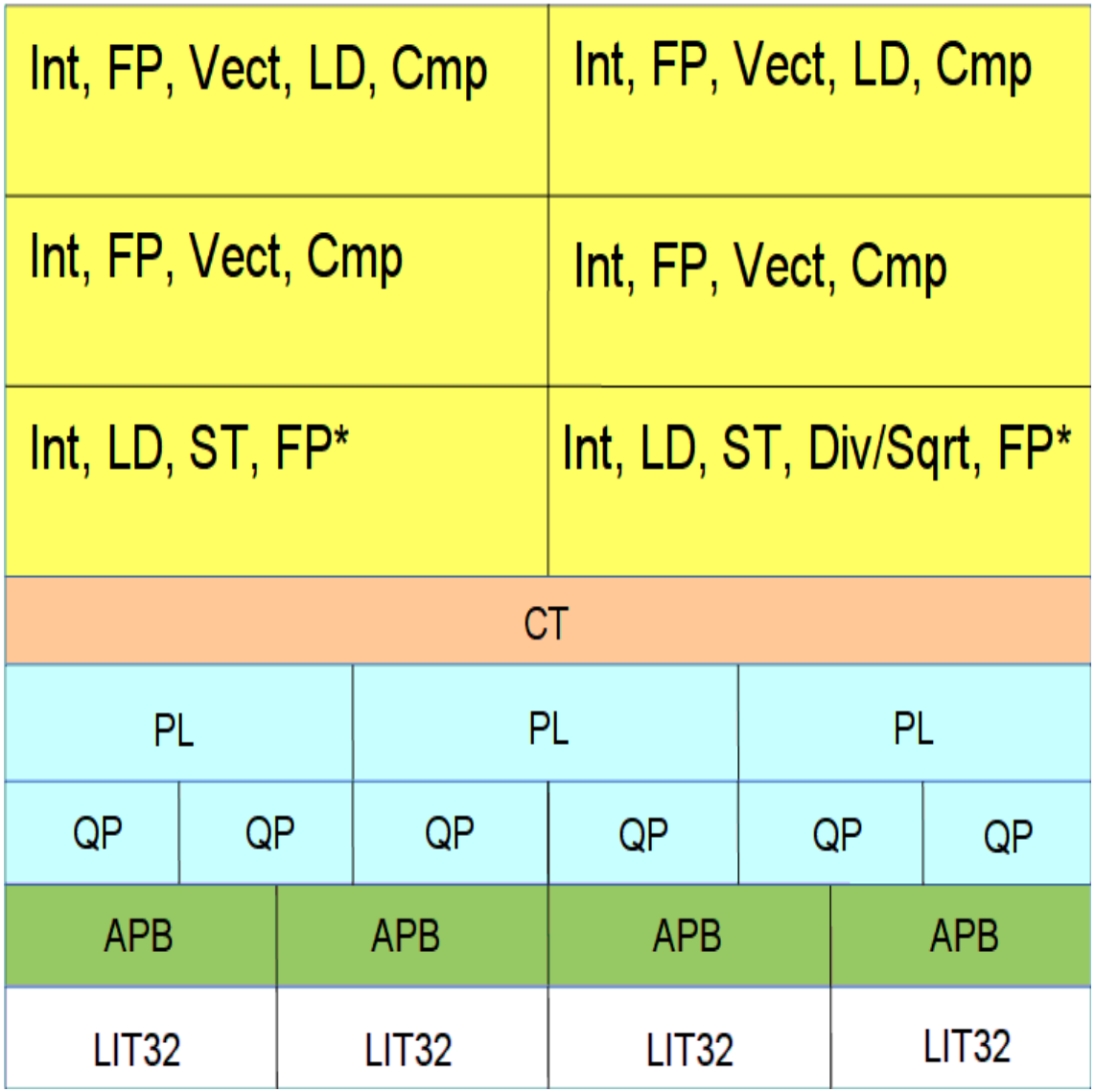
компиляции программы. Также ключевой характеристикой этой архитектуры является использование так называемых широких команд, которые позволяли выразить параллельность множества операций в ассемблере. Такая архитектура называется VLIW — «очень длинная машинная команда».

Широкая команда (ШК)

Под широкой командой Эльбрус понимается набор элементарных операций Эльбрус, которые могут быть запущены на исполнение в одном такте.

Рассмотрим ШК с точки зрения исполнительных устройств . В ШК «Эльбрус» доступны:

- 6 арифметико-логических устройств (АЛУ), исполняющих операции:
    - целочисленные (Int)
    - вещественные (FP)
    - сравнения (Cmp)
    - чтения из памяти (LD)
    - записи в память (ST)
    - над упакованными векторами (Vect)
    - деления и квадратного корня (Div/Sqrt)
- 1 устройство для операции передачи управления (CT);
- 3 устройства для операций над предикатами (PL);
- 6 квалифицирующих предикатов (QP);
- 4 устройства для команд асинхронного чтения данных по регулярным адресам в цикле (APB);
- 4 литерала размером 32 бита для хранения константных значений (LIT).



В каналах 2 и 5 операции над вещественными числами поддержаны начиная с версии системы команд v4

Средняя степень наполнения широкой команды полезными операциями определяет так называемую логическую скорость работы, которая отражает производительность процессора при условии отсутствия блокировок конвейера. Блокировки исполнения спланированного кода,

снижающие производительность процессора, могут быть вызваны различными причинами: ожидание кода, ожидание данных, неверное планирование, неподготовленный переход.

Можно сказать, что задачей повышения производительности кода на архитектурах VLIW является статическое обнаружение параллелизма на уровне операций, планирование операций с учетом найденного параллелизма, обеспечивающее высокую логическую скорость, и одновременно с тем проведение оптимизаций, снижающих количество блокировок исполнения.

Рассмотрим назначение и возможности доступных в ШК устройств.

Арифметико-логические устройства (АЛУ)

В процессорах `E4C/E8C` имеется шесть АЛУ с номерами 0-5.

В составе `АЛУ0` и `АЛУ3` присутствуют:

- целочисленное арифметическое/побитовое/сдвиговое устройство;
- устройство сравнения;
- устройство для операций с упакованными целочисленными значениями шириной 8 бит, 16 бит, 32 бита;
- вещественное арифметическое устройство;
- вещественное арифметическое устройство над упакованными 32-разрядными вещественными числами;
- устройство обращения к памяти по чтению.

В составе `АЛУ0` также присутствует устройство обращения к специальным регистрам.

В составе `АЛУ1` и `АЛУ4` присутствуют:

- целочисленное арифметическое/побитовое/сдвиговое устройство;
- устройство сравнения;
- устройство для операций с упакованными целочисленными значениями шириной 8 бит, 16 бит, 32 бита;
- вещественное арифметическое устройство;
- вещественное арифметическое устройство над упакованными 32-разрядными вещественными числами.

Вещественные устройства в `АЛУ0`, `АЛУ1`, `АЛУ3` и `АЛУ4`, а также целочисленные устройства в `АЛУ1` и `АЛУ4` позволяют исполнять две последовательно зацепленные друг за друга операции в качестве одной трехаргументной операции, например, `shl_addd ( a << 2 + 7)` или `fmul_subs ( x * y + z)`.

Промежуточный результат первой операции не записывается в регистр, а используется только в качестве аргумента второй операции. Такие операции называют комбинированными.

В составе АЛУ2 и АЛУ5 присутствуют:

- целочисленное арифметическое/побитовое/сдвиговое устройство;
- устройство обращения к памяти по чтению/записи.

Также в АЛУ5 присутствуют устройство вещественного и целочисленного деления и устройство извлечения квадратного корня.

Предикатное устройство

В широкой команде можно выполнить до трех логических операций над предикатными регистрами, причем длительность операций составляет 1/2 такта, и поэтому в одном такте можно планировать логические операции, где вторая группа операций использует результат первой группы.

Устройство асинхронной подкачки массивов (АРВ)

В одной широкой команде можно исполнить до 4 операций чтения из буфера АРВ.

Устройство управления

В одной команде можно исполнить не более одной операции передачи управления и не более одной операции подготовки перехода.

Определяющие свойства архитектуры «Эльбрус»

В общепринятой классификации архитектуру «Эльбрус» можно отнести к категории VLIW. Доступ к аппаратным ресурсам процессора базируется на использовании широких команд (ШК). При компиляции каждого фрагмента программы происходит максимальное распараллеливание вычислительного процесса по всему полю возможных вычислительных устройств.

Архитектура «Эльбрус» включает ряд универсальных решений, свойственных современным высокопроизводительным микропроцессорам:

Регистровый файл

Параллельное исполнение операций по сравнению с последовательным требует необходимого количества оперативных регистров. Архитектура определяет регистровый файл объемом 256 регистров для целочисленных и вещественных данных, 32 регистра предназначены для глобальных данных и 224 регистра — для стека процедур.

Предикатный файл

Состоит из 32 двухразрядных регистров — предикатов. Функция может использовать все 32 предиката.

Спекулятивный режим выполнения команд

Параллельному выполнению операций препятствуют определяемые при компиляции зависимости по управлению и зависимости по данным. Выполняя операции раньше, чем становится известно направление условного перехода, или считывая данные из памяти раньше предшествующей записи в память, можно ускорить выполнение программы. Но подобное перемещение операций не всегда допустимо из-за неопределенности их поведения при исполнении. В первом случае (выполнение раньше условного перехода) операция, которая не должна выполняться, может вызвать прерывание. Во втором случае (выполнение чтения раньше предшествующей записи) из памяти может быть считано неправильное значение.

Архитектура «Эльбрус» вводит режимы спекулятивности по управлению и спекулятивности по данным.

Для спекулятивной по управлению операции факт возникшей исключительной операции (чтение по недопустимому адресу, деление на *0* и т.п.) сохраняется в значении результата операции. Однако сама исключительная ситуация откладывается до выяснения того, должна ли была быть выполнена эта операция на самом деле.

Спекулятивность по данным доступна посредством пары операций:

- верхняя операция читает данные;
- нижняя проверяет, была ли спекулятивно прочтенная ячейка памяти частично перезаписана с момента первой операции, и заново выполняет чтение в том случае, если это произошло.

Подготовка передачи управления — disp

Предварительная подкачка кода в направлении ветвления, а также его первичная обработка на дополнительном конвейере (на фоне выполнения основной ветви) скрывают задержку по доступу к коду программы при передачах управления. Тем самым возможна передача управления без остановки конвейера выполнения, когда уже известно условие ветвления. Архитектура микропроцессора определяет средства предварительной подкачки кода для трех команд передачи управления.

Предикатное и спекулятивное исполнение операций

Пользуясь этими механизмами, можно планировать в одной широкой команде операции, относящиеся к различным ветвям управления, избавляться от дорогостоящих операций перехода, переносить арифметико-логические операции через операции перехода.

Программная конвейеризация циклов

Позволяет наиболее эффективно исполнять циклы с независимыми (или слабо зависимыми) итерациями.

В программно-конвейеризированном цикле последовательные итерации выполняются с наложением - одна или несколько следующих итераций начинают выполняться раньше, чем заканчивается текущая. Шаг, с которым накладываются итерации, определяет общий темп их выполнения, и этот темп может быть существенно выше, чем при строго последовательном исполнении итераций. Такой способ организации исполнения цикла позволяет хорошо использовать ресурсы широкой команды и получать преимущество в производительности.

Архитектура микропроцессора содержит средства управления режимами выполнения пролога и эпилога цикла (разгонной и завершающей части конвейера), которые позволяют единым образом программировать выполнение всего цикла. В регистровом и предикатном файлах можно определять области для организации вращательного переименования регистров по принципу конвейерной ленты. Это позволяет запускать операцию со следующей итерации цикла до того, как был использован результат этой же операции на текущей итерации - конвейерная лента из регистров сохраняет значения в течение нескольких итераций.

Асинхронный доступ к массивам

Позволяет независимо от исполнения команд основного потока буферизовать данные из памяти. Запросы к данным должны формироваться в цикле, а адреса линейно зависеть от номера итерации. Асинхронный доступ реализован в виде независимого

дополнительного цикла, в котором кодируются только операции подкачки данных из памяти в FIFO-буфера. Из буфера данные забираются операциями основного цикла. Длина буфера и асинхронность независимого цикла позволяют устранить блокировки по считыванию данных в основном потоке исполнения.

Принцип использования параллельности операций для VLIW и OOOSS

Пусть есть 5 операций, которые требуется выполнить:

1. a=x/7
2. b=y+1
3. c=b<<3
4. d=x*x
5. e=y*y

На рисунке показано, что в ассемблере для OOOSS порядок операций сохранен в соответствии с исходным кодом примера, а переупорядочивание операций происходит в процессе исполнения кода. Во VLIW операции переставлены еще на этапе оптимизирующей компиляции и построения ассемблера. Фигурными скобками обозначены широкие команды. И компилятор VLIW, и аппаратура OOOSS видят, что

операция 3 зависит от операции 2 и требует ее завершения, но при этом операции 4 и 5 никак не зависят от результата первых трех операций, поэтому их можно начать исполнять раньше операции 3.

Компилятор для VLIW обладает гораздо большим окном операций для перемешивания, чем имеется на этапе исполнения у аппаратуры OOOSS. Это позволяет в некоторых случаях лучше выявлять независимые операции для их параллельного исполнения. С другой стороны, OOOSS обладает дополнительной информацией о параллелизме, доступной в динамике исполнения, например, значения адресов операций чтения и записи. Это позволяет лучше выявлять параллелизм в некоторых других ситуациях.

Подведем краткие итоги основных отличий VLIW и OOOSS.

VLIW:

- явно выраженный в коде параллелизм исполнения элементарных операций;
- точное последовательное исполнение широких команд;
- особая роль оптимизирующей компиляции;
- дополнительные архитектурные решения для повышения параллелизма операций.

OOOSS:

- перестановка и параллельное исполнение операций обеспечивается аппаратно в пределах окна исполняемых в данный момент операций;
- для переупорядочивания используются скрытые буфера, скрытый регистровый файл, неявная спекулятивность;
- достаточно большое окно для поиска параллелизма и перестановки инструкций обеспечивается аппаратным предсказателем переходов.

Преимущества и недостатки VLIW и OOOSS (курсивом помечены недостатки).

VLIW:

- больше открытых возможностей для выражения параллелизма инструкций;
- лучшая энергоэффективность при схожей производительности;
- *возможные ухудшения производительности при исполнении legacy-кодов;*
- *более сложный код для отладки и анализа;*
- *более сложный компилятор.*

OOOSS:

- эффективное исполнение legacy-кодов;
- дополнительная информация о параллельности операций, доступная в динамике исполнения;
- *расход энергии на многократное планирование одних и тех же операций;*
- *ограниченность аппаратурой окна исполняемых операций для переупорядочивания.*

Рекомендации по работе со структурами данных

Работа с различными структурами данных может существенно отличаться по средней скорости доступа, темпу чтения и изменения. При выборе той или иной структуры данных может помочь нижеследующая информация о характеристиках таких структур:

- простые одномерные массивы.

  При регулярном считывании/записи элементов массива могут достигаться теоретически максимальные показатели темпа доступа к памяти - 32 байта в такт при попадании в `L2$`, 32 байта в 3 такта при гарантированном отсутствии в `L2$`, при условии обращения к соседним элементам, причем для считывания может применяться механизм **APB**; при регулярном обращении с большим шагом (>64b) **APB** все еще применим, но темп существенно падает (до 64 раз при побайтовой обработке);

- одномерные массивы структур.

  При регулярной обработке применим **APB**, однако, следует следить за тем, чтобы набор одновременно читаемых/записываемых полей в горячих участках был как можно более компактным; весьма полезен (в ущерб наглядности) переход от массивов структур к набору массивов, хранящих отдельные поля;

- многомерные выстраиваемые массивы.

  Многомерные массивы в языке Fortran (а также многомерные массивы в языке C при условии константных длин старших размерностей) являются одномерными по сути, но индексируемыми несколькими размерностями:

```
A(i,j,k) "FORTRAN" = a(i+j*dim1+k*dim1*dim2) "C"
```

Для повышения локальности нужно следить за тем, чтобы внутренняя размерность массивов (первая) индексировалась индуктивной переменной самого внутреннего цикла:

```
for i
   for j
     for k
       A(k,j,i) // хорошо
       B(i,j,k) // плохо
```

- многомерные не выстраиваемые массивы.

  Многомерные массивы в C являются массивами указателей на массивы (указателей на массивы и т.д. по размерностям); в связи с этим чтение одного элемента превращается в набор чтений с количеством, равным размерности; анализ зависимостей по адресам становится для компилятора весьма тяжелым;

- списки.

  Обход элементов списка представляет собой цикл с рекуррентностью (зависимостью между итерациями) вида `p=p->next`, иными словами, адрес, по которому производится чтение на текущей итерации, зависит от результата чтения на предыдущей итерации.

  При таком обходе темп перебора элементов списка не превышает 1 элемент на время доступа в `L1$`. Например, для процессора E8C этот темп в 24 раза (!) меньше максимально возможного (при размере указателя 4 байта, и при условии, что все читаемые элементы расположены в `L1$`). В случае, когда все операции чтения промахиваются и в `L1$`, и в `L2$`, темп падает до 1 элемента в `t_latency` тактов; в связи с нерегулярностью адресов **APB** неприменим, но может быть эффективен механизм `list-prefetch`;

- деревья.

  Деревья могут быть реализованы несколькими способами, но каждый из этих способов обладает тем же фундаментальным свойством, что и обычные списки: обход деревьев реализуется циклом с рекуррентностью по чтению из памяти, при этом, расположение перебираемых элементов дерева в памяти, как правило, еще хуже поддается упорядочению, чем множество перебираемых элементов списка;

- хэш-таблицы.

