
Руководство по эффективному
программированию на платформе
«Эльбрус»

Выпуск 1.2

АО «МЦСТ»

февр. 28, 2024

1	Введение в платформу «Эльбрус»	3
1.1	Предисловие	3
1.2	Система программирования	4
1.3	Опции компиляции	5
1.3.1	Уровни оптимизации lcc	5
1.3.2	Важные для lcc опции	6
1.3.3	Общие опции	7
1.3.4	Все опции	8
2	Работа с платформой	9
2.1	Основные принципы	9
2.1.1	Распознать архитектуру	9
2.1.2	Справочные данные	10
2.2	Демонстрация ассемблера	10
2.2.1	Основные операции	10
2.2.2	Дизассемблер	13
2.2.3	Вызов функций	14
2.2.4	Чтение и запись в память	17
2.2.5	Циклы	20
2.2.6	Условный код	23
2.2.7	Переходы и вызовы по косвенности	25
2.3	Работа в gdb	32
2.3.1	SIGILL как сигнал об ошибках	32
2.3.2	Отладка оптимизированного кода	33
2.3.3	Пример сессии gdb	33
2.4	Прочее	36
2.4.1	Отладка ядра	36
2.4.2	Модификация запуска задач	36
3	Отличия в интерфейсах	37
3.1	Совместимость с компиляторами	37
3.1.1	Конструкции языка	37
3.1.1.1	Variable length array inside a struct	37
3.1.1.2	Nested functions	37
3.1.2	gcc builtins	37
3.1.3	Прагмы	38
3.1.4	OpenMP	38

3.1.4.1	Возможности	38
3.1.4.2	Ограничения	38
3.1.4.3	Справочный файл	39
3.2	Системные интерфейсы	39
3.2.1	makecontext	39
3.2.2	freecontext	39
3.2.3	swapcontext/setcontext	40
3.2.3.1	Особенность переключения контекстов	40
3.2.4	sys_get_backtrace/sys_set_backtrace	40
3.2.5	sys_access_hw_stacks	42
4	Введение в архитектуру «Эльбрус»	43
4.1	Введение	43
4.2	Характеристики микропроцессоров «Эльбрус»	43
4.3	Переход от скалярных процессоров к конвейеризированным и суперскалярным	45
4.4	Широкая команда (ШК)	46
4.5	Определяющие свойства архитектуры «Эльбрус»	49
4.6	Принцип использования параллельности операций для VLIW и OOOSS	50
5	Анализ производительности программ	53
5.1	Утилита perf	54
5.2	Утилита dprof	55
5.3	Просмотр привязки к исходнику	58
6	Повышение производительности	61
6.1	Планирование кода	61
6.2	Inline - подстановки	64
6.3	Программная конвейеризация	68
6.3.1	Важность конвейеризации для VLIW и OOOSS	69
6.3.2	Пример ручной конвейеризации цикла	70
6.3.3	Логическая и физическая итерации	76
6.3.4	Примеры рекуррентности	77
6.3.5	Управление конвейеризацией	79
6.3.6	Пример с конвейеризацией цикла	79
6.4	Слияние альтернатив условий	81
6.4.1	Важность слияния кода для VLIW и OOOSS	83
6.4.2	If-Conversion	83
6.4.3	Спекулятивный режим	87
6.4.4	Итог слияния примера со сложным условным кодом	88
6.4.5	Управление слиянием кода	89
6.4.6	Пример на слияние кода	89
6.4.7	Последствия явного спекулятивного режима	91
6.5	Конфликты операций работы с памятью. Анализ указателей. Разрыв зависимостей	93
6.5.1	Виды зависимостей между операциями	93
6.5.2	Зависимости по памяти	94
6.5.3	Важность разрешения зависимостей по памяти	95
6.5.4	Способы разрыва зависимостей по памяти	95
6.5.4.1	Статический анализ конфликтов по памяти	96
6.5.4.2	Динамический разрыв конфликтов по памяти	98
6.6	Предварительная подкачка данных	101
7	Использование оптимизированных библиотек	105
7.1	Общие сведения	105
7.2	Состав	105
7.3	Информационная система	106

7.4	Примеры использования	106
7.4.1	Умножение векторов	106
7.4.2	Умножение матриц	108
8	Рекомендации по оптимизации программ под архитектуру Эльбрус	115
8.1	Рекомендации по работе со структурами данных	115
8.2	Виды локальности данных	116
8.3	Рекомендации по оптимизации процедур	117
8.3.1	Анализ процедуры: начальный этап	118
8.3.2	Короткая ациклическая процедура (не более 30 тактов)	118
8.3.3	Процедура с горячими простыми циклами/гнездами циклов	118
8.3.4	Сложный цикл с управлением, гнездо с управлением	119
8.3.5	Громоздкая процедура	119
8.3.6	Процедура с превалирующим оператором switch	119
8.3.7	Библиотечная процедура	120
9	Интерфейсные программные соглашения	121
9.1	Модель памяти	121
9.1.1	Сегменты программы	121
9.1.2	Организация обращения в память	122
9.1.3	Семантические модели организации памяти	123
9.1.4	Распределение данных	123
9.1.4.1	Глобальные переменные	123
9.1.4.2	Локальные статические данные	124
9.1.4.3	Константы	124
9.1.4.4	Динамически выделенные объекты	124
9.1.4.5	Локальные автоматически переменные	125
9.2	Представление данных	125
9.2.1	Отображение целых типов	127
9.2.2	Отображение вещественных типов	128
9.2.3	Отображение указательных типов	129
9.2.4	Агрегатные типы	129
9.2.4.1	Тип массива	130
9.2.4.2	Тип структуры	130
9.2.4.3	Тип объединение	132
9.2.4.4	Битовые поля	133
9.3	Описание регистров	135
9.3.1	Рабочие регистры	135
9.3.1.1	Механизм регистровых окон	135
9.3.1.2	Пространство регистров текущего окна. Программные соглашения использования пространства регистров текущего окна	136
9.3.1.3	Пространство регистров подвижной базы. Программные соглашения использования базированных регистров	137
9.3.1.4	Пространство глобальных регистров. Программные соглашения использования глобальных регистров	138
9.3.2	Предикатные регистры	138
9.3.3	Регистры управления	139
9.3.4	Специальные регистры	139
9.4	Локальный стек	141
9.4.1	Процедурный фрагмент стека	141
9.4.2	Доступ к компонентам фрагмента процедурного стека	142
9.5	Процедурный механизм	142
9.5.1	Передача параметров	143
9.5.2	Возврат значения	145