  Хэш-таблицы, как правило, строятся на базе обычных массивов, при этом чтение элемента хэш-таблицы предваряется вычислением хэш-функции, доступ становится нерегулярным, поэтому

APB к перебору элементов хэша неприменим, тем не менее, возможна предварительная подкачка элементов хэша, считываемых на следующих итерациях.

Виды локальности данных

Виды локальности данных связаны с возможностями компилятора распознать зависимость между обращениями к этим данным. Семантика программ на императивных языках, таких как C и C++, строго последовательна; поэтому возможность распараллеливания вычислений зависит от способности компилятора обнаружить гарантированное отсутствие зависимостей между последовательностями обращения в память за данными.

Там, где логика программы не диктует необходимости определенной локальности данных, можно делать выбор в пользу одного из следующих типов локальности:

- глобальные данные:
  - местоположение - сегмент `bss` кода;
  - время жизни - вся программа;
  - адрес общедоступен;
  - не конфликтуют с другими данными (отсутствие конфликтов очевидно, если не было операций взятия адреса `&glob`);
  - глобальные данные небольшого размера можно разместить на глобальных регистрах;
- простые локальные данные без взятия адреса:
  - местоположение - регистры;
  - время жизни - до выхода из процедуры;
  - регистры не отображаются в память, ни с кем не конфликтуют;
- сложные локальные данные, либо локальные данные со взятым адресом:
  - местоположение - пользовательский стек;
  - время жизни - до выхода из процедуры;
  - адрес доступен только внутри процедуры, для передачи данных в вызываемые процедуры нужно брать адрес;
  - в связи с часто необходимой операцией взятия адреса разрешение конфликтов по адресам становится более затруднительным;
- динамические глобальные данные (`malloc`):
  - местоположение - динамически выделяемая память;
  - время жизни - до динамического освобождения `free`;
  - адрес доступен через указатели;
  - конфликты по адресам разрешимы с затруднениями, не разрешимы конфликты между разными экземплярами `malloc` в цикле;
- динамические локальные данные (`alloca`):
  - местоположение - пользовательский стек;

- время жизни - до выхода из процедуры;
- адрес доступен через указатели;
- конфликты по адресам разрешимы с затруднениями, не разрешимы конфликты между разными экземплярами `malloc` в цикле.

Рекомендации по оптимизации процедур

Перед всякой оптимизацией необходимо получить общий профиль работы приложения или задачи. Основной интерес представляют процедуры с наибольшей долей исполнения в общем профиле. Без этого анализа вполне возможно добиться значительного ускорения отдельно взятых процедур, но итоговая производительность приложения существенно не улучшится.

Анализ процедуры: начальный этап

Для получения кода процедуры с профилем необходимо воспользоваться дизассемблером:

```
ldis -I m_program my_function1
```

В первую очередь необходимо определить тип анализируемой процедуры. Примерная классификация процедур с точки зрения анализа производительности выглядит следующим образом.

Короткая ациклическая процедура (не более 30 тактов)

Такие процедуры просты для анализа. Неоптимальность короткой процедуры как правило проявляется в виде:

- плохой наполненности широких команд:
  - вследствие зацепления операций;
  - ввиду наличия длинных операций (деление, квадратный корень, вызов по косвенности);
  - вследствие конфликтов между чтениями/записями;
- блокировки(ок) от операций чтения.

Общие рекомендации по исправлению найденных дефектов производительности:

- инлайн-подстановка: собирать в режиме -fwhole, использовать двухфазную компиляцию `-fprofile-generate` / `-fprofile-use`;
- уменьшение длины зацепления;
- принудительный разрыв конфликтов;
- включение режима выноса чтений из процедур `-fipo-invup`;

- по возможности локализация данных для лучшего использования кэш-памяти.

Процедура с горячими простыми циклами/гнездами циклов

Анализ процедуры сводится к анализу работы горячих циклов. Наиболее частые проблемы:

- плохая наполненность широких команд;
- не применился механизм **apb**;
- блокировки после операций чтения из-за промахов в кэш;
- блокировки из-за превышения пропускной способности устройства памяти.

Предлагаемые пути решения означенных проблем:

- малое число итераций может привести к отказу от применения конвейеризации (как следствие, к слабой наполненности широких команд), к отказу от использования механизма **apb**; если число итераций цикла объективно невелико (<5), следует рассмотреть возможность модификации алгоритма; если число итераций объективно велико, следует использовать двухфазную компиляцию `-fprofile-generate` / `-fprofile-use`, либо добавить в исходный текст перед циклом подсказку:

```
#pragma loop count(100);
```

- конвейеризированный цикл содержит длинную рекуррентность (длинно вычислимую зависимость между итерациями цикла). Рекомендуется проверить цикл на наличие рекуррентности, в случае нахождения — оценить ее целесообразность;
- механизм **apb** не применяется из-за нерегулярного изменения адреса; рекомендуется использовать в качестве цикловых счетчиков, определяющих адрес чтения, переменные типа `long` (не `unsigned`), не производить инкрементов счетчиков под условиями;
- механизм **apb** не применяется при невозможности статического определения выровненности чтений по размеру; рекомендуется пользоваться опцией `-faligned` (входит в состав `-ffast`), подразумевающей выровненность адресов по размеру читаемого объекта;
- блокировки от операций чтения из-за кэш-промахов (`BUB_E0`): рекомендуется попробовать опции `-fcache-opt`, `-flist-prefetch`, включающие режим предварительной подкачки данных в кэш;
- блокировки по темпу работы памяти (`BUB_E2`): рекомендуется проверить темп обработки данных - сколько тактов работает цикл, сколько в нем операций чтения и записи, каков размер этих операций, какова локальность данных, какие данные могут быть найдены в кэше. Если темп существенно ниже ожидаемого, возможно, проблема в неравномерности использования ресурсов кэша второго уровня.

Сложный цикл с управлением, гнездо с управлением

Сложный цикл - цикл с управлением, несколькими обратными дугами. Некоторые сложные циклы при наличии точной профильной информации (`-fprofile-generate` / `-fprofile-use`) могут быть сведены к простым применениям цикловых оптимизаций `loop_nesting`, `loop_unswitching` и некоторых других.

Громоздкая процедура

Громоздкие процедуры характеризуются неоднородным сложным управлением и размазанным профилем исполнения. Такие процедуры часто содержат циклы, как правило, с небольшим числом итераций, вызовы других процедур. Они сложны как для оптимизации компилятором, так и для анализа производительности, и здесь возможно дать только общие рекомендации:

- если процедура стала громоздкой вследствие inline-подстановок других процедур, можно попробовать ограничить применение `inline` опциями;
- можно произвести вручную выделение важных фрагментов в отдельные процедуры.

Процедура с превалирующим оператором switch

Конструкции `switch` с большим числом альтернатив, как правило, обрабатываются компилятором достаточно эффективно при наличии адекватного профиля (см. опции `-fprofile-generate` / `-fprofile-use`). Если конструкция `switch` имеет большое количество альтернатив с равномерно распределенной малой вероятностью, тогда компилятор выразит её с помощью чтения из таблицы меток и косвенного перехода. Более эффективно при этом работают конструкции `switch` с плотным множеством значений, т.к. в случае разреженного множества значений `switch` таблица будет иметь большой размер.

Библиотечная процедура

Библиотечная процедура собирается один раз, но используется в разных контекстах с различными параметрами. Если производительность задачи определяется производительностью библиотечной процедуры, то может быть целесообразно спрофилировать важную функцию и пересобрать ее вместе с задачей.

Модель памяти

Архитектурно для процессов задач выделяется 64-битное виртуальное адресное пространство. При этом для конкретных реализаций операционных систем и программных моделей может использоваться часть возможного адресного пространства. В этой главе рассматриваются возможные

программные модели реализации работы с адресным пространством памяти, а также разделение памяти на основные семантические компоненты.

Под семантическим компонентом пространства памяти понимается область, имеющая специфическое использование в приложении и не имеющая адресных зависимостей с любой другой областью. Такие компоненты памяти будем называть сегментами с определенными именами. Каждый сегмент может иметь разбиение на логические подразделы с различными правами доступа. Для каждой исполняемой задачи сегменты можно разделить на *видимые сегменты* и *служебные сегменты*.

- К видимым сегментам задача имеет возможность обращения.
- Служебные сегменты необходимы для функционирования задачи.

К служебным сегментам возможно обращение только в привилегированном режиме, что могут делать только процедуры операционной системы. В данном документе служебные сегменты не рассматриваются.

Сегменты программы

В нижеприведенной таблице приведен список сегментов программы с кратким описанием характеристик каждого сегмента.

Атрибут разделения характеризует возможность использования содержимого сегмента разными процессами. Реальное использование этой возможности зависит от операционной системы.

Атрибут адресации описывает возможность обращения к содержимому сегмента различными способами адресации. Реализация той или иной возможности зависит от программной модели реализации кода задачи. Описание семантических моделей будет дано ниже.

Сегменты программы

Название сегмента	Разделение	Количество	Возможная адресация	Содержимое
TEXT	Да	1 на модуль	Относительно CUD абсолютная	Исполняемый код процедур
DATA	Нет	1 на модуль	Относительно GD абсолютная	Глобальные данные
HEAP	Нет	1 на задачу	Абсолютная	Динамические данные
STACK	Нет	1 на thread	Относительно USD	Локальные данные процедур

Название сегмента	Разделение	Количество	Возможная адресация	Содержимое
CHAIN STACK	Нет	1 на thread	?	Сохранение информации процедурного механизма
REGISTER STACK	Нет	1 на thread	Относительно WD	Локальные и рабочие данные процедуры
THREAD DATA	Нет	1 на thread	?	?
SHARED DATA	Да	Любое	?	?

Организация обращения в память

Доступная задаче память имеет страничное разделение. Соответственно, минимальный квант размещения сегмента равен странице. Для каждой страницы устанавливаются права доступа. Возможен доступ по чтению (R), по записи (W), по исполнению (X). Возможны также любые комбинации этих прав.

Доступ может иметь признак привилегированности. В страницы, имеющие признак привилегированного доступа, возможно обращение только в привилегированном режиме, доступном операционной системе.

Регулярные страницы не имеют особых признаков доступа, таких как привилегированность или адресная защищённость. В регулярную страницу возможен доступ по абсолютному адресу в виде целого числа, а также по дескриптору.

Архитектурно обращение в память поддерживается следующими семействами операций:

- LD/ST - обращение в память по целому. Этими командами реализуется обращение в память по абсолютному адресу. Обращение возможно только в регулярные страницы. Операции имеют два адресных операнда формата 64 разряда. Адрес доступа в память формируется суммированием операндов.
- LDGD/STGD – обращение в память относительно регистра GD. Операции имеют два адресных операнда формата 32 разряда. Адрес доступа в память формируется суммированием операндов и адреса начала области памяти из регистра GD. Если полученный адрес выходит за границу области, описываемой регистром GD, при обращении в память возникает прерывание.

Все семейства операций обращения в память поддерживают обращения по следующим форматам:

- Байт (B) – обращение в память размером 8 разрядов.
- Половина слова (H) - обращение в память размером 16 разрядов.

- Слово (W) - обращение в память размером 32 разряда.
- Двойное слово (D) - обращение в память размером 64 разряда.
- Квадро слово (Q) - обращение в память размером 128 разрядов.

Примечание. Для операций доступа в память семейств операций LD/ST, LDSS/STSS, LSDS/STDS, LDGS/STGS, LDFS/STFS, LDCS/STCS, LDES/STES, обращение по Q формату не поддерживается.

Семантические модели организации памяти

В соответствие с различными возможностями обращения в память могут быть реализованы семантические модели, различающиеся способом представления адресных данных и устройства распределения памяти. Отметим, что сборка кода задачи требует однородности семантической модели всех её компонентов. Нельзя собрать программу из модулей разных семантических моделей. Это, однако, не относится к операционной системе. Семантическая модель операционной системы может отличаться от семантической модели задачи. Поэтому интерфейс с операционной системой может иметь различия с интерфейсом между программными компонентами задачи.

- Режим 32-х разрядной адресации (далее режим 32). В этой семантической модели регистры GD и CUD описывают всю доступную задаче область памяти. Представителями адресных данных являются 32-х разрядные смещения относительно этих регистров. В рамках 32-х разрядного адресного пространства эти смещения можно считать абсолютными адресами. Размеры указателей равны 4 байтам. Обращение в память реализуется методом доступа по регистру GD через семейство операций LDGD/STGD.
- Режим 64-х разрядной адресации (далее режим 64). В этой семантической модели всё выделенное задаче пространство памяти должно быть регулярным. Обращение в память реализуется методом доступа по целому семейством операций LD/ST. Представителями адресной информации являются абсолютные адреса. Размеры указателей равны 8 байтам.

Распределение данных

В разделе даются правила размещения данных в памяти. Отметим, что требование выравнивания продиктовано исключительно соображениями эффективности работы полученного кода. Обращение в память по невыровненным адресам может иметь неэффективную аппаратную реализацию.

Глобальные переменные

При распределении в памяти глобальных переменных применяется следующее правило выравнивания. Переменные размера большего, чем 8 байт, должны быть выровнены на 16 байт. Переменные меньшего размера должны быть выровнены на границу следующей большей степени 2.

Переменные типа Common Block (FORTRAN) должны быть выровнены на 16 байт независимо от размера. В таблице приведены требования выравнивания для переменных различного размера.

Сегменты программы

Размер в байтах	Требование выравнивания в байтах
1	1
2	2 (четные адреса)
3-4	4
5-8	8
9 и более	16

Глобальные переменные размещаются в сегменте DATA. Доступ к глобальным переменным зависит от семантической модели.

Доступ к глобальным переменным в режиме 32-х разрядной адресации

Доступ к глобальным переменным для статически собираемого кода осуществляется по смещению относительно регистра GD. Это смещение в сегменте DATA модуля плюс адрес загрузки сегмента DATA. Для статически собираемого кода установку этих значений обеспечивает редактор связей.

Доступ к глобальным переменным для случая позиционно-независимого кода реализуется через дополнительную косвенность посредством считывания адреса ссылки из таблицы GOT модуля. Доступ к элементу таблицы GOT осуществляется по адресу, вычисленному как динамически полученный адрес процедуры использования плюс статически известное смещение до таблицы GOT плюс статически известное смещение нужной ссылки в таблице. Инициализация таблиц GOT модулей производится при загрузке задачи динамическим редактором связей. Обращение к данным реализуется через семейство операций LDGD/STGD.

Доступ к глобальным переменным в режиме 64

Модель обращения к переменным аналогична режиму 32-х разрядной адресации. Все вышеизложенное остаётся верным за исключением того, что используется доступ по целому, и обращение к данным реализуется через семейство операций LD/ST.

Локальные статические данные

Размещение статических локальных переменных подчиняется тем же правилам, что и для глобальных переменных.

Доступ к локальным статическим переменным осуществляется как доступ к собственным данным модуля.

Константы

Множество констант может быть логически разделено на подмножество символьных констант (строк) и подмножество массивов констант. Каждое подмножество может иметь свое размещение. Распределение возможно либо в сегменте TEXT, либо в сегменте DATA. При размещении действуют правила выравнивания для глобальных переменных.

Доступ к константам такой же, как доступ к собственным данным, за следующим исключением: если константы распределены в сегменте TEXT, то доступ осуществляется операциями LDCUD (режим 32-х разрядной адресации).

Динамически выделенные объекты

Динамически выделенные объекты должны иметь выравнивание 16 байт.

Обращение к динамическим объектам в режиме 32-х разрядной адресации осуществляется операциями LDGD/STGD, в режиме 64 — LD/ST.

Локальные автоматические переменные

Локальные автоматические переменные выделяются в программном стеке. Механизмы и правила работы с программным стеком будут даны ниже в соответствующем разделе. Здесь отметим, что выделение памяти в стеке удовлетворяет выравниванию на 16 байт. Обращение к объектам в стеке в режиме 32-х разрядной адресации осуществляется операциями LDGD/STGD, в режиме 64-х разрядной адресации — LD/ST.

Представление данных

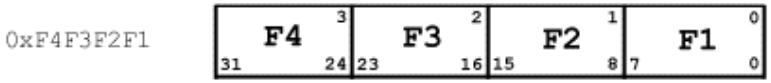
Аппаратно поддерживается работа со следующими форматами данных:

- Байт (UB) – беззнаковое целое размера 8 разрядов.
- Знаковый байт (SB) - целое со знаком размера 8 разрядов (знаковый разряд и 7 значащих разрядов).
- Полслова (UH) - беззнаковое целое размера 16 разрядов.
- Знаковые полслова (SH) - целое со знаком размера 16 разрядов (знаковый разряд и 15 значащих разрядов).

- Слово (UW) – беззнаковое целое размера 32 разряда.
- Знаковое слово (SW) – целое со знаком размера 32 разряда (знаковый разряд и 31 значащий разряд).
- Двойное слово (UD) – беззнаковое целое размера 64 разряда.
- Знаковое двойное слово (SD) – целое со знаком размера 64 разряда (знаковый разряд и 63 значащих разряда).
- Квадро слово (NQ) – числовое значение размера 128 разрядов.
- Вещественное число (FW) – вещественное значение в формате IEEE single precision.
- Вещественное число удвоенной точности (FD) - вещественное значение в формате IEEE double precision.
- Расширенное вещественное число (FX) - вещественное значение в формате IEEE double-extended precision.
- Двойной дескриптор (DD) – адресное значение размера 64 разряда.
- Квадро дескриптор (DQ) – адресное значение размера 128 разрядов.