9.5.3	Процедурный переход	145
9.5.4	Возврат из процедуры	147
10	Команды микропроцессора	149
10.1	Структура описания операции	149
10.2	Спекулятивное исполнение	150
10.3	Обзор целочисленных операций	151
10.3.1	Операции сложения, вычитания, обратного вычитания	151
10.3.2	Операции умножения	151
10.3.3	Операции деления и вычисления остатка	152
10.3.4	Операции сравнения целых чисел	153
10.3.5	Логические поразрядные операции	154
10.3.6	Операции «взять поле произвольной длины»	155
10.3.7	Операции «вставить поле»	156
10.3.8	Расширение знаком или нулем	156
10.3.9	Выбор из двух операндов	157
10.4	Обзор вещественных скалярных операций	157
10.4.1	Операции умножения на целую степень двойки	158
10.4.2	Операции вычисления квадратного корня	158
10.4.3	Скалярные операции преобразования формата	159
10.5	Предикатные операции	160
10.5.1	Операции вычисления предикатов	160
10.5.2	Вычисление первичного логического предиката (Evaluate Logical Predicate - ELP)	161
10.5.3	Направить логический предикат (Route Logical Predicate - RLP)	161
10.5.4	Условие для операции MERGE (Merge Condition - MRGC)	161
10.5.5	Вычисление логического предиката (Calculate Logical Predicate - CLP)	162
10.5.6	Условная пересылка логического предиката (Conditional Move Logical Predicate - MLP)	162
10.6	Операции обращения в память	162
10.6.1	Операции считывания из незащищенного пространства	163
10.6.2	Операции записи в незащищенное пространство	163
10.6.2.1	Операции считывания в режиме -mptr32	163
10.6.2.2	Операции записи в режиме -mptr32	164
10.6.3	Операции обращения к массиву	164
10.6.3.1	Операции считывания массива	164
10.6.3.2	Операции записи в массив	164
10.7	Операции преобразования адресных объектов	165
10.7.1	Взять указатель стека (GETSP)	165
10.7.2	Переслать тэгированное значение (MOVt)	165
10.8	Операции доступа к регистрам состояния	165
10.8.1	Операции управления регистровым окном	166
10.9	Операции подготовки передачи управления	166
10.9.1	Подготовка перехода по литеральному смещению (DISP)	168
10.9.2	Подготовка перехода по динамическому смещению (GETPL)	169
10.9.3	Подготовка перехода по метке из регистрового файла RF (MOVtd)	169
10.9.4	Подготовка возврата из процедуры (RETURN)	169
10.9.5	Подготовка программы предподкачки массива (LDISP)	170
10.9.6	Предварительная подкачка кода по литеральному смещению (PREF)	170
10.10	Операции передачи управления (CT)	170
10.10.1	Подготовленный переход (BRANCH)	171
10.10.2	Непосредственный переход (IBRANCH)	171
10.10.3	Операция CALL	171
10.10.4	Возврат из аппаратного обработчика прерываний (DONE)	172
10.11	Операции поддержки наложений цикла	172

10.11.1	Операции Set BR	172
10.11.2	Продвинуть базу вращения числовых регистров NR (ABN)	172
10.11.3	Продвинуть базу вращения предикатных регистров PR (ABP)	173
10.11.4	Продвинуть базу вращения глобальных числовых регистров NR (ABG)	173
10.11.5	Продвинуть счетчики циклов (ALC)	173
10.11.6	Операции асинхронной подкачки в буфер предподкачки массива	173
10.11.7	Начать предподкачку массива (BAP)	175
10.11.8	Остановить предподкачку массива (EAP)	175
10.11.9	Операции пересылки буфера предподкачки массива	175
10.12	Разные операции	175
10.12.1	Ожидание исполнения предыдущих операций (WAIT)	175
10.12.2	Операция вставить пустые такты (BUBBLE)	176
10.12.3	Операции записи в регистры AAU	176
10.12.4	Операции считывания регистров AAU	176
10.12.5	Операции записи в управляющие регистры	176
10.12.6	Операции считывания управляющих регистров	177
11	Режим Безопасных Вычислений	179
11.1	Назначение и общий принцип работы РБВ	179
11.2	Устройство указателя в РБВ	180
11.3	Тэги	180
11.4	Дополнительные ограничения на исходные коды	181
11.5	Частичные аналоги РБВ	181
11.6	Пример сборки и исполнения теста	181
11.7	Ассемблер теста в РБВ	182
12	Планирование последовательности операций	185
12.1	Зависимости по передаче данных и управления	185
12.1.1	Передача данных через регистровый файл	185
12.1.1.1	Общие положения	185
12.1.1.2	Времена исполнения операций	188
12.1.1.3	Зависимости между операциями	191
12.1.2	Передача предикатов через предикатный файл	193
12.1.3	Передача атрибутов СТ-операций через регистры CTPR	195
	Алфавитный указатель	199

Выпуск 1.2

Мурад Нейман-заде.
Сергей Королёв.

Распространяется по лицензии [Creative Commons BY 4.0 \(CC BY 4.0\)](https://creativecommons.org/licenses/by/4.0/).
Копирайт © 2021-2024, АО «МЦСТ».

Введение в платформу «Эльбрус»

1.1 Предисловие

Данное руководство содержит основные материалы для обучения программированию на платформе Эльбрус.

Руководство применимо на любом варианте Linux-подобной операционной системы.

Первый блок содержит материалы, чтобы адаптировать ПО к Эльбрусу и добиться его корректной работы.

Текущая глава содержит общие вводные данные.

Глава *Работа с платформой* описывает приёмы работы на этапе компиляции программ, примеры ассемблера, отладку.

Глава *Отличия в интерфейсах* перечисляет все внешние особенности, присущие платформе, которые требуют изменений в реализации программ.

Следующий блок содержит материалы для тех, кто будет работать над производительностью программ.

Глава *Введение в архитектуру «Эльбрус»* описывает теоретические основы архитектуры платформы.

Глава *Анализ производительности программ* даёт инструкции о том, как оценить эффективность тех или иных преобразований над исследуемой программой.

Глава *Повышение производительности* описывает техники для наиболее эффективного использования аппаратуры Эльбрус: ускорение за счёт распараллеливания исполнимого кода на уровне инструкций.

Глава *Использование оптимизированных библиотек* описывает использование компонент с высокоэффективными алгоритмами для платформы.

Глава *Рекомендации по оптимизации программ под архитектуру Эльбрус* описывает эмпирические правила для работы с производительностью в типовых ситуациях.

Справочный блок наиболее полно описывает детали реализации. Его потребуется изучить для написания архитектурно-зависимого кода.

Интерфейсные программные соглашения

Команды микропроцессора

Планирование последовательности операций

1.2 Система программирования

Основные компоненты системы программирования:

- `lcc` — оптимизирующий компилятор с языков C, C++;
- `glibc` — библиотека языка Си;
- `libstdc++` — библиотека языка C++.
- `binutils` — набор утилит для обращения с бинарным кодом (`ar`, `ld`, `as`, `nm`, `ranlib`, `strip` и др.);
- `gdb` — отладчик.

Фундаментом программной платформы Эльбрус является оптимизирующий компилятор. Производительность достигается не только за счёт аппаратуры, но и за счёт компилятора, который строит эффективный код.

Платформы Out-Of-Order Superscalar, такие как x86 или arm, ускоряют код за счёт динамического планирования инструкций, которое выполняет аппаратура. Этот процесс практически скрыт от программиста. На данных платформах оптимизирующий компилятор не играет такой существенной роли в производительности задач, как на Эльбрусе.

Ниже перечислены основные стандарты языков, поддерживаемые в текущей версии `lcc` (1.24).

Для языка C:

- номинальная совместимость с `gcc-7.3.0`;
- стандарт C90 (ANSI/ISO 9899:1990) поддержан полностью;
- стандарт C99 (ISO/IEC 9899:1999 as modified by Technical Corrigenda 1 through 3) поддержан полностью;
- стандарт C11 (ISO/IEC 9899:2011) поддержан полностью, за исключением необязательного расширения `_Atomic`.

По умолчанию включен режим `-std=gnu11` (язык C11 с `gnu`-расширениями).

Для языка C++:

- номинальная совместимость с `g++-7.3.0`;
- библиотека `libstdc++` от `gcc-7.3.0`;
- технология `zero cost exceptions (0eh)`;
- стандарт C++03 (ISO/IEC 14882:2003) поддержан полностью;
- стандарт C++11 (ISO/IEC 14882:2011) поддержан полностью;
- стандарт C++14 (ISO/IEC 14882:2014) поддержан полностью.

По умолчанию включен режим `-std=gnu++14` (язык C++14 с gnu-расширениями).

С выходом новых версий `lcc` повышается и версия соответствующего ему `gcc`. Подробное описание всех опций `lcc` приведено в документации, идущей вместе с компилятором:

`/opt/mcst/doc/lcc_options.htm`

Многие элементы тулчейна являются портом компонент GNU. Данный подход позволяет добиться максимальной совместимости со сборкой существующих программ, написанных для Linux или POSIX и тестируемых на других аппаратных платформах.

В систему программирования входит оптимизирующий компилятор `lfortran`.

- номинальная совместимость с `gfortran-5.5.0`, есть неполная совместимость с `gfortran-6.5.0` и `gfortran-7.3.0`;
- стандарт Fortran 95 (final draft ISO/IEC 1539-1:1997) поддержан полностью;
- стандарт Fortran 2003 (final draft ISO/IEC 1539-1:2004(E)) поддержка экспериментальная;
- Стандарт Fortran 2008 (latest draft, nearly FDIS, 2010-04-27; published 2010-10-06 as ISO/IEC 1539-1:2010) поддержка экспериментальная;
- стандарт Fortran 2018 (latest draft J3/18-007, WG5/N2146, 28th December 2017) поддержка экспериментальная.

По умолчанию включен режим `-std=gnu` (язык Fortran 95 с gnu-расширениями).

1.3 Опции компиляции

Основная функция оптимизирующего компилятора - преобразования исходного алгоритма программы для построения наиболее эффективного исполнимого кода. Данные преобразования называются оптимизациями. Компилятор применяет оптимизации не произвольным образом, а в составе пакетных наборов, согласованных между собой. Данные наборы также называются уровнями оптимизации.

Самые низкие уровни включают минимум оптимизаций и выдают код, который проще всего анализировать. Более высокие уровни генерируют более быстрый код. Чем выше уровень, тем больше времени тратится на сам процесс компиляции.

1.3.1 Уровни оптимизации `lcc`

-00

Начальный уровень. Сохраняет соответствие между исходным текстом программы и ее двоичным кодом, что позволяет формировать отладочную информацию для символьного отладчика.

-01

Локальные оптимизации потока данных и управления, не требующие аппаратной поддержки, выполняются в рамках линейного участка.

-02

Внутрипроцедурные оптимизации потока данных и управления, использующие все аппаратные возможности, цикловые оптимизации, слабо увеличивающие размер кода, минимальные межпроцедурные оптимизации.