Упаковка меньших форматов в большие соответствует little endian. На следующем рисунке для числа 0xF4F3F2F1 проиллюстрировано соответствие нумерации битов и байтов.

Рисунок D-1. Правило упаковки значения.



Правила упаковки значения

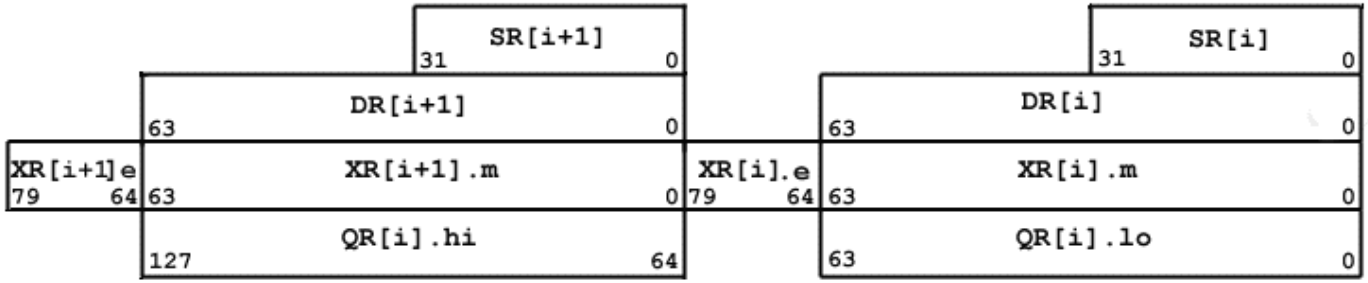
Здесь F1 – младший значащий байт, содержащий младшую часть значения. Цифры сверху показывают порядок байтов, цифры снизу – нумерацию битов.

При работе программы данные располагаются как в памяти, так и на рабочих регистрах. Рабочие регистры имеют следующие форматы:

- Регистр (SR) – 32-х разрядный регистр.
- Двойной регистр (DR) – 64-х разрядный регистр.
- Расширенный регистр (XR) – 80-и разрядный регистр.
- Квадро регистр (QR) – 128-и разрядный регистр.

Правило соответствия вложенности и нумерации рабочих регистров продемонстрировано на следующем рисунке.

Рисунок D-2. Правило соответствия вложенности рабочих регистров.



Правила соответствия вложенности рабочих регистров

SR[i]
: регистр с номером i, под которым подразумевается четное число (i = 2*n).

SR[i+1]
: регистр со следующим, нечетным номером.

XR[i].m
: поле расширенного регистра, содержащее 64 младших разряда (мантисса).

XR[i].e
: поле расширенного регистра, содержащее 16 старших разрядов (экспонента).

QR[i].lo
: младшая часть квадро регистра.

QR[i].hi
: старшая часть квадро регистра.

Одинарный регистр наложен на двойной с совпадением младших разрядов. Операциям, работающим в формате 32, старшая часть регистра недоступна. Двойные регистры накладываются на квадро регистры таким образом, что регистр с четным номером соответствует младшей половине охватывающего квадро регистра, а регистр с нечетным номером – старшей половине. Старшая часть расширенного регистра не наложена ни на какие регистры, и доступна только подмножеству

операций вещественной арифметики расширенной точности. Младшая часть расширенного регистра доступна как двойной регистр.

Отображение языковых типов данных на архитектурные форматы и соответствующие размеры занимаемых ресурсов зависят от семантической модели.

Отображение целых типов¶

Отображение целых типов для режима 32-х разрядной адресации приведено в таблице.

Отображение целых типов режима 32-х разрядной адресации¶

Тип	Формат представления	Отображение в памяти (размер)	Отображение в памяти (выравнивание)	Отображение на регистрах
char	SB	1	1	SR
signed char	SB	1	1	SR
unsigned char	UB	1	1	SR
short	SH	2	2	SR
signed short	SH	2	2	SR
unsigned short	UH	2	2	SR
int	SW	4	4	SR
signed int	SW	4	4	SR
enum	SW	4	4	SR
unsigned int	UW	4	4	SR
long	SW	4	4	SR
unsigned long	UW	4	4	SR
long long	SD	8	8	SR
unsigned long long	UD	8	8	SR
__int128	NQ	16	16	SR

Отображение целых типов для режима 64-х разрядной адресации приведено в таблице.

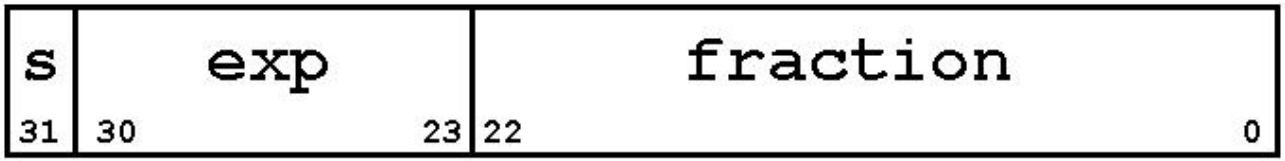
Отображение целых типов режима 64-х разрядной адресации¶

Тип	Формат представления	Отображение в памяти (размер)	Отображение в памяти (выравнивание)	Отображение на регистрах
char	SB	1	1	SR
signed char	SB	1	1	SR
unsigned char	UB	1	1	SR
short	SH	2	2	SR
signed short	SH	2	2	SR
unsigned short	UH	2	2	SR
int	SW	4	4	SR
signed int	SW	4	4	SR
enum	SW	4	4	SR
unsigned int	UW	4	4	SR
long	SD	8	8	DR
unsigned long	UD	8	8	DR
long long	SD	8	8	DR
unsigned long long	UD	8	8	DR
__int128	NQ	16	16	QR

Отображение вещественных типов¶

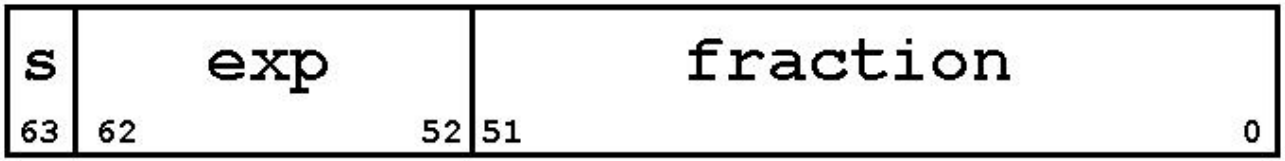
Вещественные типы представляются в соответствии со стандартом вещественной арифметики ANSI/IEEE 754-1985. Аппаратно поддерживается работа с тремя форматами: простой формат (single), формат удвоенной точности (double) и расширенный формат (extended). Представление форматов вещественных данных приведено на рисунках ниже.

Рисунок D-13. Простой вещественный формат.



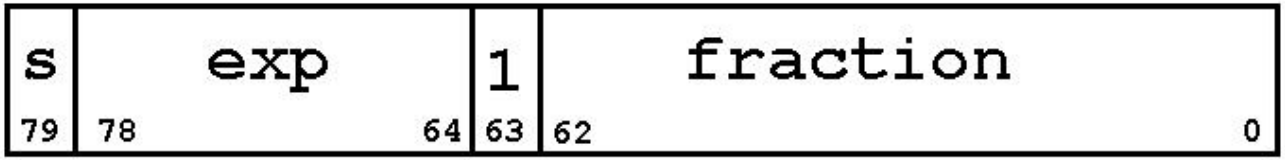
Простой вещественный формат.

Рисунок D-14. Вещественный формат удвоенной точности.



Вещественный формат удвоенной точности

Рисунок D-15. Расширенный вещественный формат.



Расширенный вещественный формат

Отображение вещественных типов для всех режимов приведено в таблице.

Отображение вещественных типов¶

Тип	Формат представления	Отображение в памяти (размер)	Отображение в памяти (выравнивание)	Отображение на регистрах
float	FW	4	4	SR
double	FD	8	8	DR
__float80	FX	16	16	XR
__float128	NQ	16	16	QR
long double	FX	16	16	XR

Отображение указательных типов¶

Отображение указательных типов для режима 32 приведено в таблице.

Отображение указательных типов для режима 32-х разрядной адресации¶

Тип	Формат представления	Отображение в памяти (размер)	Отображение в памяти (выравнивание)	Отображение на регистрах
any_type*	UW	4	4	SR
any_type(*)()	UW	4	4	SR

Отображение указательных типов для режима 64 приведено в таблице.

Отображение указательных типов для режима 64-х разрядной адресации¶

Тип	Формат представления	Отображение в памяти (размер)	Отображение в памяти (выравнивание)	Отображение на регистрах
any_type*	UD	8	8	DR
any_type(*)()	UD	8	8	DR

Агрегатные типы¶

Агрегатные типы включают в себя структуры (struct), объединения (union), классы (class) и массивы. Под структурами и объединениями понимаются типы в нотации языка C: struct и union соответственно. Под классом понимаются типы языка C++: class, struct и union. Они имеют свойства, не имеющие аналогов в типах struct и union языка C.

Выравнивание объектов агрегатных типов должно соответствовать выравниванию их наиболее строго выровненных компонентов. Размер объекта агрегатного типа должен быть кратен выравниванию наиболее строго выровненного компонента. Для структур и объединений это может потребовать расширения размера пустыми полями (паддинг). Значение полей паддинга не определено.

Тип массива¶

Выравнивание объекта типа массив определяется выравниванием элемента этого массива. Размер объекта типа массив равен размеру элемента массива, умноженному на число элементов массива.

Тип структуры¶

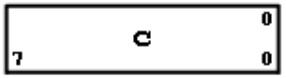
Выравнивание и размер объекта типа структуры определяются правилами упаковки членов (полей) этой структуры. Правила упаковки полей для структурных типов:

- Объект типа структуры должен иметь выравнивание не хуже выравнивания наиболее строго выровненного компонента.
- Поля упаковываются в структуру по порядку так, что очередное поле получает наименьшее возможное смещение от начала структуры, удовлетворяющее его выравниванию. Это может привести к возникновению внутреннего паддинга, когда требуется пропустить место для размещения очередного компонента, если текущее смещение не соответствует требуемому выравниванию.
- Размер структуры должен быть увеличен, если суммарный размер всех полей, включая внутренний паддинг, не соответствует кратности выравнивания. Это приводит к возникновению хвостового паддинга.

Нижеприведенные рисунки иллюстрируют действие правил упаковки полей для структур.

Рисунок D-3. Простая маленькая структура

```
struct {
    char   c;
};
```

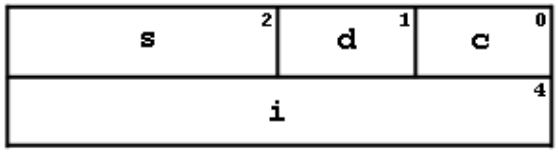


Простая маленькая структура

Размер структуры – 1 байт. Выравнивание – 1 байт (не требуется).

Рисунок D-4. Плотно упакованная структура без паддинга

```
struct {
    char    c;
    char    d;
    short   s;
    int     i;
};
```

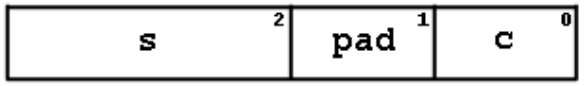


Плотно упакованная структура без паддинга

Размер структуры – 8 байт. Выравнивание определяет поле i – 4 байта.

Рисунок D-5. Структура с внутренним паддингом

```
struct {
    char    c;
    short   s;
};
```

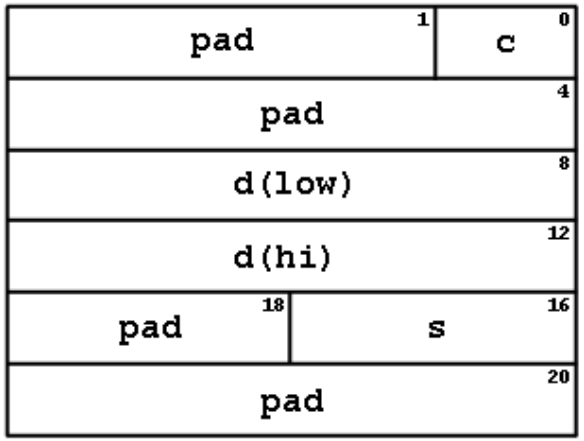


Структура с внутренним паддингом

Размер структуры – 4 байта. Выравнивание определяет поле s - 2 байта. Поле s не может быть размещено сразу после поля c, поскольку это не соответствует его выравниванию. Поэтому в байте 1 возникает поле паддинга.

Рисунок D-6. Структура с внутренним и хвостовым паддингами

```
struct {
    char    c;
    double  d;
    short   s;
};
```



Структура с внутренним и хвостовым паддингами

Размер структуры – 24 байта. Выравнивание определяет поле d - 8 байт. Между полем c и полем d, в байтах с 1 по 7, из-за требования выравнивания поля d появляется внутренний паддинг. После распределения последнего поля s, размер структуры равен 18 байтам, что не кратно выравниванию в 8 байт. Поэтому размер структуры увеличен до 24 байт. В байтах с 18 по 23 находится поле хвостового паддинга.

Тип объединение¶

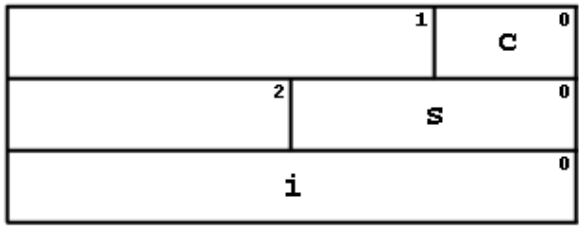
Выравнивание и размер объекта типа объединение определяются правилами упаковки членов (полей) типа объединение. Правила упаковки полей для типа объединение:

- Выравнивание объекта типа объединение должно быть не хуже, чем у поля с наиболее строгим выравниванием.
- Поля упаковываются в объединение так, что начала всех полей совпадают и равны началу объединения.
- Размер объекта типа объединения должен быть не меньше размера максимального поля и кратен выравниванию. Если размер максимального поля не кратен выравниванию, то размер объединения увеличивается добавлением поля хвостового паддинга.

Нижеприведенные рисунки иллюстрируют правила упаковки полей для объединений.

Рисунок D-7. Объединение

```
union {
   char   c;
   shopt  s;
   int    i;
};
```

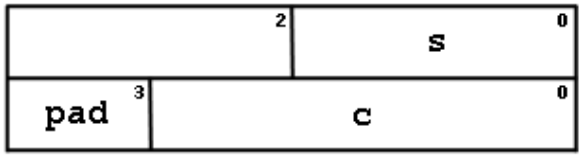


Объединение

Размер объединения – 4 байта. Выравнивание – 4 байта.

Рисунок D-8. Объединение

```
union {
   short   s;
   char    c[3];
};
```



Объединение с хвостовым паддингом

Размер объединения – 4 байта. Выравнивание – 2 байта. Размер максимального поля – 3 байта, что не кратно выравниванию. Поэтому размер объединения увеличен добавлением в 3 байте поля паддинга.

Битовые поля¶

Битовые поля являются членами объекта типа структуры, класса или объединения с заданным в виде числа разрядов размером. Битовые поля определяются базовым типом и числом разрядов. Число разрядов не может быть больше, чем число разрядов в базовом типе. Базовым типом может быть любой целочисленный тип знаковой или беззнаковой модификации. Область памяти, определенная базовым типом, в которой располагается битовое поле, называется контейнером.

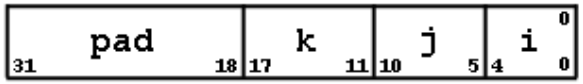
Битовые поля подчиняются тем же правилам упаковки, что и не битовые поля с некоторыми добавлениями:

- Битовые поля располагаются справа налево от менее значащих разрядов к более значащим.
- Контейнер битового поля должен быть размещен в соответствии с правилами размещения базового типа.
- В контейнере битового поля могут размещаться другие, в том числе и не битовые поля.
- Неименованные битовые поля не участвуют в определении общего выравнивания объекта.
- Неименованные битовые поля ненулевой длины используются для явного задания паддинга.
- Неименованные битовые поля нулевой длины используются для принудительного выравнивания последующего поля на границу, соответствующую базовому типу этого битового поля.

Следующие рисунки иллюстрируют правила упаковки битовых полей.

Рисунок D-9. Структура с битовыми полями

```
struct {
    int i:5;
    int j:6;
    int k:7;
};
```

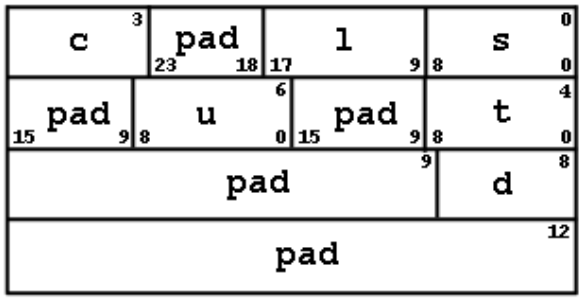


Структура с битовыми полями

Размер структуры – 4 байта. Выравнивание – 4 байта, определяется базовым типом полей, который для всех один. Структура дополняется хвостовым паддингом для того, чтобы размер структуры удовлетворял кратности выравнивания.