-03

Все межпроцедурные оптимизации, возможность оптимизации в режиме «вся программа», все цикловые оптимизации.

-O4

Включает все предыдущие и дополнительные агрессивные оптимизации. Стоит пробовать экспериментально. Может приводить как к повышению производительности, так в некоторых случаях и к деградации производительности.

Для генерации кода с привязкой к исходному тексту нужно использовать опции `-O0 -g`.

1.3.2 Важные для Icc опции

Опции компиляции, которые присутствуют только в `icc`. Их можно добавлять независимо от основных оптимизаций `-O*`. Они задействуют дополнительные техники оптимизации, которые налагают ограничения на исходный код либо на процесс сборки.

Опции профилирования:

`-fprofile-generate[=<path>]`
генерировать компиляторный профиль;

`-fprofile-use[=<file>]`
использовать компиляторный профиль.

Опции данной секции реализуют технику двухфазной компиляции: на первой фазе программа собирается в инструментирующем режиме. Затем программа выполняется на тестовых данных или в выбранном сценарии использования, который требует ускорения. Программа выполняется по сценариям (одному или нескольким), в результате чего в файле формируется профиль исполнения. Данный профиль затем используется для второй фазы компиляции с помощью `-fprofile-use`.

Основным требованием для применения данного режима является наличие набора представительных сценариев использования программы. Если в дальнейшем реальное исполнение программы существенно отличается от того варианта, на котором получали профиль, то производительность реального исполнения может значительно ухудшиться.

Статья с большим количеством информации о профилировании в документации на компилятор: </opt/mcst/doc/profile.html>

Опции режима «вся программа»:

`-fwhole`
опция компиляции в режиме «вся программа». Несовместима с позиционно-независимым кодом, в частности, с динамическими библиотеками `.so`.

`-fwhole-shared`
опция компиляции в режиме «вся программа», совместимая с позиционно-независимым кодом. Требуется работы с `-fPIC` либо `-fPIE`.

Режим «вся программа» позволяет выполнять межпроцедурные оптимизации. В этом режиме анализируемый контекст не ограничивается единственной функцией.

Опции необходимо подавать как при генерации объектных файлов `.o`, так и при линковке итогового файла (исполнимого либо динамической библиотеки).

Основным требованием для данных опций является необходимость разделять цели в сборочной системе программы, и для разных целей по-своему модифицировать флаги сборки:

- для динамических библиотек использовать `-fPIC -fwhole-shared`;
- для исполнимых файлов использовать `-fwhole` либо `-fPIE -fwhole-shared`;

Если собираемый пакет содержит одновременно исполнимые файлы и библиотеки, то глобально на уровне CFLAGS возможно выставить только сочетание `-fPIC -fwhole-shared`.

Примечание: Опция `-fwhole` работает с уровнем оптимизации `-O1` и выше, а также несовместима с опцией генерации отладочной информации `-g`.

1.3.3 Общие опции

Ниже представлены базовые опции компилятора, характерные для всех архитектур.

Опции управления процессом компиляции:

- `-c`
компилировать до объектного файла;
- `-S`
записать в файл сгенерированный ассемблер;
- `-o<file> (-o <file>)`
задать название бинарного файла;
- `-v (--verbose)`
опция подробного вывода.

Опции компиляции целевой архитектуры:

- `-m32`
компилировать в режиме 32-битной адресации;
- `-m64`
компилировать в режиме 64-битной адресации;
- `-m128`
компилировать в режиме 128-битной защищённой адресации с аппаратным контролем доступа к объектам.

В данном режиме указатель на данные и функции занимает 128 бит. В нём содержится 64-разрядный адрес объекта, его размер (не более 4 GB) и позиция указателя внутри объекта. Режим усиливает контроль памяти программ при исполнении.

Опции управления режимом компиляции:

- `-fPIC (-fpic)`
генерировать позиционно-независимый код;
- `-fPIE (-fpie)`
генерировать позиционно-независимый код для исполнимых файлов.

Опции управления препроцессированием:

- I <dir> (-I<dir>)
указать каталог с заголовочными файлами;
- D/-U
взвести/убрать макрос;
- E
препроцессировать;
- C
сохранить комментарии при процессировании;
- H
вывести все подхваченные хедеры при компиляции файла.

Опции управления линковкой:

- L<dir> (-L <dir>)
подать каталог с библиотеками;
- l<name> (-l <name>)
подать имя конкретной библиотеки;
- shared (--shared)
генерировать динамическую (разделяемую) библиотеку;
- static
генерировать статический исполнимый файл.

1.3.4 Все опции

Все опции компилятора перечислены здесь: /opt/mcst/doc/lcc_options.html

2.1 Основные принципы

Система программирования и все сборочные инструменты ориентированы на максимальную совместимость с существующей экосистемой. Во многих случаях программа для сборки и корректной работы вообще не потребует какой-либо адаптации. В случае правок зачастую они сводятся к тому, что поведение архитектуры описывается тем же кодом, что используется для `gcc` и `x86_64`. В этом случае достаточно распознать архитектуру.

При нативной сборке можно указывать `gcc` и `g++` в качестве компиляторов `C` и `C++`. Это синонимы для компилятора `lcc` в режимах `C` и `C++`.

Этапы сборки open source программ те же, что и для других архитектур:

- конфигурация (`./configure`, `cmake`, `mvn`, `scons` ...)
- компиляция (пример - `make`)
- установка (пример - `make install`)

2.1.1 Распознать архитектуру

Если пакет собирается с помощью `autotools`, вам потребуется вариант файлов `config.guess` и `config.sub`, в которых есть поддержка `e2k`. Вы можете взять их здесь:

- `/usr/share/automake/config.sub`
- `/usr/share/automake/config.guess`

Замените `config.guess` и `config.sub` из ваших исходников на экземпляры выше.