Рисунок D-10. Выравнивание в структуре с битовыми полями

```
struct {
    short     s:9;
    long long l:9;
    char      c;
    short     t:9;
    short     u:9;
    char      d;
};
```



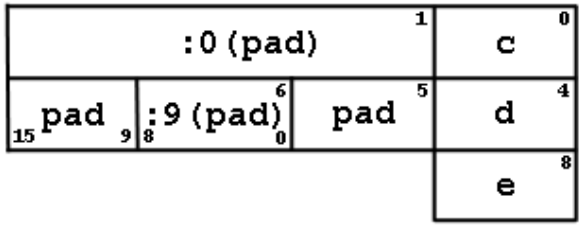
Выравнивание в структуре с битовыми полями

Размер структуры – 16 байт. Выравнивание – 8 байт. Выравнивание определяет базовый тип битового поля l. Контейнер для этого поля может быть размещен, начиная с нулевого байта. Но в этот контейнер уже попадает предыдущее поле s. Следующее поле c требует выравнивание на 1 байт. Стало быть, оно может быть размещено только начиная с 24 бита. В битах с 18 по 23 остается поле внутреннего паддинга. По этой же причине возникает внутренний паддинг между полями u и d.

Между полями t и u причина возникновения паддинга следующая. Если контейнер для битового поля u разместить, начиная с 4 байта, а поле начать размещать сразу после поля t, то оно не поместится в отведенный контейнер. Поэтому контейнер для этого поля размещается со следующей границы его выравнивания. После распределения всех полей размер структуры не соответствует кратности выравнивания, что требует добавления хвостового паддинга.

Рисунок D-11. Структура с неименованными битовыми полями нулевой длины

```
struct {
    char   c;
    int     :0;
    char   d;
    short   :9;
    char   e;
    char    :0;
};
```



Структура с неименованными битовыми полями нулевой длины

Размер структуры – 9 байт. Выравнивание – 1 байт. Неименованные битовые поля не участвуют в определении выравнивания. Все остальные поля имеют тип, требующий выравнивания на 1 байт. Неименованное битовое поле нулевой длины перед полем d требует выравнивания поля d в соответствии со своим базовым типом int, что приводит к возникновению паддинга с 1 по 3 байт. Неименованное битовое поле длины 9 не может быть размещено сразу после поля d: если разместить контейнер, начиная с 4 бита, поле в него не поместится. Следующее удовлетворяющее выравниванию размещение контейнера для этого битового поля – с 6 байта. Поэтому возникает внутренний паддинг в 5 байте. Само по себе это битовое поле тоже приводит к внутреннему паддингу в 9 бит.

9.3. Описание регистров¶

Существуют следующие группы программно доступных регистров:

- рабочие регистры.
- предикатные регистры.
- регистры управления.
- специальные регистры.

Характеристики и назначение каждой группы приведены ниже.

9.3.1. Рабочие регистры¶

Основное назначение рабочих регистров – расположение данных для вычислительных операций. Из рабочих регистров в операции поступают входные данные. После проведения вычисления результат сохраняется в рабочих регистрах.

Рабочие регистры составляют регистровый файл. Размер регистрового файла - 256 регистров. Нумерация регистров и их форматы приведены выше на рисунке D-2. Регистровый файл разделен на две части: глобальную и стековую. Глобальная область находится в верхней части регистрового файла (младшие номера) и состоит из 32 регистров. Остальные 224 регистра составляют стековую часть регистрового файла. Глобальная часть регистрового файла доступна во всех процедурах и не участвует в процедурных механизмах. Стековая область регистрового файла используется в процедурных механизмах и может быть аппаратно откачана в память или загружена из памяти.

Обращение в регистровый файл к рабочим регистрам реализуется с помощью нескольких механизмов: абсолютной адресацией в регистровый файл и адресацией относительно специального регистра.

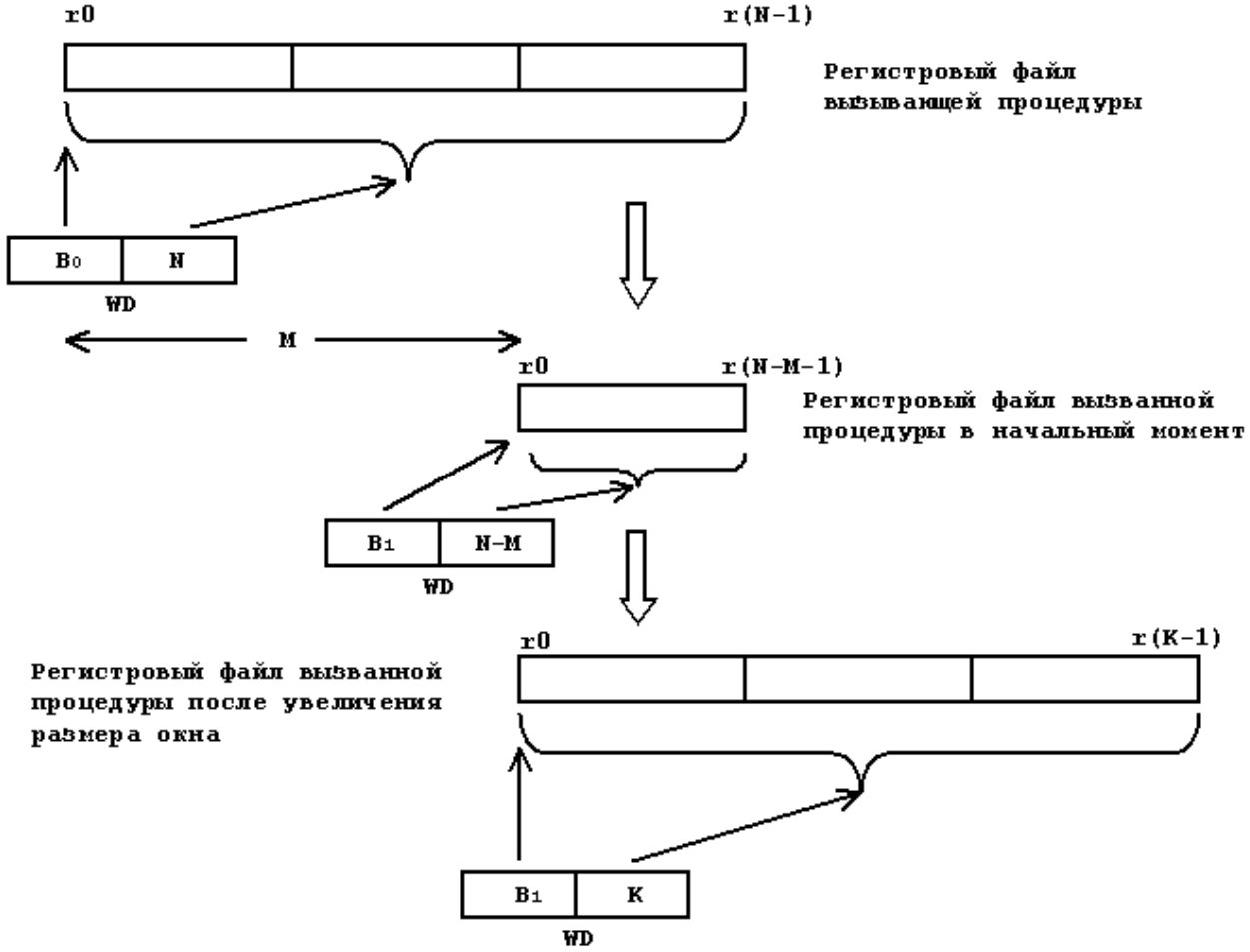
9.3.1.1. Механизм регистровых окон¶

Механизм регистровых окон является важной частью процедурного механизма. Регистровые окна организуются в стековой части регистрового файла. Для этого используется регистр текущего регистрового окна WD. В регистре WD содержится базовый абсолютный адрес начала области (регистровое окно) в регистровом файле и размер этой области.

При процедурном вызове происходит изменение содержимого регистра WD следующим образом. Новое состояние базового адреса может быть установлено вызывающей процедурой в любое место ее регистрового окна. Новый размер устанавливается как разность размера вызывающей процедуры и размера области от начала окна вызывающей процедуры до нового значения базового адреса. Область регистрового файла нового окна доступна и вызванной, и вызывающей процедуре. Эта

область используется для передачи параметров и возврата значения (область параметров). Процедурный механизм смены окна проиллюстрирован на рисунке.

Рисунок R-1. Процедурное переключение окна



Процедурное переключение окна

Вызванная процедура может изменить переданное ей окно по своему усмотрению. Для изменения размеров окна используется операция **SETWD**. С помощью этой операции можно увеличить или уменьшить размер текущего регистрового окна. Но размер текущего окна нельзя установить меньше, чем область для параметров у вызывающей процедуры (размер N-M на рисунке). Расширенная часть окна не будет доступна вызвавшей процедуре.

9.3.1.2. Пространство регистров текущего окна. Программные соглашения использования пространства регистров текущего окна¶

Пространство регистров текущего окна реализовано в стековой части регистрового файла. Для обращения к пространству регистров текущего окна используется адресация относительно регистра WD. Адресом (номером регистра) является смещение относительно базового адреса, которое хранится в регистре WD. При попытке обратиться за пределы регистрового окна (номер регистра больше размера, заданного в WD) возникает прерывание.

Максимально возможный размер регистрового окна определяется размером стековой области регистрового файла - 224 регистра. Максимальный размер пространства регистров текущего окна определяется кодировкой операций и равен 64 регистрам. Следовательно, в общем случае пространство регистров текущего окна может не накрывать все регистровое окно.

Ассемблерная мнемоника обращения к регистрам текущего окна:

```
%r[number]   - SR регистр
 %dr[number]  - DR регистр
 %qr[number]  - QR регистр
```

где number – номер регистра относительно базового в регистре WD (0 – размер из регистра по модулю 64).

Регистры текущего окна используются для получения параметров процедуры, размещения локальных и рабочих переменных, возврата процедурой результата.

Подробно соглашение о передаче параметров и возврате результата описано в разделе Процедурный механизм. Механизм регистровых окон обеспечивает возможность заведения локальных и рабочих переменных как автоматически сохраняемых и восстанавливаемых при процедурных вызовах. Это будет обеспечено только при условии того, что переменные не будут размещаться в области передачи параметров. Регистры области параметров могут быть изменены вызовом процедуры.

9.3.1.3. Пространство регистров подвижной базы. Программные соглашения использования базированных регистров¶

В текущем окне процедуры с помощью специального регистра BR может быть выделено отдельное подпространство со своей относительной адресацией. Регистр BR называется регистром подвижной базы. Организованное таким образом пространство регистров и есть регистры подвижной базы, или базированные регистры.

Регистр BR устанавливается относительно регистра WD. Максимально возможное смещение установки - 128 регистров. Максимальный размер области, описываемой регистром BR, составляет 128 регистров. Таким образом, с использованием регистров подвижной базы можно организовать доступ к регистрам текущего окна, недоступным адресацией относительно WD.

Ассемблерная мнемоника обращения к базированным регистрам:

```
%b[number]      - SR регистр
 %db[number]     - DR регистр
 %qb[number]     - QR регистр
```

где number – номер регистра относительно начала базированной области.

В пространстве базированных регистров аппаратно поддержан механизм вращения.

Поскольку базированные регистры являются подпространством текущего регистрового окна, то на них распространяются все свойства, определенные процедурным механизмом. Если базированные регистры располагаются в области параметров, то их значение может быть изменено вызовом процедуры. Если они находятся вне области параметров, то их значение сохраняется после вызова процедуры.

Особенностью использования базированных регистров является произвольная (определенная пользователем) установка области в текущем регистровом окне. Это означает, что в процедуре может быть произвольное количество пространств базированных регистров. Новая установка значения регистра BR означает возникновение нового пространства и недоступность старого. При восстановлении значения регистра BR в старое значение сохранение значений базированных регистров зависит от того, пересекались ли области пространств или нет.

Наличие механизма вращения позволяет использовать базированные регистры в цикловых оптимизациях.

9.3.1.4. Пространство глобальных регистров. Программные соглашения использования глобальных регистров¶

Глобальные регистры реализованы на глобальной части регистрового файла. Адресация к глобальным регистрам осуществляется с помощью абсолютной адресации (при обращении адрес вычисляется по модулю 32). В старшей части адресов (с 24 по 31) глобальных регистров аппаратно поддержан механизм вращения.

Ассемблерная мнемоника обращения к глобальным регистрам:

```
%g[number]  - SR регистр
 %dg[number] - DR регистр
 %qg[number] - QR регистр
```

где number – абсолютный адрес в регистровом файле (0-31).

Глобальный регистр g13 используется для указателя TLS, g12 зарезервирован для дальнейших нужд, остальные используются в качестве scratch-регистров (не сохраняют значения при вызовах).

Поведение ОС:

- регистры g12, g13 являются глобальными для потока, т.е. сохраняются/восстанавливаются только при переключении контекста;
- все остальные регистры дополнительно сохраняются/восстанавливаются при входе в пользовательский обработчик сигналов.

Использование в приложении некоторых глобальных регистров для хранения глобальных переменных включается по опции компилятора, и возможно только при условии однопоточности приложения и отсутствия пользовательских обработчиков сигналов.

9.3.2. Предикатные регистры¶

Предикатные регистры используются для управления вычислениями. Предикатные регистры могут содержать два значения: TRUE (1) или FALSE (0). Выполнение операций в условном режиме ставится в зависимость от значения предиката. Операция либо выполняется, либо не выполняется. Предикатные регистры располагаются в предикатном регистровом файле. Размер предикатного файла – 32 регистра. Адресация предикатных регистров осуществляется указанием абсолютного номера регистра в предикатном файле.

Ассемблерная мнемоника обращения к предикатным регистрам:

```
%pred[number]
```

где number - абсолютный номер регистра в предикатном файле (0 – 31).

При процедурных переходах весь предикатный регистровый файл сохраняется и при возврате восстанавливается. Таким образом, предикатные регистры являются локальными автоматическими объектами.

В предикатном файле может быть выделена область с вращающимся механизмом. Для этого требуется установить поле начала области и её размер в регистре BR.

9.3.3. Регистры управления¶

Регистры управления используются для организации переходов, в том числе и процедурных. В них содержится информация о типе перехода и адресе назначения. Формирование регистров управления осуществляется в операциях подготовки переходов. Использование – в операциях переходов. Регистры управления являются специальными регистрами и могут быть доступны по чтению и записи (только в привилегированном режиме) для операций доступа к специальным регистрам.

Существует три регистра управления. Ассемблерная мнемоника обращения к регистрам:

```
%ctpr[number]
```

где number может принимать значения 1,2,3.

Все типы перехода за исключением перехода типа RETURN (возврат из процедурного вызова) могут использовать любой из этих регистров управления. Возврат из процедурного вызова реализуется только на %ctpr3.

При процедурных переходах значения регистров не сохраняются.

9.3.4. Специальные регистры¶

Под специальными регистрами понимаются регистры процессора, используемые для организации вычислительного процесса. Здесь не будут рассматриваться все такие регистры, имеющиеся в архитектуре. В рассмотрение включены только регистры, доступные непривилегированной пользовательской задаче.

В нижеследующей таблице приведены описания регистров. Программные соглашения отражены в требованиях по сохранению значения регистров:

- Auto. Регистры автоматически сохраняются и восстанавливаются при процедурных вызовах.
- Scratch. При процедурных вызовах значение регистров не сохраняется, поэтому перед вызовом значение регистров должно быть сохранено, чтобы иметь возможность восстановить их значение после вызова (если это необходимо).
- Special. Предполагается, что процедурный вызов не портит значение регистра. Соответственно, если в обычной процедуре значение регистра необходимо изменить, то при возврате должно быть восстановлено исходное значение. Это, однако, не относится к специально

документированным процедурам, назначение которых состоит именно в известном изменении значения таких регистров.

С точки зрения доступа к регистру возможны варианты, которые отражены в виде следующей мнемоники: R/W/M. Позиция R означает возможность чтения содержимого регистра, W - возможность записи значения в регистр, M - возможность модификации регистра определенными командами. Если какая либо из возможностей отсутствует, в соответствующей позиции ставится прочерк.

Специальные регистры¶

Регистр	Описание	Доступ	Сохранение
WD	Описание текущего окна в регистровом файле. Сохраняется при вызове процедуры, восстанавливается при выходе из нее. В процедуре значение может быть изменено операцией SETWD.	R/-/M	Auto
BR	Регистр подвижной базы в текущем окне регистрового файла. Сохраняется при вызове процедуры, восстанавливается при выходе из нее. В процедуре значение устанавливается операцией SETBN.	R/-/M	Auto
TR	Регистр текущего типа. Содержит абсолютный номер типа. Должен быть установлен для процедуры-метода в соответствии с классом этого метода. Для процедур, не являющихся методами, значение регистра	R/-/M	Auto

Регистр	Описание	Доступ	Сохранение
	равно 0. Устанавливается операцией SETTR.		
PSR	Регистр состояния процессора. Содержит флаги, управляющие работой процессора. В частности, в регистре содержится признак привилегированного режима. Регистр недоступен для модификации в непривилегированном режиме.	R/-/-	?
UPSR	Содержит некоторые флаги, управляющие работой процессора и доступные пользовательской задаче.	R/W/-	Special
IP	Адрес текущей команды. Изменяется в процессе вычислений и выполнения переходов.	R/-/M	Auto
NIP	Адрес следующей команды.	R/-/M	Auto
CTPR[1-3]	Регистры подготовки переходов. Не сохраняются при процедурных переходах. Изменяются операциями подготовки переходов.	R/-/M	Scratch
PFPFR	Статусный регистр вещественной	R/W/M	Special

Регистр	Описание	Доступ	Сохранение
	арифметики упакованных значений. Содержит маски прерываний, накапливаемые флаги и правила вычислений. Накапливаемые флаги могут изменяться операциями вещественной арифметики упакованных значений.		
FPFR	Статусный регистр вещественной арифметики. Содержит маски прерываний, накапливаемые флаги и правила вычислений. Накапливаемые флаги могут изменяться операциями вещественной арифметики.	R/W/M	Special
LSR	Статусный регистр цикла.	R/W/-	Scratch
ILCR	Регистр счетчика цикла.	R/W/-	Scratch
USD	Регистр указателя пользовательского стека в памяти. Модифицируется операциями заказа локальной памяти процедуры GETSP. Сохраняется и восстанавливается в процедурном механизме.	R/-/M	Auto

Регистр	Описание	Доступ	Сохранение
CUD	Сегментный регистр кода. Содержит базовый адрес текущего сегмента кода и размер области. При межмодульных процедурных переходах автоматически сохраняется и восстанавливается.	R/-/-	Auto
GD	Сегментный регистр данных. Содержит базовый адрес текущего сегмента данных и его размер. При межмодульных процедурных переходах автоматически сохраняется и восстанавливается.	R/-/-	Auto
TSD	Регистр типов текущего модуля. Содержит абсолютный номер первого типа модуля и число типов модуля. При межмодульных процедурных переходах автоматически сохраняется и восстанавливается.	R/-/-	Auto
CUIR	Регистр содержит индекс текущего загрузочного модуля (current compilation unit)	R/-/-	Auto

9.4. Локальный стек¶

Локальный стек в памяти используется для размещения автоматических локальных переменных процедуры, сохранения локальных рабочих данных процедуры, механизма передачи параметров. Организуется как последовательность соответствующих процедурам фрагментов, начинающихся с процедуры main в глубине стека, и растущих в направлении активации текущей процедуры в верхушке этого стека.