Для платформы Эльбрус предусмотрен макрос языка `C`:

```
__e2k__
```

Компилятор `lcc` взводит свой макрос (для архитектур Эльбрус и Спарк):

```
__LCC__
```

Примеры использования:

```
#if (defined __LCC__) && (! defined __OPTIMIZE__)
    #define MY_MACROS MY_MACROS_LCC
#endif
```

```
#ifdef __e2k__
    #include "implementation_elbrus.h"
#endif
```

Макрос `__LCC__` хранит значение мажорной версии компилятора. Те или иные действия можно задавать только для определённых версий.

```
#if (defined __LCC__) && (__LCC__ <= 123)
    # error "Not supported for lcc-1.23 or earlier."
#endif
```

2.1.2 Справочные данные

Платформа поддерживает 64-битную и 32-битную адресацию. Основной режим системного ПО 64 бита.

Организация байтов — little endian.

Локальный стек в процедурах растёт от больших адресов к меньшим.

Размеры данных подробно описаны в разделе *Интерфейсные программные соглашения*.

2.2 Демонстрация ассемблера

2.2.1 Основные операции

Рассмотрим код с простыми числовыми расчётами, который сможет продемонстрировать основные операции в ассемблере.

Пример 1. арифметические операции

```
1 int f(int a, int b, int c)
2 {
3     int s1, s2, s4;
4     double s3;
5     s1 = a + b;
6     s2 = b - c;
7     s3 = s1 / s2;
8     s4 = s3 * s1;
9     return (int) (s1 * s4);
10 }
```

Для просмотра ассемблера подадим компилятору опцию `-S`:

```
gcc -O3 -S t.c
```

Листинг 1. Ассемблер примера 1:

```
.text
.global f
.type f, #function
.align 8
f:
{
  setwd wsz = 0x4, nfx = 0x1
  return      %ctpr3; ipd 2
  subs,0      %r1, %r2, %g16
  adds,3      %r0, %r1, %g17
}
{
  nop 2
  istofd,3    %g17, %g18
}
{
  nop 7
  sdivs,5     %g17, %g16, %g16
}
{
  nop 2
}
{
  nop 3
  istofd,3    %g16, %g16
}
{
  nop 3
  fmuld,3     %g16, %g18, %g16
}
{
  nop 3
  fdtoistr,3  %g16, %g16
}
{
  nop 5
  muls,3      %g17, %g16, %g16
}
{
  ct    %ctpr3
  sxt,3 0x2, %g16, %r0
}
```

Обратим внимание, что каждая широкая команда в ассемблере помещена в пару фигурных скобок.

Широкая команда — набор операций, которые запускаются процессором параллельно в одном такте.

Большинство операций имеют формат <мнемоника>, <канал> <аргумент>, <аргумент>, ... , <результат>

В качестве аргументов и результатов операций чаще всего выступают регистры. Приведённый пример содержит распространённые виды регистров:

%r0, %r1, %r<N> рабочие регистры, доступные только в текущей процедуре.

%g0, %g1, %g<N> глобальные регистры, доступные всей программе. В примере выступают в качестве временных регистров.

%ctpr1, %ctpr2, %ctpr3 особые регистры, используемые для передачи управления. С их помощью устанавливаются адреса переходов.

Про все виды регистров, правила работы с ними, а также использование регистров в процедурном механизме, подробно рассказывается в описании программных соглашений, раздел *Описание регистров*.

Про регистры передачи управления более подробно рассказывается в описании архитектурных решений, раздел *Определяющие свойства архитектуры «Эльбрус»*, и наиболее детально в разделе *Операции подготовки передачи управления*.

В начале некоторых широких команд можно видеть служебное слово **nop** с параметром. Оно является подсказкой процессору выдержать задержку в несколько тактов до следующей широкой команды. Без таких подсказок код может исполняться с блокировками, и итоговая производительность будет ниже, чем у кода с расставленными **nop**-ами.

В приведенном примере встречаются операции:

setwd задает размер и конфигурацию регистрового окна процедуры; присутствует в первом такте подавляющего большинства процедур.

add арифметическое сложение.

return подготовка возврата из процедуры.

sub арифметическое вычитание.

sdivs целочисленное деление.

fmuld вещественное умножение.

istofd преобразование формата из int в double.

fdtoistr преобразование формата из double в int с обрубанием точности (truncate).

muls целочисленное умножение.

ct передача управления. В данном примере работает совместно с return.

sxt расширение знаком или нулем 32-битного значения до 64-битного.

Арифметические операции имеют разные мнемоники для типов и разрядностей аргументов. Мнемоника модифицируется префиксами и суффиксами. Рассмотрим их на примере команды сложения **add**:

adds 32-разрядные целочисленные аргументы.

addd 64-разрядные целочисленные аргументы.

fadds 32-разрядные вещественные аргументы.

faddd 64-разрядные вещественные аргументы.

Более подробное описание ассемблера находится в разделе *Команды микропроцессора*.

2.2.2 Дизассемблер

Для просмотра дизассемблера объектного или исполняемого файла можно использовать команду `ldis`.

```
ldis ./t.o
```

Возможно включить привязку строк ассемблера к строкам исходного кода. Эту функцию выполняет опция `-gline`.

```
gcc -O3 -gline t.c -c
ldis ./t.o
```