Стек в памяти начинается с адреса, определенного операционной системой, и растет в направлении уменьшения адресов. В свободную часть стека всегда указывает регистр USD, что соответствует наименьшему адресу фрагмента стека текущей процедуры.

Продвижение стека поддержано аппаратно. Для продвижения стека вперед (заказ памяти в стеке текущей процедуры) используются операции GETSP, GETSAP и GETSOD. Значение регистра USD продвигается этими операциями в соответствии с размером выделенной памяти. При возврате из процедуры автоматически восстанавливается прежнее значение регистра USD. Значение базового адреса из регистра USD может быть использовано в регулярных режимах как указатель стека.

В регулярном режиме для заказа памяти в стеке используется операция GETSP. Операции аргументом подается требуемый размер в байтах. Операция возвращает базовый адрес заказанной области, выровненный на 16 байт.

9.4.1. Процедурный фрагмент стека¶

Логически процедурный фрагмент стека может состоять из нескольких областей. Наличие той или иной области определяется её необходимостью для процедуры: нужно ли разместить данные или обеспечить размещение данных в вызываемых процедурах. Однако при использовании для обращения к данным указателя стека (регулярные режимы) появление областей возможно только в представленном на рисунке порядке. Размер любой области, если она появляется в процедуре, должен быть кратен 16 байтам.

Рисунок S-1. Области стека процедурного фрагмента

Области стека процедурного фрагмента

В области локальных переменных могут быть размещены собственно локальные переменные процедуры и рабочие ячейки, используемые процедурой. Область является недоступной для вызванных процедур. Хотя наличие этой области не является обязательным, обычно эта область всегда присутствует в процедурном фрагменте стека. Отсутствие этой области означает, что процедуре не требуется размещать никакие локальные данные в памяти.

Динамическая область предназначена для динамического выделения памяти процедуры, например, для реализации стандартной библиотечной функции alloca. Наличие этой области является необязательным.

Область фактических параметров служит для передачи параметров вызываемой процедуре. Соглашения об использовании этой области будут приведены в разделе Процедурный механизм. Область может отсутствовать:

- в процедурах без вызовов;
- в процедурах с вызовами, которые не требуют размещения параметров в памяти.

В процедурах, имеющих вызовы, для которых требуется передача параметров через память, а также для процедур с неочевидным интерфейсом передачи параметров, эта область является обязательной. Размер этой области должен быть таков, чтобы ее размер удовлетворял всем возможным вызовам процедуры. Поскольку область является доступной для вызванных процедур, вся информация, расположенная в ней, может не сохраняться при вызовах.

Зарезервированная область предназначена для отображения в памяти параметров, передаваемых через регистры. Соответственно, ее размер определяется размером области регистрового файла для передачи параметров. Наличие или отсутствие этой области определяется интерфейсом вызываемых

из процедуры функций. Для процедур без вызовов и процедур с вызовами, где гарантированно отсутствуют параметры, область может отсутствовать.

9.4.2. Доступ к компонентам фрагмента процедурного стека¶

При входе в процедуру в регистре USD содержится базовый адрес свободной части стека (указатель стека). В сторону увеличения адресов от этого значения находится область стека, соответствующая формальным параметрам этой процедуры (фактические параметры вызывающей процедуры). Обращение к формальным параметрам может быть реализовано либо по смещению относительно сохраненного начального значения указателя стека, либо по смещению относительно текущего значения. При этом, если в процедуре происходит динамическое выделение памяти в стеке, требуется динамическое вычисление смещения, что может оказаться неэффективным. Отметим, что обращение к формальным параметрам происходит с положительным смещением.

После выделения памяти в стеке для процедуры обращение к локальным переменным (в области локальных переменных) возможно либо относительно старого значения указателя стека, либо относительно текущего значения. Относительно старого значения указателя смещение будет отрицательным. Относительно текущего значения смещение будет положительным, но в связи с необходимостью пересчета при динамическом заказе памяти способ может оказаться неэффективным.

Область фактических параметров должна быть доступна в вызванной процедуре, которая будет иметь возможность получения текущего значения указателя стека. Поэтому между заполнением значениями области фактических параметров для вызова и самим вызовом не должно быть операций заказа памяти в стеке.

9.5. Процедурный механизм¶

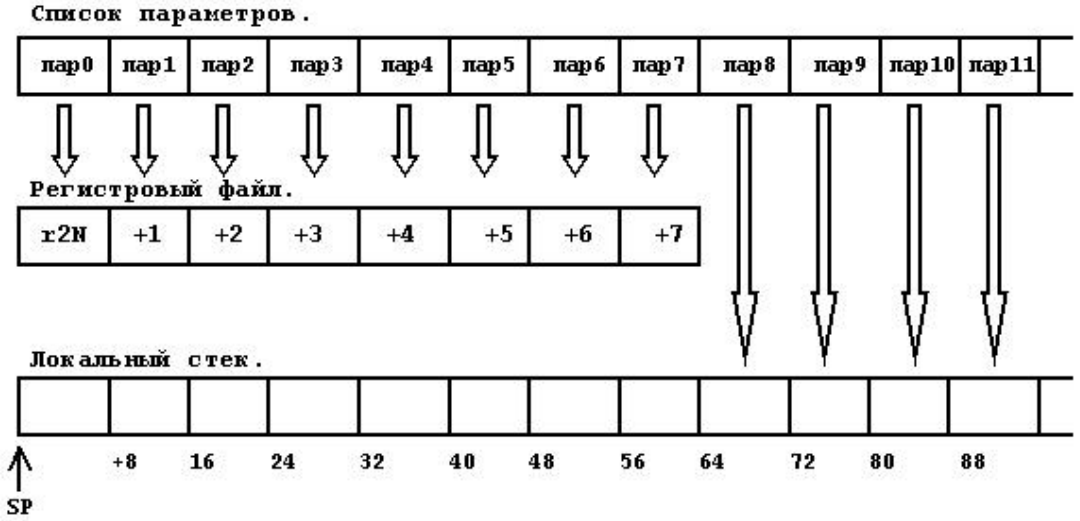
К процедурному механизму относится обеспечение процедурных переходов, способы передачи параметров и возврата результата вызова. Отметим, что в этом разделе не обсуждаются вопросы вызовов функций операционной системы.

9.5.1. Передача параметров¶

Параметры передаются через регистровое окно и через локальный стек в памяти. Список параметров формируется в виде массива с размером элемента 8 байт. Каждый параметр размера 8 байт или меньше размещается в одном элементе списка параметров. Параметры большего размера размещаются в необходимом количестве последовательных элементов. В любом случае в одном элементе списка параметров может находиться только один параметр или часть одного параметра.

Размещение списка параметров на физические ресурсы осуществляется следующим образом. Первые 8 элементов размещаются в регистровом файле. При вызове должно быть обеспечено переключение регистра WD на регистр, в котором находится первый параметр. Это обеспечивается установками операции процедурного перехода. Регистр, в который помещается значение первого параметра, должен иметь четный номер, т.е. соответствовать выравниванию на квадро регистр. После выполнения процедурного перехода этот регистр будет иметь номер 0. Остальные элементы списка параметров передаются через область фактических параметров локального стека в памяти.

Рисунок P-1. Размещение списка параметров на физические ресурсы



Размещение списка параметров на физические ресурсы

Отметим, что хотя первые 8 параметров передаются через регистровый файл, в локальном стеке для них резервируется место. Потому, если возникает необходимость работы с параметром в памяти (например, в случае взятия адреса на формальный параметр в вызываемой процедуре), этот параметр может быть откачан в зарезервированное место в локальном стеке.

Отметим также, что проиллюстрированный на рисунке P-1 способ отображения списка параметров на физические ресурсы является общей схемой, в которую могут быть внесены дополнения при наличии или отсутствии информации об интерфейсе передачи параметров конкретного вызова. Зависимость передачи параметров от интерфейса процедур приведена ниже.

Зависимость передачи параметров от интерфейса процедуры

При генерации кода компилятор вправе использовать и доверять информации об интерфейсе процедуры в точке ее вызова. Эта информация получается из заданного предописания процедуры. Возможен и анализ по вызову при отсутствии предописания. Все предописания можно разделить на три группы:

- предописание со спецификацией всех параметров;
- предописание со спецификацией переменного числа параметров;
- предописание без спецификации параметров.

Передача параметров для вызова со спецификацией всех параметров осуществляется по общей схеме, приведенной выше.

Интерфейс обработки списка переменного числа параметров подразумевает нахождение их в памяти. Поэтому при передаче параметров для вызова процедуры с переменным числом параметров, параметры, входящие в список переменного числа (начиная с параметра перед эллипсом), сразу размещаются в соответствующие места локального стека, даже если они могут быть помещены в первые восемь регистров.

Если для вызова процедуры нет предописания со спецификацией параметров, необходимо предусмотреть все возможные случаи. Поэтому при формировании списка фактических параметров первые восемь параметров помещаются и на регистры (как в случае процедур с фиксированным числом параметров), и в память (как в случае процедур с переменным числом параметров).

Таким образом, процедура с переменным числом параметров всегда может предполагать, что переменная часть параметров находится в памяти. А для процедуры с фиксированным числом параметров первые параметры находятся в первых восьми регистрах.

Зависимость размещения параметров в списке параметров от типа

Как было указано выше, размер элемента списка параметров составляет 8 байт. Соответственно, необходимо иметь правила размещения параметров размера, отличного от 8 байт, в списке параметров. Эти правила приведены в следующей таблице.

Таблица. Правила размещения в списке параметров.

Правила размещения в списке параметров¶

Размер (байт)	Правило размещения	Число элементов списка
1-8	Следующий свободный	1
9-16	Следующий четный	2
17 и более	Следующий четный	(размер + 7)/8

Размещение «следующий свободный» означает, что параметр может быть размещен в очередном свободном элементе списка параметров. Размещение «следующий четный» означает, что параметр может быть размещен только в очередном свободном элементе списка параметров с четным номером. Если очередной свободный элемент имеет нечетный номер, то он должен быть пропущен (паддинг).

При размещении целочисленного параметра размера меньшего, чем `int`, в соответствии со стандартом языка делается расширение значения до `int`. Если параметр имеет размер меньше 8 байт, значение старших байтов не определено.

При размещении параметра скалярного типа размером больше 8 байт младшая часть параметра распределяется в элементе списка параметров с меньшим номером, старшая в элементе с большим номером.

Особо отметим, что хотя параметр типа `__float80` с точки зрения регистров может быть размещен в одном элементе списка параметров, с точки зрения размера и размещения в памяти требуется два элемента списка параметров.

Если при размещении параметра только его часть может быть размещена в регистровом файле, весь параметр должен быть размещен в памяти.

Особенности передачи параметров в режиме 64

В режиме 64 все передаваемые целочисленные параметры короче 64 бит должны расширяться до 64 бит (так называемый promotion). При этом значения знакового типа расширяются знаком, а значения беззнакового типа - нулём.

9.5.2. Возврат значения¶

Возврат значения производится либо через регистровый файл, либо через область памяти, выделенную вызывающей процедурой. В любом случае вызывающая процедура должна обеспечить необходимые ресурсы для возвращаемого значения. Если для вызова отсутствует предописание, то тип возвращаемого значения определяется по правилам языка (обычно int). Особо отметим случай вызова процедур без предописания, не использующих результат вызова. Для таких вызовов также необходимо обеспечить ресурсы для возвращаемого значения типа int.

Возврат значения размером не больше 64 байт производится через регистровый файл. Регистры для возврата значения начинаются с регистра `%r0`.

Возврат значения размером больше 64 байт производится через память области параметров. Соответственно, вызывающая процедура должна обеспечить необходимый размер области фактических параметров.

Особенности возврата значения в режиме 64

В режиме 64 возвращаемый целочисленный результат короче 64 бит должен расширяться до 64 бит (так называемый promotion). При этом значения знакового типа расширяются знаком, а значения беззнакового типа - нулём.

9.5.3. Процедурный переход¶

Процедурный переход осуществляется в два этапа. На первом этапе делается подготовка перехода, на втором собственно сам переход. На этапе подготовки перехода вычисляется адрес назначения перехода и тип перехода. На этапе перехода делается переключение контекста с сохранением необходимой информации в стеке связующей информации и переход на вычисленный адрес назначения.

Подготовка процедурного перехода

Для процедурного перехода возможны следующие подготовки:

- DISP – подготовка статически известного перехода по относительному смещению. Смещение суть разность адреса назначения и адреса операции подготовки перехода. Эта подготовка может быть использована для подготовки вызова статически известных процедур, находящихся в одном загрузочном модуле с точкой вызова. Операция подготавливает переход типа `ctpll` (переход на локальную метку).
- MOVTD – подготовка по значению. Значение, которое определяет адрес перехода, подается аргументом операции и имеет формат двойного слова. В зависимости от значения и режима подготавливаются переходы разных типов.
  - Если значение имеет диагностические теги, подготавливается переход типа `ctpdw` (переход по диагностическому значению).
  - Иначе в регулярных режимах подготавливается переход типа `ctpnl`.

- GETPL – подготовка по смещению относительно регистра CUD. Аргументом операции подается смещение адреса назначения относительно базового адреса регистра CUD. Формат аргумента - слово. В зависимости от значения и режима подготавливаются переходы разных типов.
  - Если значение имеет диагностические теги, подготавливается переход типа `ctpdw`.
  - Иначе в регулярных режимах подготавливается переход типа `ctppl`.

Все подготовки формируют код операции перехода `disp`.

Результат подготовки перехода записывается в любой из регистров управления (`%ctprN`, где N ={ 1, 2, 3}).

Выполнение процедурного перехода

Выполнение процедурного перехода реализуется операцией CALL. Операция принимает аргумент, который должен быть одним из регистров управления `ctrpN`, и параметры. Если подготовленный переход имеет тип `ctpdw` или `ctpew`, возникает прерывание.

При выполнении процедурного перехода автоматически сохраняется информация для возврата. Сохранение делается в стеке связующей информации. Сохраняются следующие значения:

- адрес следующей команды, на которую будет произведен возврат (значение регистра nIP);
- параметры текущего регистрового окна (регистры WD и BR);
- предикатный файл;
- состояние регистра CUIR;
- состояние регистра TR;
- состояние регистра USD;
- состояние регистра PSR. Сохранение этого регистра необходимо только для целей обработки прерываний операционной системой.

Формируется новое значение регистра IP, которое вычислено в операции подготовки. Это значение находится в регистре управления, который подаётся аргументом в операцию перехода.

В регистры управления записываются типы переходов `ctpew` (переход по пустому значению).

Формируются новые значения контекста и регистровых файлов.

Переключение регистровых файлов

Переключение регистрового окна проиллюстрировано на рисунке R-1 (см. раздел Механизм регистровых окон). Параметром wbs для операции CALL задается величина смещения регистрового окна при вызове (значение M на рисунке R-1). Это значение не может быть больше размера текущего окна и не может быть меньше начального размера текущего окна (размера окна для передачи параметров). Если значение не удовлетворяет указанным условиям, переход не происходит и возникает прерывание.

При процедурном переходе формируется регистровое окно с размером, равным размеру области передаваемых параметров. Этот размер вычисляется как разность общего размера регистрового окна и значения смещения при вызове (параметр wbs операции CALL, значение M на рисунке R-1).

При процедурном переходе создается новый предикатный файл.

9.5.4. Возврат из процедуры¶

Возврат из процедуры осуществляется подготовленным переходом. Для этого используется операция подготовки RETURN. Операция из стека связующей информации загружает сохраненный адрес точки возврата. Подготовка формирует код операции перехода return. Результат операции может быть помещен только в регистр управления `%ctpr3`.

Выполнение возврата осуществляется операцией выполнения перехода CT. При выполнении операции восстанавливаются значения специальных регистров, сохраненные в стеке связующей информации.

Если индекс модуля адреса точки возврата отличается от индекса текущего модуля, производится переключение контекстных регистров TSD, GD и CUD.

Команды микропроцессора¶

Данный раздел - справочное руководство по командам ассемблера «Эльбрус».

Здесь представлены наиболее часто используемые команды в ассемблерной мнемонике. Их можно увидеть в ассемблерном коде, получаемом с помощью компилятора, при подаче в строку компиляции опции `-S`. Вместо файла `<sourcefile>.o` будет сгенерирован файл `<sourcefile>.s`. Возможно также использовать дизассемблер (**objdump** из `binutils`, **ldis** из `/opt/mcst/`, встроенный дизассемблер отладчика и т.д.) для просмотра команд объектного кода в ассемблерной мнемонике.

Структура описания операции¶

Краткое описание формы:

```
ADDs/d        (.s)    sss/ddd         сложение целых 32/64
  |            |       |||              |
  |            |       |||             краткий комментарий
  |            |       |||
  |            |       формат операндов и результата (результат занимает правую
  |            |       позицию): в примере операция ADDs принимает
  |            |       операнды одинарного формата и производит результат одинарного
  |            |       формата, тогда как операция ADDd  - значения двойного формата;
  |            |       используются следующие правила:
  |            |       s - операнд или результат является значением одинарного формата
  |            |           в регистровом файле
```

```
  |             |       d - операнд или результат является значением двойного формата
  |             |           в регистровом файле
  |             |       x - операнд или результат является значением расширенного
  |             |           формата в регистровом файле
  |             |       q - операнд или результат является значением квадро формата
  |             |           в регистровом файле
  |             |       b - операнд или результат является предикатом
  |             |           в предикатном файле
  |             |       v - операнд или результат является предикатом,
  |             |           вычисленным в текущей широкой команде
  |             |       e - результат операции управляет выполнением других операций
  |             |           в текущей широкой команде (предикатное выполнение)
  |             |       r - операнд или результат является регистром состояния
  |             |       i - операнд является непосредственной константой из текущей
  |             |           широкой команды
  |             |       - - отсутствие операнда
  |             |
  |            признак спекулятивного исполнения операции
  |
 мнемоника операции (в примере - ADDs или ADDd)
```

Для обозначения битовых векторов приведём фрагмент описания операции:

```
getfs           src1, src2, dst
```

Пример структуры числового аргумента в операции:

```
Size = (bitvect)src2[10:6];
```

Здесь и далее (bitvect)value означает представление числа value в виде битового вектора, а (bitvect)value[beg:end] - подвектор битового вектора между позициями beg и end.

Спекулятивное исполнение¶

Большую часть команд микропроцессора «Эльбрус» можно исполнять в **спекулятивном режиме** - это означает, что при значениях аргументов, в обычном режиме приводящих к выработке исключительной ситуации, в спекулятивном режиме операция запишет в тэг результата признак диагностического значения. Более подробное описание смотри в Спекулятивный режим.

Здесь ограничимся краткой мнемонической таблицей:

таблица спекулятивного режима¶

режим/ аргументы	допустимые	недопустимые	хотя бы один диагностический
обычный	результат	исключение	исключение
спекулятивный	результат	диагностический	диагностический

Обзор целочисленных операций¶

Операции, описанные в данном разделе, в зависимости от своего типа, вырабатывают либо числовое значение (включая флаги), либо предикат. В первом случае результат записывается в регистровый файл (RF), во втором - в предикатный файл (PF). При нормальном завершении числовой операции результат имеет тег «tagnum» соответствующего формата; в случае особой ситуации выдается результат диагностического типа. При нормальном завершении предикатной операции результат имеет тег, равный `0`; в случае особой ситуации выдается предикат диагностического типа.

Операции сложения, вычитания, обратного вычитания¶

```
ADDs/d          sss/ddd  сложение 32/64
 SUBs/d          sss/ddd  вычитание 32/64
 RSUBs/d         sss/ddd  обратное вычитание 32/64; используется только в качестве
                          второй стадии комбинированной операции
```

Операции сложения **ADDs/d** выполняют целое сложение двух операндов.

Операции вычитания **SUBs/d** вычитают операнд 2 из операнда 1.

Операции обратного вычитания **RSUBs/d** вычитают операнд 1 из операнда 2. Они используются только в качестве операции второй стадии комбинированной операции. Их первый операнд является третьим операндом комбинированной операции, а второй операнд представляет собой результат первой стадии комбинированной операции.

Язык ассемблера:

```
adds            src1, src2, dst
 addd            src1, src2, dst
 subs            src1, src2, dst
 subd            src1, src2, dst
 add_rsubd       scr1, src2, scr3, dst
```

Операции умножения¶

```
MULs/d          sss/ddd  умножение 32/64
 UMULX           ssd      умножение целых без знака 32 * 32 -> 64
 SMULX           ssd      умножение целых со знаком 32 * 32 -> 64
 UMULHd          ddd      умножение целых без знака 64 * 64 -> 128 (старшая часть)
 SMULHd          ddd      умножение целых со знаком 64 * 64 -> 128 (старшая часть)
```

Операция **UMULX** умножает значения в формате int32 без знака и вычисляет произведение в формате int64 без знака.

Операция **SMULX** умножает значения в формате int32 со знаком и вычисляет произведение в формате int64 со знаком.

Операция **UMULHd** умножает значения в формате int64 без знака, вычисляет произведение в формате int128 без знака и в качестве результата выдает старшие 64 разряда.

Операция **SMULHd** умножает значения в формате int64 со знаком, вычисляет произведение в формате int128 со знаком и в качестве результата выдает старшие 64 разряда.

Операции **MULs/d** выполняют целое умножение двух 32/64-разрядных операндов, вырабатывая 32/64-разрядный результат.

Язык ассемблера:

```
muls            src1, src2, dst
 muld            src1, src2, dst
 umulx           src1, src2, dst
 smulx           src1, src2, dst
 umulhd          src1, src2, dst
 smulhd          src1, src2, dst
```

Операции деления и вычисления остатка¶

```
UDIVX           dss      деление целых без знака 64/32->32
 UMODX           dss      остаток от деления целых без знака 64/32->32
 SDIVX           dss      деление целых со знаком 64/32->32
 SMODX           dss      остаток от деления целых со знаком 64/32->32
 SDIVs/d         sss/ddd  деление целых со знаком 32/32->32 или 64/64->64
 UDIVs/d         sss/ddd  деление целых без знака 32/32->32 или 64/64->64
```

Операция **UDIVX** выполняет деление без знака операнда 1 на операнд 2. Нецелочисленные частные округляются отсечением (отбрасыванием дробной части - truncate toward 0). Особая ситуация exc_div

вырабатывается, если делитель равен 0, или частное слишком велико для формата регистра назначения.

Операция **UMODX** вычисляет остаток, получаемый при делении без знака операнда 1 на операнд 2. Остаток всегда меньше делителя по абсолютной величине и имеет тот же знак, что и делимое. Особая ситуация exc_div вырабатывается, если делитель равен 0, или нарушены ограничения, применимые к особым ситуациям.

Операция **SDIVX** выполняет деление со знаком операнда 1 на операнд 2. Нецелочисленные частные округляются отсечением (отбрасыванием дробной части - truncate toward 0). Особая ситуация exc_div вырабатывается, если делитель равен 0, или частное слишком велико для формата регистра назначения.

Операция **SMODX** вычисляет остаток, получаемый при делении со знаком операнда 1 на операнд 2. Остаток всегда меньше делителя по абсолютной величине и имеет тот же знак, что и делимое. Особая ситуация exc_div вырабатывается, если делитель равен 0, или нарушены ограничения, применимые к особым ситуациям.

Операции **UDIVs/UDIVd** выполняют деление без знака операнда 1 на операнд 2. Особая ситуация exc_div вырабатывается, если делитель равен 0.

Операции **SDIVs/SDIVd** выполняют деление со знаком операнда 1 на операнд 2. Особая ситуация exc_div вырабатывается, если делитель равен 0. Если наибольшее отрицательное число делится на -1, то результатом является наибольшее отрицательное число.

Язык ассемблера:

```
udivx           src1, src2, dst
 umodx           src1, src2, dst
 sdivx           src1, src2, dst
 smodx           src1, src2, dst
 udivs           src1, src2, dst
 udivd           src1, src2, dst
 sdivs           src1, src2, dst
 sdivd           src1, src2, dst
```

Операции сравнения целых чисел¶

```
CMP(s/d)b       группа  из 8-ми операций сравнения:

 CMPO(s/d)b      ssb/ddb  сравнение 32/64 "переполнение"
 CMPB(s/d)b      ssb/ddb  сравнение 32/64 "< без знака"
```

```
CMPE(s/d)b      ssb/ddb  сравнение 32/64 "равно"
CMPBE(s/d)b     ssb/ddb  сравнение 32/64 "<= без знака"
CMPS(s/d)b      ssb/ddb  сравнение 32/64 "отрицательный"
CMPP(s/d)b      ssb/ddb  сравнение 32/64 "нечетный"
CMPL(s/d)b      ssb/ddb  сравнение 32/64 "< со знаком"
CMPLE(s/d)b     ssb/ddb  сравнение 32/64 "<= со знаком"

CMPAND(s/d)b    группа  из 4-х операций проверки:

CMPANDE(s/d)b   ssb/ddb  поразрядное "and" и проверка 32/64 "равно 0"
CMPANDS(s/d)b   ssb/ddb  поразрядное "and" и проверка 32/64 "отрицательный"
CMPANDP(s/d)b   ssb/ddb  поразрядное "and" и проверка 32/64 "нечетный"
CMPANDLE(s/d)b  ssb/ddb  поразрядное "and" и проверка 32/64 "<=0 со знаком"
```

Операции **CMP** вычитают операнд 2 из операнда 1 и определяют флаги, как это делают операции **SUB**. Далее по состоянию флагов формируется результат - предикат «true» или «false».

Операции **CMPAND** выполняют поразрядное логическое «and» операнда 1 и операнда 2 и определяют флаги, как это делают операции **AND**. Далее по состоянию флагов формируется результат - предикат «true» или «false».

Язык ассемблера:

```
cmpodb          src1, src2, predicate
 cmpbdb          src1, src2, predicate
 cmpedb          src1, src2, predicate
 cmpbedb         src1, src2, predicate
 cmposb          src1, src2, predicate
 cmpbsb          src1, src2, predicate
 cmpesb          src1, src2, predicate
 cmpbesb         src1, src2, predicate
 cmpsdb          src1, src2, predicate
 cmppdb          src1, src2, predicate
 cmpldb          src1, src2, predicate
 cmpledb         src1, src2, predicate
 cmpssb          src1, src2, predicate
 cmppsb          src1, src2, predicate
 cmplsb          src1, src2, predicate
 cmplesb         src1, src2, predicate
 cmpandesb       src1, src2, predicate
 cmpandssb       src1, src2, predicate
 cmpandpsb       src1, src2, predicate
 cmpandlesb      src1, src2, predicate
 cmpandedb       src1, src2, predicate
 cmpandsdb       src1, src2, predicate
 cmpandpdb       src1, src2, predicate
```

```
 cmpandledb      src1, src2, predicate
```

Логические поразрядные операции¶

```
ANDs/d          sss/ddd  логическое "and" 32/64
 ANDNs/d         sss/ddd  логическое "and" 32/64 с инверсией операнда 2
 ORs/d           sss/ddd  логическое "or" 32/64
 ORNs/d          sss/ddd  логическое "or" 32/64 с инверсией операнда 2
 XORs/d          sss/ddd  логическое исключительное "or" 32/64
 XORNs/d         sss/ddd  логическое исключительное "or" 32/64 с инверсией операнда 2
```

Эти операции выполняют поразрядные логические операции. Операции **ANDN**, **ORN** и **XORN** логически инвертируют операнд 2, прежде чем выполнить основную (**AND**, **OR** или исключающее **OR**) операцию.

Язык ассемблера:

```
ands            src1, src2, dst
 andd            src1, src2, dst
 andns           src1, src2, dst
 andnd           src1, src2, dst
 ors             src1, src2, dst
 ord             src1, src2, dst
 orns            src1, src2, dst
 ornd            src1, src2, dst
 xors            src1, src2, dst
 xord            src1, src2, dst
 xorns           src1, src2, dst
 xornd           src1, src2, dst


SHLs/d          sss/ddd  сдвиг влево 32/64
 SHRs/d          sss/ddd  сдвиг вправо логический 32/64
 SCLs/d          sss/ddd  сдвиг влево циклический 32/64
 SCRs/d          sss/ddd  сдвиг вправо циклический 32/64
 SARs/d          sss/ddd  сдвиг вправо арифметический 32/64
```

Операции **SHLs/d** сдвигают операнд 1 влево на число разрядов, указанных в операнде 2. Самый старший разряд сдвигается во флаг CF. Освободившиеся позиции младших разрядов заполняются нулями.

Операции **SHRs/d** сдвигают операнд 1 вправо на число разрядов, указанных в операнде 2. Самый младший разряд сдвигается во флаг CF. Освободившиеся позиции старших разрядов заполняются

нулями.

Операции **SCLs/d** сдвигают операнд 1 влево на число разрядов, указанных в операнде 2. Самый старший разряд сдвигается во флаг CF. Освободившиеся позиции младших разрядов заполняются выдвинутыми старшими разрядами операнда.

Операции **SCRs/d** сдвигают операнд 1 вправо на число разрядов, указанных в операнде 2. Самый младший разряд сдвигается во флаг CF. Освободившиеся позиции старших разрядов заполняются выдвинутыми младшими разрядами операнда.

Операции **SARs/d** сдвигают операнд 1 вправо на число разрядов, указанных в операнде 2. Самый младший разряд сдвигается во флаг CF. Освободившиеся позиции старших разрядов заполняются самым старшим значимым разрядом операнда.

Эти операции выдают либо числовой результат, либо флаг.

Язык ассемблера:

```
shls            src1, src2, dst
 shld            src1, src2, dst
 shrs            src1, src2, dst
 shrd            src1, src2, dst
 scls            src1, src2, dst
 scld            src1, src2, dst
 scrs            src1, src2, dst
 scrd            src1, src2, dst
 sars            src1, src2, dst
 sard            src1, src2, dst
```

Операции «взять поле произвольной длины»¶

```
GETFs/d         sss/ddd  выделить поле произвольной длины
```

Операции **GETFs/d** выделяют произвольное поле первого операнда. Остальные разряды результата заполняются либо нулями, либо старшим значащим разрядом выделенного поля. Параметры поля определяются значением второго операнда:

```
Для GETFs
 Правый разряд поля:
   ShiftCount = (bitvect)src2[4:0];
 Длина поля:
   Size = (bitvect)src2[10:6];
```

```
Для GETFd
Правый разряд поля:
  ShiftCount = (bitvect)src2[5:0];
Длина поля:
  Size = (bitvect)src2[11:6];
```

Здесь и далее (bitvect)value означает представление числа value в виде битового вектора, а (bitvect)value[beg:end] - подвектор битового вектора между позициями beg и end.

Язык ассемблера:

```
getfs           src1, src2, dst
 getfd           src1, src2, dst
```

Операции «вставить поле»¶

```
INSFs/d         ssss/dddd  вставить поле 32/64
```

Операции **INSFs/d** циклически сдвигают вправо 1-й операнд и вставляют произвольное количество самых правых разрядов 3-го операнда в самые правые разряды циклически сдвинутого 1-го операнда. Параметры поля определяются значением 2-го операнда:

```
Для INSFs
Правый разряд поля:
  ShiftCount = (bitvect)src2[4:0];
Длина поля:
  Size = (bitvect)src2[10:6];
Для INSFd
Правый разряд поля:
  ShiftCount = (bitvect)src2[5:0];
Длина поля:
  Size = (bitvect)src2[11:6];
```

Язык ассемблера:

```
insfs           src1, src2, src3, dst
 insfd           src1, src2, src3, dst
```

Расширение знаком или нулем¶

```
SXT             ssd  расширение знаком или нулем 8/16/32 до 64
```

Операция **SXT** преобразует значение 2-го аргумента в формате байт/полуслово/одинарное слово в формат двойное слово. Операция заполняет остальные разряды результата либо нулями (zero-extended), либо знаком (старшим разрядом) байта/полуслова/слова (sign-extended). Разрядность и знаковость определяются по 1-му аргументу.

```
Формат:
   src1[1:0] == 0 -> 8 бит
   src1[1:0] == 1 -> 16 бит
   src1[1:0] == 2 -> 32 бит
   src1[1:0] == 3 -> 32 бит
   src1[2:2] == 0 - беззнаковое
   src1[2:2] == 1 - знаковое
```

Язык ассемблера:

```
sxt             src1, src2, dst
```

Выбор из двух операндов¶

```
MERGEs/d        sss/ddd  выбрать один из операндов 32/64 как результат
                          (требует комбинированной операции RLP)
```

Операция **MERGE** выдает в качестве результата один из двух числовых операндов в зависимости от значения третьего операнда - предиката. От тернарного оператора языка C/C++:

```
cond ? altT : altF
```

отличается тем, что выбирается значение src1 при `predicate == F`, и src2 при `predicate == T`.