Листинг 2. Использование `ldis` с `-gline`, соответствует ассемблеру в Листинге 1

```
! function 'f', entry = 9, value = 0x000000, size = 0x070, sect = ELF_TEXT num = 1

0000<00000000000000> f:
        ipd 2
        subs,0 %r1, %r2, %g16           ! t.c : 6
        adds,3 %r0, %r1, %g17         ! t.c : 5
        return %ctpr3                 ! t.c : 9
        setwd wsz = 0x4, nfx = 0x1

0001<00000000000020> :nop 2
        istofd,3 %g17, %dg18          ! t.c : 8

0004<00000000000028> :nop 7
        sdivs,5 %g17, %g16, %g16     ! t.c : 7

0012<00000000000030> :nop 2

0015<00000000000038> :nop 3
        istofd,3 %g16, %dg16          ! t.c : 7

0019<00000000000040> :nop 3
        fmuld,3 %dg16, %dg18, %dg16  ! t.c : 8

0023<00000000000048> :nop 3
        fdtoistr,3 %dg16, %g16       ! t.c : 8

0027<00000000000050> :nop 5
        muls,3 %g17, %g16, %g16     ! t.c : 9

0033<00000000000060> :
        ct %ctpr3                     ! t.c : 9
        ipd 3                          ! t.c : 9
        sxt,3 0x2, %g16, %dr0         ! t.c : 9
```

Дизассемблер перед каждой командой показывает дополнительно:

- номер такта от начала процедуры (десятичное число слева);
- IP-адрес команды (шестнадцатеричное число в угловых скобках):
 - в случае объектного файла - относительно начала модуля;
 - в случае исполняемого файла - абсолютный адрес;
 - в случае динамической библиотеки - относительно точки связывания библиотеки.

2.2.3 Вызов функций

Пример 2. Использование функций

```

1      int func_mul(int, int);
2
3      int main(){
4          int a=2,b=11,s=0;
5          s=func_mul(a,b);
6          return s;
7      }
8
9      int func_mul(int x, int y){
10         return x*y;
11     }

```

Ниже приведён ассемблер примера 2 для оптимизаций *-O0* и *-O3*.

Листинг 3. Ассемблер примера 2 с оптимизацией *-O0*:

```

.file    "t.c"
.ignore ld_st_style
.ignore strict_delay
.text
.global main
.type   main, #function
.align  8
main:
{
    setwd wsz = 0x8, nfx = 0x1, dbl = 0x0
    setbn rsz = 0x3, rbs = 0x4, rcur = 0x0
    getspl,0      _f16s,_ltslhi 0xffff0, %r2
}
{
    adds,0,sm     0x0, 0x2, %r3
    adds,1,sm     0x0, 0xb, %r4
    adds,2,sm     0x0, 0x0, %r5
}
{
    sxt,0,sm      0x2, %r3, %b[0]
    sxt,1,sm      0x2, %r4, %b[1]
}
.LCS.1:
{
    nop 4
    disp %ctpr1, func_mul
}
{
    call %ctpr1, wbs = 0x4
}
.LCS.2:
{
    adds,0,sm     0x0, %b[0], %r6
    return       %ctpr3
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```

}
{
  adds,0,sm    0x0, %r6, %r5
}
{
  nop 3
  sxt,0,sm    0x2, %r5, %r0
}
{
  ct    %ctpr3
}
.size    main, .- main
.global func_mul
.type    func_mul, #function
.align  8
func_mul:
{
  setwd wsz = 0x4, nfx = 0x1, dbl = 0x0
}
{
  nop 5
  muls,0      %r0, %r1, %r4
  return      %ctpr3
}
{
  sxt,0,sm    0x2, %r4, %r0
  ct    %ctpr3
}

```

Листинг 4. Ассемблер примера 2 с оптимизацией -03:

```

.file    "t.c"
.ignore ld_st_style
.ignore strict_delay
.text
.global main
.type    main, #function
.align  8
main:
{
  nop 5
  setwd wsz = 0x4, nfx = 0x1, dbl = 0x0
  return    %ctpr3
  addd,0    0x16, 0x0, %r0
}
{
  ct    %ctpr3
}
.size    main, .- main
.global func_mul
.type    func_mul, #function
.align  8

```

(continues on next page)

(продолжение с предыдущей страницы)

```

func_mul:
{
  nop 5
  setwd wsz = 0x4, nfx = 0x1, dbl = 0x0
  return      %ctpr3
  muls,0      %r0, %r1, %g16
}
{
  ct      %ctpr3
  sxt,0 0x2, %g16, %r0
}

```

Листинг 5. Использование ldis. Пример 2, оптимизация -03.

```
ldis ./a.out
```

```

! function 'main', entry = 56, value = 0x0104e8, size = 0x020, sect = ELF_TEXT num = 12
0000<0000000104e8> main: nop 5
                    addd,0 0x16, 0x0, %dr0                ! t.c : 6
                    return %ctpr3                        ! t.c : 6
                    setwd wsz = 0x4, nfx = 0x1, dbl = 0x0
0006<000000010500> :
                    ct %ctpr3                             ! t.c : 6
                    ipd 3                                  ! t.c : 6

! function 'func_mul', entry = 44, value = 0x010508, size = 0x028, sect = ELF_TEXT num = 12
↪12
0000<000000010508> func_mul: nop 5
                    muls,0 %r0, %r1, %g16                ! t.c : 10
                    return %ctpr3                        ! t.c : 10
                    setwd wsz = 0x4, nfx = 0x1, dbl = 0x0
0006<000000010520> :
                    ct %ctpr3                             ! t.c : 10
                    ipd 3                                  ! t.c : 10
                    sxt,0 0x2, %g16, %dr0                ! t.c : 10

```

В листингах ассемблера примера 2 появились новые команды:

setbn установить базу вращения числовых регистров. Эта команда дополняет setwd.

Механизм вращаемых регистров описан в разделе

Пространство регистров подвижной базы. Программные соглашения использования базированных регистров.

Использование вращаемых регистров в процедурном механизме описано в разделе

Процедурный механизм.

getsp выделить или вернуть область в стеке пользователя локального фрейма.

Пользовательский стек и работа с ним описаны в разделе *Локальный стек*.

disp подготовка адреса перехода, в данном случае для операции `call`.

call выполнить вызов функции.