Язык ассемблера:

```
merges          src1, src2, dst, predicate
 merged          src1, src2, dst, predicate
```

Обзор вещественных скалярных операций¶

Перечень операций.

```
FADDs/d         sss/ddd  сложение fp32/fp64
 FSUBs/d         sss/ddd  вычитание fp32/fp64
 FRSUBs/d        sss/ddd  обратное вычитание fp32/fp64; используется только в
                          качестве второй стадии комбинированной операции
 FMAXs/d         sss/ddd  максимум fp32/fp64
 FMINs/d         sss/ddd  минимум fp32/fp64

 FMULs/d         sss/ddd  умножение fp32/fp64
 FSCALEs/d       sss/dsd  умножение fp32/fp64 на целую степень двойки

 FDIVs/d         sss/ddd  деление fp32/fp64
 FRCPs           -ss      обратная величина fp32

 FSQRTs          -ss      квадратный корень fp32
 FSQRTId         -dd      квадратный корень fp64 начальная команда
 FSQRTTd         ddd      квадратный корень fp64 конечная команда
 FRSQRTs         -ss      обратная величина квадратного корня fp32

 FCMPEQs/d       sss/ddd fp32/fp64  сравнение на равно, результат в регистровом
                                    файле
 FCMPLTs/d       sss/ddd fp32/fp64  сравнение на меньше, результат в регистровом
                                    файле
 FCMPLEs/d       sss/ddd fp32/fp64  сравнение на меньше или равно, результат в
                                    регистровом файле
 FCMPUODs/d      sss/ddd fp32/fp64  сравнение на не упорядочено, результат в
                                    регистровом файле
 FCMPNEQs/d      sss/ddd fp32/fp64  сравнение на не равно, результат в регистровом
                                    файле
 FCMPNLTs/d      sss/ddd fp32/fp64  сравнение на не меньше, результат в регистровом
                                    файле
 FCMPNLEs/d      sss/ddd fp32/fp64  сравнение на не меньше или равно, результат
                                    в регистровом файле
 FCMPODs/d       sss/ddd fp32/fp64  сравнение на упорядочено, результат в
                                    регистровом файле

 FCMPEQ(s/d)b    ssb/ddb fp32/fp64  сравнение на равно с формированием
                                    результата в виде предиката
 FCMPLT(s/d)b    ssb/ddb fp32/fp64  сравнение на меньше с формированием
                                    результата в виде предиката
 FCMPLE(s/d)b    ssb/ddb fp32/fp64  сравнение на меньше или равно с формированием
                                    результата в виде предиката
 FCMPUOD(s/d)b   ssb/ddb fp32/fp64  сравнение на неупорядочено с формированием
                                    результата в виде предиката
 FCMPNEQ(s/d)b   ssb/ddb fp32/fp64  сравнение на не равно с формированием
                                    результата в виде предиката
 FCMPNLT(s/d)b   ssb/ddb fp32/fp64  сравнение на не меньше с формированием
                                    результата в виде предиката
 FCMPNLE(s/d)b   ssb/ddb fp32/fp64  сравнение на не меньше или равно
```

```
 FCMPOD(s/d)b    ssb/ddb fp32/fp64  с формированием результата в виде предиката
                                    сравнение на упорядочено с формированием
                                    результата в виде предиката
```

Операции сложения, вычитания, умножения, деления, сравнения, вычисления максимума и минимума имеют достаточно понятную мнемонику и в детальном описании не нуждаются.

Операции умножения на целую степень двойки¶

Операции **FSCALEs/d** выполняют умножение вещественного числа соответствующего формата, содержащегося в 1-м операнде, на целую степень двойки, задаваемую 2-м операндом форматов; результат имеет формат 1-го операнда.

```
FSCALEs/d       sss/dsd  умножение fp32/fp64 на целую степень двойки
```

Язык ассемблера:

```
fscales         src1, src2, dst
 fscaled         src1, src2, dst
```

Операции вычисления квадратного корня¶

Операция **FSQRTs** вычисляет квадратный корень из 2-го операнда формата fp32.

Операция **FSQRTId** вычисляет первую аппроксимацию квадратного корня из 2-го операнда формата fp64.

Операция **FSQRTTd** завершает вычисление квадратного корня из 1-го операнда формата fp64, используя первую аппроксимацию, вычисленную операцией **FSQRTId** и содержащуюся во 2-м операнде. Результат, полученный последовательным выполнением двух операций **FSQRTId** и **FSQRTTd**, соответствует стандарту IEEE Standard 754.

```
FSQRTs          -ss  квадратный корень fp32
 FSQRTId         -dd  квадратный корень fp64 начальная команда
 FSQRTTd         ddd  квадратный корень fp64 конечная команда
```

Язык ассемблера:

```
fsqrts          src2, dst
 fsqrtid         src2, dst
```

```
 fsqrttd         src1, src2, dst
```

Скалярные операции преобразования формата¶

```
FSTOFD          -sd  fp32 в fp64
 FDTOFS          -ds  fp64 в fp32

 FSTOIFs         sss  целая часть fp32 в fp32
 FDTOIFd         ddd  целая часть fp64 в fp64

 FSTOIS          -ss  fp32 в int32
 FSTOID          -sd  fp32 в int64
 FDTOIS          -ds  fp64 в int32
 FDTOID          -dd  fp64 в int64
 FSTOIStr        -ss  fp32 в int32 с обрубанием
 FDTOIStr        -ds  fp64 в int32 с обрубанием
 FSTOIDtr        -sd  fp32 в int64 с обрубанием
 FDTOIDtr        -dd  fp64 в int64 с обрубанием

 ISTOFS          -ss  int32 в fp32
 ISTOFD          -sd  int32 в fp64
 IDTOFS          -ds  int64 в fp32
 IDTOFD          -dd  int64 в fp64

 FSTOFD          -sd  fp32 to fp64
 FDTOFS          -ds  fp64 to fp32
```

Операции FSTOIFs и FDTOIFd имеют два аргумента (в отличие от других операций преобразования формата). Из 1-го аргумента используются 3 младших бита, определяющих режим округления:

```
if ((bitvect)scr1[2:2] == 0)
     rounding_mode = (bitvect)src1[1:0];
 else
     rounding_mode = PFPFR.rc;
```

Язык ассемблера:

```
fstofd          src2, dst
 fdtofs          src2, dst
 fstoifs         src1, src2, dst
 fdtoifd         src1, src2, dst
 fstois          src2, dst
 fstoid          src2, dst
 fdtois          src2, dst
 fdtoid          src2, dst
```

```
fstoistr        src2, dst
fdtoistr        src2, dst
fstoidtr        src2, dst
fdtoidtr        src2, dst
```

Предикатные операции¶

Логический предикат представляет собой тегированные 1-разрядные булевские данные, принимающие следующие значения:

```
тег      значение
 0        0        - "false"
 0        1        - "true"
 1        x        - "DP" (диагностический предикат)
```

Результатом операции также является тегированный предикат.

Операции вычисления предикатов¶

Операции над логическими предикатами размещаются в PLS слогах. В PLS слогах могут размещаться до 7 операций трех основных типов:

- вычисление первичного логического предиката (Evaluate Logical Predicate - ELP) является ссылкой на первичный (primary) предикат, хранящийся в предикатном регистре PR, или определяет так называемые специальные предикаты; эти операции поставляют исходные предикаты для операций **CLP** и **MLP**;
- вычисление вторичного логического предиката (Calculate LogicalPredicate - CLP) является логической функцией с двумя аргументами; её результат можно записать в предикатный регистр PR;
- условная пересылка логического предиката (Conditional Move Logical Predicate - MLP) записывает или теряет результат операций **ELP** или **CLP**, в зависимости от значения предиката-условия.

Широкая команда может включать до:

- 4 ELP;
- 3 CLP/MLP.

Обозначения для предикатов

В этом разделе вводятся следующие обозначения.

До семи промежуточных предикатов с номерами от 0 до 6 (p0, p1,…p6) могут формироваться операциями **ELP** и **CLP**.

Предикатам, формируемым операциями **ELP**, присваиваются номера от 0 до 3 (`p0...p3`); предикатам, формируемым операциями **CLP**, присваиваются номера от 4 до 6 (`p4...p6`); операции **MLP** не формируют промежуточных предикатов.

В соответствии с этими номерами операции **CLP/MLP** обращаются к своим операндам — предикатам, выработанным в данной команде (операции **CLP** могут быть каскадными (см. ниже)).

При упаковке в слоги PLS операция, формирующая предикат с конкретным номером, может занимать только определенное положение. Поэтому далее в данном разделе операции **ELP** и **CLP** могут нумероваться как `ELP0`, `ELP1`, `ELP2`, `ELP3`, `CLP0`, `CLP1`, `CLP2`.

Вычисление первичного логического предиката (Evaluate Logical Predicate - ELP)¶

Операция **ELP** считывает для использования либо предикат из предикатного файла PF, либо один из специальных предикатов.

Язык ассемблера:

```
pass                    predicate, local_predicate
 spred, elp_number       alu_channel, ...
```

где:

```
predicate               - один из описанных ниже;
 local_predicate         - один из @p0, @p1, @p2, @p3;
 elp_number              - один из 0, 1, 2, 3;
 alu_channel             - список любых из >alc0, >alc1, ... >alc5.
```

Направить логический предикат (Route Logical Predicate - RLP)¶

Операция **RLP** задает предикатное выполнение операции арифметико-логического канала и определяет (направляет) предикат для управления этой операцией.

Язык ассемблера:

```
adds     src1, src2, dst ? predicate
```

где:

```
predicate               - один из описанных ниже:
                            * исчерпание счетчиков цикла - %lcntex;
                            * значение счетчика пролога - %pcnt<N>;
                            * предикат в предикатном файле - %pred<N>.
```

Условие для операции MERGE (Merge Condition - MRGC)¶

Операции **MRGC** вырабатывают условия для алгоритма операции **MERGE**.

Язык ассемблера:

```
merged  src1, src2, dst, predicate
```

где:

```
predicate               - один из описанных ниже:
                            * исчерпание счетчиков цикла - %lcntex;
                            * значение счетчика пролога - %pcnt<N>;
                            * предикат в предикатном файле - %pred<N>.
```

Вычисление логического предиката (Calculate Logical Predicate - CLP)¶

Операция **CLP** является логической функцией с двумя аргументами, в качестве аргументов она получает предикаты, сформированные операциями **ELP** или **CLP** (но не **MLP**) из той же самой широкой команды, и ее результат - логическое «И» аргументов (с возможной инверсией) может записываться в PF.

Язык ассемблера:

```
andp            [~]local_predicate, [~]local_predicate, local_predicate_dst
 landp           [~]local_predicate, [~]local_predicate, local_predicate_dst
 pass          local_predicate_dst, predicate
```

где:

```
local_predicate         - один из @p0, @p1, ... @p6;
 ~                       - инверсия значения предиката перед выполнением функции;
 predicate               - предикат в PF - %pred<N>.
```

Условная пересылка логического предиката (Conditional Move Logical Predicate - MLP)¶

Операция **MLP** условно записывает предикат, выработанный операциями **ELP** или **CLP**. В результате этой операции первый аргумент будет записан в результат, если второй аргумент равен 1.

Язык ассемблера:

```
movep           local_predicate, local_predicate, local_predicate_dst
 pass            local_predicate_dst, predicate
```

Операции обращения в память¶

Операции обращения в память включают операции считывания и записи. Операции считывания читают несколько байтов из пространства памяти и помещают их в регистр назначения. Операции записи пишут несколько байтов из регистра источника в пространство памяти.

Определяются следующие порции считываемой/записываемой информации: байт (byte), полуслово (half-word), одинарное слово (word), двойное слово (double-word), квадро слово (quad-word). Порция определяется кодировкой операции.

Размещение в пространстве памяти определяется операндами операции. Один из них обычно является адресным типом, другие (если присутствуют) являются индексом(ами) в терминах байтов.

Операции считывания из незащищенного пространства¶

```
LDB             ddd  считывание байта без знака
 LDH             ddd  считывание полуслова без знака
 LDW             ddd  считывание одинарного слова
 LDD             ddd  считывание двойного слова
```

Язык ассемблера:

```
ldb             [ address ] mas, dst
 ldh             [ address ] mas, dst
 ldw             [ address ] mas, dst
 ldd             [ address ] mas, dst
```

Операции записи в незащищенное пространство¶

```
STB             dds  запись байта
 STH             dds  запись полуслова
 STW             dds  запись одинарного слова
 STD             ddd  запись двойного слова
```

Язык ассемблера:

```
stb             src3, [ address ] { mas }
 sth             src3, [ address ] { mas }
 stw             src3, [ address ] { mas }
 std             src3, [ address ] { mas }
```

Операции считывания в режиме -mptr32¶

```
LDGDB     r,ssd  считывание байта без знака
 LDGDH     r,ssd  считывание полуслова без знака
 LDGDW     r,ssd  считывание одинарного слова
 LDGDD     r,ssd  считывание двойного слова
 LDGDQ     r,ssq  считывание квадро слова
```

Язык ассемблера:

```
ldgdb           [ address ] { mas }, dst
 ldgdh           [ address ] { mas }, dst
 ldgdw           [ address ] { mas }, dst
 ldgdd           [ address ] { mas }, dst
 ldgdq           [ address ] { mas }, dst
```

Операции записи в режиме -mptr32¶

Операции данного раздела относятся к группе операций с «защищенными» данными.

```
STGDB           r,sss  запись байта
 STGDH           r,sss  запись полуслова
 STGDW           r,sss  запись одинарного слова
 STGDD           r,ssd  запись двойного слова
 STGDQ           r,ssq  запись квадро слова
```

Язык ассемблера:

```
stgdb           src3, [ address ] { mas }
 stgdh           src3, [ address ] { mas }
 stgdw           src3, [ address ] { mas }
 stgdd           src3, [ address ] { mas }
 stgdq           src3, [ address ] { mas }
```

Операции обращения к массиву¶

Операции считывания массива¶

```
LDAAB           ppd  считывание байта
 LDAAH           ppd  считывание полуслова
 LDAAW           ppd  считывание одинарного слова
 LDAAD           ppd  считывание двойного слова
 LDAAQ           ppq  считывание квадро слова
```

Язык ассемблера:

```
ldaab           %aadN [ %aastiL {+ literal32} ], dst
 ldaah           %aadN [ %aastiL {+ literal32} ], dst
 ldaad           %aadN [ %aastiL {+ literal32} ], dst
 ldaaw           %aadN [ %aastiL {+ literal32} ], dst
 ldaaq           %aadN [ %aastiL {+ literal32} ], dst

 incr            %aaincrM {? <предикат для модификации адреса>}
                 ,где K, L, M, N - целые без знака
```

Операции записи в массив¶

```
STAAB           pps  запись байта
 STAAH           pps  запись полуслова
 STAAW           pps  запись одинарного слова
 STAAD           ppd  запись двойного слова
 STAAQ           ppq  запись квадро слова
```

Язык ассемблера:

```
staab           src3, %aadN [ %aastiL {+ literal32} ] { mas }
 staah           src3, %aadN [ %aastiL {+ literal32} ] { mas }
 staad           src3, %aadN [ %aastiL {+ literal32} ] { mas }
 staaw           src3, %aadN [ %aastiL {+ literal32} ] { mas }
 staaq           src3, %aadN [ %aastiL {+ literal32} ] { mas }
```

```
 incr            %aaincrM {? <предикат для модификации адреса>}
                 ,где K, L, M, N - целые без знака
```

Операции преобразования адресных объектов¶

Взять указатель стека (GETSP)¶

```
GETSP           rsd  взять подмассив стека пользователя
```

Операция **GETSP**, в зависимости от знака операнда 2, либо выделяет свободную область в незащищенном стеке пользователя, либо возвращает ранее выделенную память. В обоих случаях операция модифицирует указатель стека.

Язык ассемблера:

```
getsp           src2, dst
```

Переслать тэгированное значение (MOVT)¶

```
MOVTs           -ss  переслать тэгированный адресный объект 32
 MOVTd           -dd  переслать тэгированный адресный объект 64
 MOVTq           -qq  переслать тэгированный адресный объект 128
```

Операция **MOVT** копирует значение регистра с сохранением тэгов в регистр назначения.

Язык ассемблера:

```
movts           src2, dst
 movtd           src2, dst
 movtq           src2, dst
 movtd           src2, ctp_reg
```

Операции доступа к регистрам состояния¶

Архитектура определяет несколько методов доступа к регистрам состояния:

- операции **RW** и **RR** обычно обеспечивают доступ к регистрам, которые контролируют всю работу процессора;
- операции **SETxxx** предлагаются как оптимальный способ модификации некоторых предопределенных регистров;
- операции **{STAAxx + MAS}** и **{LDAAxx + MAS}** обычно обеспечивают доступ к регистрам AAU;
- операции **{STxx + MAS}** и **{LDxx + MAS}** обычно обеспечивают доступ к регистрам MMU.

Операции «установить регистры» и «проверить области параметров»¶

```
SETBN           -ir  установить вращаемую базу NR-ов
 SETBP           -ir  установить вращаемую базу PR-ов
 SETWD           -ir  изменить размер окна стека процедур
 VFRPSZ          -i-  проверить размер регистровой области параметров процедуры
```

Язык ассемблера:

```
setbn           { rbs = NUM } { , rsz = NUM } { , rcur = NUM }
 setbp           { psz = NUM }
 setwd           { wsz = NUM } { , nfx = NUM } { , dbl = NUM }
 vfrpsz          { rpsz = NUM }
```

Операции подготовки передачи управления¶

Есть два типа передачи управления:

- немедленная («instant»);
- подготовленная («prepared»).

Для немедленной передачи управления необходима одна операция, которая содержит всю необходимую информацию и передаёт управление немедленно.

Передачи управления типа «подготовленная» разбиты на две операции:

- подготовка передачи управления (control transfer preparation - CTP);
- фактическая передача управления (control transfer - CT).

Операции **CTP** предназначены для подготовки информации, необходимой для быстрой фактической передачи управления. Целью является выполнение всей подготовительной работы «на фоне» и одновременно с основной активностью обработки данных. Операции **CTP** содержат всю или часть

информации о передаче управления (тип, адрес, etc); эта информация сохраняется до операции **CT** на одном из регистров CTPRj и используется ею для фактической передачи управления.

Передачи управления могут включать следующие элементы в различных разумных сочетаниях:

- Переключение указателя команды IP (кода) - присутствует всегда; указатель команды IP перехода может быть получен одним из следующих способов:
    - литеральное смещение относительно текущего указателя команды IP;
    - динамическое/литеральное смещение относительно текущего дескриптора модуля компиляции CUD;
    - тэгированная метка (для защищенного адресного пространства), поступающая из регистрового файла RF;
    - целочисленная метка (для незащищенного адресного пространства), поступающая из регистрового файла RF;
    - указатель команды IP возврата, поступающий из регистров стека связующей информации процедур.
- Переключение регистрового окна - характерно для процедурных передач; при вызовах это статически известная информация, кодируемая литерально; при возвратах информация поступает из регистров стека связующей информации.
- Переключение фрейма стека пользователя - характерно для процедурных передач; при возвратах информация поступает из регистров стека связующей информации; при входах пространство стека пользователя не назначается.
- Переключение контекста - характерно для процедурных передач; контекст глобальных процедур включает:
    - глобалы, описываемые дескриптором глобалов GD;
    - литеральные скалярные данные и массивы, описываемые дескриптором модуля компиляции CUD;

  оба дескриптора берутся из таблицы модулей компиляции CUT; соответствующая строка в таблице модулей компиляции CUT определяется PTE точки перехода.

Существуют следующие типы передач управления (как подготовленных, так и немедленных), то есть разумные сочетания элементов, описанных выше:

- BRANCH - непроцедурная подготовленная передача управления; она включает переключение указателя команды IP; IP задается операцией **DISP**, **GETPL** или **MOVTd**.

Переключение (передача управления) осуществляется операцией **CT**;

- IBRANCH - непроцедурная немедленная передача управления; она включает переключение указателя команды IP; IP задается операцией **IBRANCH**.

  Переключение (передача управления) осуществляется операцией **IBRANCH**;

- CALL - подготовленный вызов процедуры; он включает:

  - переключение указателя команды IP; IP задается операцией **DISP**, **GETPL** или **MOVTd**;
  - переключение регистрового окна; окно задается операцией **CALL**;
  - переключение контекста; новый контекст включает глобалы, литералы и классы. Контекст автоматически считывается из памяти операцией **CALL**.

  Переключение (передача управления) осуществляется операцией **CALL**;

- SCALL - подготовленный вызов процедуры OS (глобальной); он включает:

  - переключение указателя команды IP; IP задается операцией **SDISP**;
  - переключение регистрового окна; окно задается операцией **SCALL**;
  - переключение контекста; новый контекст включает глобалы и литералы, хранящиеся на регистрах процессора.

  Переключение (передача управления) осуществляется операцией **SCALL**;

- RETURN - подготовленный возврат из процедуры; он включает:

  - восстановление указателя команды IP; IP задается операцией **RETURN** (считывается из стека связующей информации);
  - восстановление регистрового окна и фрейма стека пользователя; окно и фрейм считываются из стека связующей информации операцией **CT**;
  - восстановление контекста; новый контекст включает глобалы, литералы и классы. Контекст автоматически считывается из памяти операцией **CT**.

  Переключение (передача управления) осуществляется операцией **CT**;

- DONE - немедленный возврат из обработчика прерываний; он включает:

  - восстановление указателя команды IP; IP задается операцией **DONE** (считывается из стека связующей информации);
  - восстановление регистрового окна и фрейма стека пользователя; окно и фрейм считываются из стека связующей информации операцией **DONE**;

    - восстановление контекста; новый контекст включает глобалы, литералы и классы. Контекст автоматически считывается из памяти операцией **DONE**.

  Переключение (передача управления) осуществляется операцией **DONE**;

- аппаратный вход в обработчик прерываний; он включает:

    - переключение указателя команды IP; IP задается регистром процессора;
    - переключение регистрового окна; новое окно - пустое;
    - переключение контекста; новый контекст включает глобалы и литералы, хранящиеся на регистрах процессора.

  Переключение (передача управления) осуществляется аппаратно, по сигналам прерываний.

Подготовка перехода по литеральному смещению (DISP)¶

DISP подготавливает передачу управления типа BRANCH/CALL.

Язык ассемблера:

```
disp           ctp_reg, label  [, ipd NUM]
```

где:

```
ctp_reg - регистр подготовки перехода;
 label - метка целевого адреса;
 NUM - необязательный параметр, глубина подкачки кода
       в терминах количества строк L1$I ; NUM = 0, 1, 2.
```

Подготовка перехода по динамическому смещению (GETPL)¶

GETPL используется для подготовки передачи управления типа BRANCH/CALL. Далее приводится случай, когда **GETPL** используется как операция подготовки передачи управления.

Язык ассемблера:

```
getpl          src2, ctp_reg   [, ipd NUM]
```

где:

```
ctp_reg - регистр подготовки перехода;
 src2 - содержит смещение целевого адреса относительно CUD;
 NUM - необязательный параметр, глубина подкачки кода
       в терминах количества строк L1$I ; NUM = 0, 1, 2.
```

Подготовка перехода по метке из регистрового файла RF (MOVTd)¶

MOVTd используется для подготовки передачи управления типа BRANCH/CALL.

MOVTd в общем виде описывается в разделе Переслать тэгированное значение (MOVT); здесь приведен случай, когда **MOVTd** используется как операция подготовки передачи управления.

Считается, что эта операция пересылает метку в CTPR.

Язык ассемблера:

```
movtd          src2, ctp_reg   [, ipd NUM]
```

где:

```
ctp_reg - регистр подготовки перехода;
 src2 - содержит целевой адрес;
 NUM - необязательный параметр, глубина подкачки кода
       в терминах количества строк L1$I ; NUM = 0, 1, 2.
```

Подготовка возврата из процедуры (RETURN)¶

RETURN используется для подготовки возврата из процедуры.

Язык ассемблера:

```
return         %ctpr3   [, ipd NUM]
```

где:

```
%ctpr3 - регистр подготовки перехода;
 NUM - необязательный параметр, глубина подкачки кода
       в терминах количества строк L1$I ; NUM = 0, 1, 2.
```

Подготовка программы предподкачки массива (LDISP)¶

Программа предподкачки массива подготавливается операцией **LDISP**.

Язык ассемблера:

```
ldisp           %ctpr2, label
```

где:

```
%ctpr2 - регистр подготовки перехода;
 label - метка адреса начала асинхронной программы;
```

Предварительная подкачка кода по литеральному смещению (PREF)¶

PREF подкачивает код в кэш-память команд.

Язык ассемблера:

```
pref            prefr, label [,ipd=NUM]
```

где:

```
prefr - один из %ipr0, %ipr1,.. %ipr7;
 label - метка адреса подкачки;
 NUM - глубина подкачки; NUM = 0, 1; по умолчанию = 0.
```

Операции передачи управления (CT)¶

CT операции предназначены для условной или безусловной передачи управления из одной программной ветви в другую. Все виды передач управления гарантируют, что следом за командой, содержащей **CT** операцию, исполняется команда выбранной программной ветви (если условие истинно, то команда цели перехода; если ложно, то команда, следующая за командой перехода). Никакие команды из противоположной ветви не изменяют состояния процесса.

Существует два типа **CT** операций:

- непосредственная («instant»);

- подготовленная («prepared»).

Передача управления любого типа может быть условной.

В общей форме запись на языке ассемблера следующая:

```
ct_operation    { ? control_condition }
```

Подготовленный переход (BRANCH)¶

BRANCH выполняет непроцедурную подготовленную передачу управления.

BRANCH выполняется последовательностью двух операций:

- CT подготовка в ctp_reg, **CTP**;
- фактическая **CT** операция из этого ctp_reg.

Язык ассемблера для CT подготовки, один из вариантов:

```
disp            ctp_reg, label
 getpl           src2, ctp_reg ! - подготовка перехода по косвенности в режиме -mptr32
 movtd           src2, ctp_reg ! - подготовка перехода по косвенности в режиме -mptr64
```

Язык ассемблера для фактической **CT**:

```
ct              ctp_reg { control_condition }
```

где:

```
ctp_reg - регистр подготовки перехода;
```

Нет необходимости размещать операции **CTP** и **CT** в программе непосредственно одну за другой, они могут быть размещены на любом расстоянии друг от друга, при условии, что ctp_reg не используется повторно.

Непосредственный переход (IBRANCH)¶

IBRANCH выполняет непосредственную непроцедурную передачу управления.

Язык ассемблера:

```
ibranch         label { control_condition }
```

Вариант операции:

```
ibranch         label ? %MLOCK
```

является синонимом:

```
rbranch         label
```

Операция CALL¶

CALL выполняет процедурную подготовленную передачу управления.

CALL выполняется последовательностью двух операций:

- CT подготовка в ctp_reg, **CTP**;
- фактическая **CT** операция из этого ctp_reg.

Язык ассемблера для CT подготовки, один из вариантов:

```
disp            ctp_reg, label
 getpl           src2, ctp_reg  ! - подготовка перехода по косвенности в режиме -mptr32
 movtd           src2, ctp_reg  ! - подготовка перехода по косвенности в режиме -mptr64
 sdisp           ctp_reg, label
```

Язык ассемблера для фактической **CT**:

```
call            ctp_reg, wbs = NUM { control_condition }
```

где:

```
wbs - величина смещения регистрового окна при вызове.
```

Действие параметра wbs описано в разделе Переключение регистровых файлов.

Нет необходимости размещать операции **CTP** и **CT** в программе непосредственно одну за другой, они могут быть размещены на любом расстоянии друг от друга, при условии, что ctp_reg не используется повторно.

Возврат из аппаратного обработчика прерываний (DONE)¶

Операция **DONE** выполняет непосредственный возврат из аппаратного обработчика системных прерываний.

Язык ассемблера:

```
done            { control_condition }
```

Операции поддержки наложений цикла¶

Операции Set BR¶

```
SETBP           -ir  установить базу вращения предикатных регистров PR
 SETBN           -ir  установить базу вращения числовых регистров NR
```

Продвинуть базу вращения числовых регистров NR (ABN)¶

Операция **ABN** продвигает базу вращения числовых регистров NR. **ABN** может выполняться условно, в зависимости от условия передачи управления, закодированного в той же команде.

Язык ассемблера:

```
abn             abnf=fl_f, abnt=fl_t
```

fl_f = 0, 1;
: флаг продвижения базы при отсутствии факта передачи управления после текущей команды;

fl_t = 0, 1;
: флаг продвижения базы при наличии факта передачи управления после текущей команды (чаще всего используется для перехода по обратной дуге цикла).

Продвинуть базу вращения предикатных регистров PR (ABP)¶

Операция **ABP** продвигает базу вращения предикатных регистров PR. **ABP** может выполняться условно в зависимости от условия передачи управления, кодированного в той же команде.

Язык ассемблера:

```
abp             abpf=fl_f, abpt=fl_t
```

fl_f = 0, 1;
    флаг продвижения базы при отсутствии факта передачи управления после текущей команды;
fl_t = 0, 1;
    флаг продвижения базы при наличии факта передачи управления после текущей команды (чаще всего используется для перехода по обратной дуге цикла).

Продвинуть базу вращения глобальных числовых регистров NR (ABG)¶

Операция **ABG** продвигает базу вращения глобальных числовых регистров NR.

Язык ассемблера:

```
abg             abgi=fl_f, abgd=fl_t
```

abgi = 0, 1;
    инкрементировать базу вращения глобальных числовых регистров NR;
abgd = 0, 1;
    декрементировать базу вращения глобальных числовых регистров NR.

Продвинуть счетчики циклов (ALC)¶

Операция **ALC** продвигает счетчики циклов. **ALC** может выполняться условно в зависимости от условия передачи управления, закодированного в той же команде.

Язык ассемблера:

```
alc             alcf=fl_f, alct=fl_t
```

fl_f = 0, 1;
    флаг продвижения циклового счетчика при отсутствии факта передачи управления после текущей команды;
fl_t = 0, 1;

флаг продвижения циклового счетчика при наличии факта передачи управления после текущей команды.

Операции асинхронной подкачки в буфер предподкачки массива¶

Операция **FAPB** асинхронно подкачивает в область буфера предподкачки APB.

Язык ассемблера:

```
fapb            {,d=<number>} {,incr=<number>} {,ind=<number>} {,disp=<number>}
                 {,fmt=<number>} {,mrng=<number>} {dcd=<number>}
                 {,asz=<number>} {,abs=<number>}
```

asz

спецификатор размера области назначения в APB; размер области определяется как

area_size = (64 байта)*(2**asz).

Замечание для программиста: диапазон корректных значений asz ограничивается размером APB и для данной реализации включает числа от 0 до 5;

abs

адрес базы области назначения в APB (в терминах 64 байтов):

area_base = (64 байта)*abs;

база области должна быть выровнена до размера области; для последовательности операций асинхронной программы области APB должны назначаться также последовательно, по возрастающим адресам; области, назначенные для разных операций, не должны перекрываться;

mrng

кодирует размер записей, читаемых в область APB, и для всех значений, кроме 0, содержит количество байтов; значение 0 кодирует 32 байта; размер записи также определяет максимальное количество байтов, читаемых из области APB последующими операциями MOVAx; на соотношение величины mrng и используемых адресов обращения в память накладываются аппаратные ограничения, описанные ниже. Длина записи APB:

length = (fapb.mrng != 0) ? fapb.mrng : 32;

fmt

формат элемента массива; он используется:

a) для вычисления текущего значения индекса
b) для проверки выравнивания эффективного адреса;

если читаемый фрагмент памяти в действительности содержит несколько значений различного формата, поле fmt должно кодировать, по крайней мере, самый длинный из них (или длиннее), иначе соответствующая операция MOVAx может не выполниться;

d

ссылка на дескриптор, то есть номер j регистра AADj;

ind

ссылка на значение начального индекса (init_index), то есть номер j регистра AAINDj;

init_index = AAIND<ind>;

заметим, что AAIND0 всегда содержит 0 и не может быть перезагружен чем либо иным;

incr

ссылка на значение приращения (increment), то есть номер j регистра AAINCRj; заметим, что AAINCR0 всегда содержит 1 и не может быть перезагружен чем либо иным;

cd

флаг отключения кэш-памятей:

0 - все кэши подключены;
1 - резерв;
2 - отключен DCACHE2;
3 - отключены DCACHE2, ECACHE.

Начать предподкачку массива (BAP)¶

Операция **BAP** («begin array prefetch») начинает выполнение программы предподкачки массива, подготовленной операцией **LDISP**.

Язык ассемблера:

```
bap
```

Остановить предподкачку массива (EAP)¶

Операция **EAP** («end array prefetch») завершает выполнение программы предварительной выборки массива, подготовленной операцией **LDISP**.

Язык ассемблера:

```
eap
```

Операции пересылки буфера предподкачки массива¶

```
MOVAB           -rd  пересылка массива в формате байт без знака
 MOVAH           -rd  пересылка массива в формате полуслово без знака
 MOVAW           -rd  пересылка массива в формате одинарное слово
 MOVAD           -rd  пересылка массива в формате двойное слово
 MOVAQ           -rq  пересылка массива в формате квадро слово
```

Операции **MOVAx** пересылают предподкаченные элементы массива из буфера предподкачки массива APB в регистровый файл RF, обеспечивающий данные для арифметической обработки в цикле.

Язык ассемблера:

```
movab           { be=NUM, } { area=NUM, } { ind=NUM, } { am=NUM, } dst
 movah           { be=NUM, } { area=NUM, } { ind=NUM, } { am=NUM, } dst
 movaw           { be=NUM, } { area=NUM, } { ind=NUM, } { am=NUM, } dst
 movad           { be=NUM, } { area=NUM, } { ind=NUM, } { am=NUM, } dst
 movaq           { be=NUM, } { area=NUM, } { ind=NUM, } { am=NUM, } dst
```

Разные операции¶

Ожидание исполнения предыдущих операций (WAIT)¶

Операции **WAIT** обеспечивают возможность ожидания опустошения части или всего конвейера перед выдачей команды, которая содержит операцию **WAIT**.

Язык ассемблера:

```
wait            NUM
```

Операция вставить пустые такты (BUBBLE)¶

Операция **BUBBLE** вставляет некоторое число пустых тактов.

Язык ассемблера:

```
nop             { NUM }
```

Операции записи в регистры AAU¶

```
STAAW + MAS     pps  запись в регистр 32
 STAAD + MAS     ppd  запись в регистр 64
 STAAQ + MAS     ppq  запись в регистр 128
```

Язык ассемблера:

```
apurw           src3, aau_reg
 apurwd          src3, aau_reg
 apurwq          src3, aau_reg
```

Названия операций apurw(d/q) имеют синонимы aaurw(d/q), которые можно использовать наравне с основными именами.

Операции считывания регистров AAU¶

```
LDAAW + MAS     pps     считать регистр 32
 LDAAD + MAS     ppd     считать регистр 64
 LDAAQ + MAS     ppq     считать регистр 128
```

Язык ассемблера:

```
apurr           aau_reg, dst
 apurrd          aau_reg, dst
 apurrq          aau_reg, dst
```

Названия операций apurr(d/q) имеют синонимы aaurr(d/q), которые можно использовать наравне с основными именами.

Операции записи в управляющие регистры¶

```
RWs             -sr  запись в регистр состояния 32
 RWd             -dr  запись в регистр состояния 64
```

Язык ассемблера:

```
rws             src2, state_register
 rwd             src2, state_register
```

Операции считывания управляющих регистров¶

```
RRs             r-s  считать регистр состояния 32
 RRd             r-d  считать регистр состояния 64
```

Язык ассемблера:

```
rrs             state_register, dst
 rrd             state_register, dst
```