Вызов функции легко заметить по подготовке перехода `disp`:

```
disp %ctpr1, func_mul
```

и последующему вызову функции через `call`:

```
call %ctpr1, wbs = 0x4
```

Обратите внимание, что в результате оптимизаций в режиме `-O3` вызов был заменён на:

```
addd,0 0x16, 0x0, %r0
```

Конечный возвращаемый результат определился статически как `0x16`, или `22` в десятичной системе.

2.2.4 Чтение и запись в память

Пример 3. Чтение и запись в память

```

1 long int global_var;
2 extern void g(int *);
3
4 void f(short int *p)
5 {
6     int local_var = 14;
7
8     g(&local_var);
9     local_var++;
10    (*p)++;
11    g(&local_var);
12    global_var++;
13
14    return;
15 }
```

Покажем на примере, как выглядят обращения в память по глобальным переменным, локальным переменным в стеке и по косвенности.

Для этого примера ассемблер с оптимизацией `-O3` и `-O0` отличаются несущественно, и будет приведен листинг с оптимизацией `-O3`.

Листинг 6. Чтение и запись в память.

```

f:
{
  nop 1
  setwd    wsz = 0x8, nfx = 0x1
  setbn    rsz = 0x3, rbs = 0x4, rcur = 0x0
  disp    %ctpr1, g; ipd 2
```

(continues on next page)

(продолжение с предыдущей страницы)

```

    getsp,0    _f32s,_lts1 0xffffffffe0, %r2
    adds,1    0xe, 0x0, %r3
}
{
    addd,0    %r2, _f64,_lts0 0x20, %r1
}
{
    subd,0    %r1, 0x4, %r4
}
{
    addd,0,sm 0x0, %r4, %b[0]
    stw,2     %r1, _f16s,_lts0lo 0xfffc, %r3
}
.LCS.1:
{
    call     %ctpr1, wbs = 0x4
}
{
    nop 2
    disp    %ctpr1, g; ipd 2
    ldh,0   %r0, 0x0, %r3
    addd,1,sm 0x0, %r4, %b[0]
    ldw,2   %r1, _f16s,_lts0lo 0xfffc, %r5
}
{
    adds,0   %r3, 0x1, %r3
    adds,1   %r5, 0x1, %r4
}
{
    sth,2    %r0, 0x0, %r3
    stw,5    %r1, _f16s,_lts0lo 0xfffc, %r4
}
{
    call     %ctpr1, wbs = 0x4
}
{
    nop 2
    return  %ctpr3; ipd 2
    ldd,0   0x0, [ _f64,_lts0 global_var ], %r0
}
{
    addd,0   %r0, 0x1, %r0
}
{
    nop 1
    std,2    0x0, [ _f64,_lts0 global_var ], %r0
}
{
    ct      %ctpr3
}

```

В примере значения всех трех ячеек памяти увеличиваются на 1. Для этого они считываются операцией ld, увеличиваются с помощью операции add, и записываются операцией st. Суффикс операций ld и st

означает формат данных:

b - 1 байт
h - 2 байта
w - 4 байта
d - 8 байт

Соответствующие операции:

ldb, stb - 1 байт
ldh, sth - 2 байта
ldw, stw - 4 байта
ldd, std - 8 байт

В приведенном примере используются переменные формата *short int* (2 байта), *int* (4 байта) и *long int* (8 байт). Адрес операций ld и st формируется как сумма двух аргументов, как правило, это база адреса и смещение. Третий аргумент операции записи - это записываемое по адресу значение.

Адрес глобальной переменной `global_var` задается в виде символа:

```
ldd,0 0x0, [ _f64,_lts0 global_var ], %r0
```

Кроме имени, в квадратных скобках указаны ключевые слова `_f64,` `_lts0`. Они отображают формат и позицию константного значения (напомним, что адрес глобальной переменной становится константой после линковки).

Локальная переменная `local_var` хранится в стеке. Из-за того, что на нее взят адрес и передан в функцию `g()`, ее нельзя хранить на регистре. Адрес в стеке задается как сумма регистра, хранящего указатель на стек, и смещения.

```
ldw,2 %r1, _f16s,_lts0lo 0xffffc, %r5
```

В начале процедуры видно, как формируется регистр указателя на локальное окно стека:

```
getsp,0      _f32s,_lts1 0xffffffe0, %r2
...
addd,0      %r2, _f64,_lts0 0x20, %r1
```

Здесь операция `getsp` заказывает новую порцию стека размером `0x20` и размещает указатель на новую вершину стека в `%r2`, а в `%r1` заносится «дно» локального окна стека. Подробно механизм описан в разделе *Локальный стек*.

Работа с указателем, переданным в качестве параметра, ведется по регистру `%r0`, содержащему параметр `p`.

```
ldh,0 %r0, 0x0, %r3
```

2.2.5 Циклы

Пример 4. Циклы

```

1 void f(int *v0, int N)
2 {
3     int i;
4     for (i=0; i<N; i++)
5         v0[i] += (v0[i] + 3) * v0[i];
6 }

```

Листинг 7. Пример 3, оптимизация -O0:

```

f:
{
    setwd     wsz = 0x5, nfx = 0x1
}
{
    adds,0,sm 0x0, 0x0, %r4
}
.L4:
{
    nop 4
    cmplsb,0 %r4, %r1, %pred0
    disp     %ctpr1, .L6; ipd 2
}
{
    ct       %ctpr1 ? ~%pred0
}
.L9:
{
    sxt,0,sm 0x2, %r4, %r5
    sxt,1,sm 0x2, %r4, %r6
    sxt,2,sm 0x2, %r4, %r7
    adds,3   %r4, 0x1, %r4
    disp     %ctpr1, .L4; ipd 2
}
{
    shld,0   %r5, 0x2, %r5
    shld,1   %r6, 0x2, %r6
    shld,2   %r7, 0x2, %r7
}
{
    addd,0   %r0, %r5, %r5
    addd,1   %r0, %r6, %r6
    addd,2   %r0, %r7, %r7
}
{
    ldw,0    %r6, 0x0, %r6
    ldw,2    %r7, 0x0, %r8
}
{
    nop 2
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    ldw,0    %r5, 0x0, %r5
  }
  {
    adds,0   %r5, 0x3, %r5
  }
  {
    nop 5
    muls,0   %r5, %r6, %r5
  }
  {
    adds,0   %r8, %r5, %r5
  }
  {
    stw,2,sm %r7, 0x0, %r5
    ct      %ctpr1
  }
.L6:
  {
    nop 5
    return  %ctpr3; ipd 2
  }
  {
    ct      %ctpr3
  }

```

В метке L4 можно увидеть две широкие команды. В первой проверяется попадание в цикл и выполняется подготовка перехода. Во второй команде делается переход по отрицанию условия выхода из цикла в метку L6, находящуюся в голове цикла. Содержимое между метками L9 и L6 является циклом.

Листинг 8. Пример 3, оптимизация `-O3` в сочетании с опцией `-fmax-iter-for-ovlpeel=0`

```

f:
  {
    setwd    wsz = 0x4, nfx = 0x1
    return  %ctpr3; ipd 2
  }
  {
    nop 2
    cmplsb,0 0x0, %r1, %pred0
  }
  {
    ct      %ctpr3 ? ~%pred0
  }
  {
    setwd    wsz = 0x10, nfx = 0x1
    setbn    rsz = 0xb, rbs = 0x4, rcur = 0x0
    return  %ctpr3; ipd 2
    sxt,0,sm 0x6, %r1, %g16
    addd,1   0x0, _f64,_lts1 0x2dff2d00000000, %g17
    addd,2,sm 0x1, 0x0, %g18
    addd,3   0x0, 0x0, %g19
  }

```

(continues on next page)

(продолжение с предыдущей страницы)

```

{
  nop 1
  disp      %ctpr1, .L75; ipd 2
  cmplsb,0,sm      0x0, %r1, %pred0
  addd,1,sm 0x0, 0x0, %b[15]
  aaurwd,2  %r0, %aad0
  aaurwd,5  %g19, %aasti1
}
{
  insfd,0,sm      %g17, _f32s,_lts0 0x8800, %g16, %g16 ? %pred0
  insfd,1,sm      %g17, _lit32_ref,_lts0 0x8800, %g18, %g16 ? ~%pred0
}
{
  nop 3
  rwd,0      %g16, %lsr
}
.L75:
{
  loop_mode
  alc      alcf=1, alct=1
  abn      abnf=1, abnt=1
  ct      %ctpr1 ? %NOT_LOOP_END
  muls,0,sm %g17, %b[10], %b[1]
  addd,1,sm 0x4, %b[15], %b[13]
  adds,2,sm %b[8], 0x3, %g17
  ldw,3,sm %r0, %b[17], %b[0] ? %pcnt12
  adds,4,sm %b[22], %b[13], %g16
  staaw,5  %g16, %aad0[ %aasti1 ]
  incr,5   %aaincr0
}
{
  setwd    wsz = 0x4, nfx = 0x1
  adds,0   0x0, 0x0, %g16
}
{
  ct      %ctpr3
  aaurw,2  %g16, %aabf0
}

```

При оптимизации *-O3* для циклов компилятор строит более компактный код. Дополнительная опция *-fmax-iter-for-ovlpeel=0* применяется при компиляции данного примера, чтобы получить более наглядный и читаемый листинг. Значение этой опции описано в разделе *Программная конвейеризация*.

В листинге появились новые команды и служебные слова:

insf команда «вставить битовое поле»; формирует значение из двух регистров, заданных в первом и третьем аргументах, управляется значением, заданным во втором аргументе. В данном примере формирует значение, составленное из старших 32 бит одного регистра и младших 32 бит другого.

rwd операции записи в специальные регистры. В данном случае происходит запись в регистр управления циклом **lsr**.

aaurwd операции записи в регистры описания массивов данных.

loop_mode метка, которая говорит о том, что в пределах цикла работает аппаратная поддержка

счетчика цикла. Не является самостоятельной операцией.

alc продвинуть счетчик цикла (advance loop counter).

abn продвинуть вращаемые регистры (advance base numeric).

staa incr

записать значение в массив и продвинуть указатель на фиксированную величину. Возможны только в паре.

Специальный регистр `%lsr` содержит в себе счётчик цикла, который декрементируется на каждой итерации. При исчерпании (обнулении) счетчика условие `%NOT_LOOP_END` становится ложным, а переход в голову цикла, находящийся под этим условием, не срабатывает.

Тело цикла компактно упаковано в одну широкую команду, и исполняется с темпом 1 итерация за 1 такт. Примененная техника оптимизации называется *конвейеризацией* и описана в разделе *Программная конвейеризация*.

Работа с массивами, или в более общем виде работа с чтениями и записями по регулярно изменяющимся адресам, описана в разделе *Предварительная подкачка данных*.

2.2.6 Условный код

Пример 5. Операции под условием

```

1 void f(int c, int* p)
2 {
3
4     if (c>0)
5     {
6         (*p) = (*p) * (*p);
7     }
8     else
9     {
10        (*p) = -(*p);
11    }
12
13 }
```

Условный код существенно изменяется, если применять оптимизацию `-O2` и выше. Рассмотрим код приведенного примера для `-O1` и `-O3`.

Листинг 9. Пример 5, оптимизация `-O1`

```

f:
{
    nop 1
    setwd    wsz = 0x4, nfx = 0x1
    disp    %ctpr1, .L6; ipd 2
}
{
    nop 2
    cmplesb,0 %r0, 0x0, %pred0
}
{
    ct      %ctpr1 ? ~%pred0
```

(continues on next page)

(продолжение с предыдущей страницы)

```

}
{
  nop 2
  ldw,0    %r1, 0x0, %g16
}
{
  subs,0   0x0, %g16, %g16
}
{
  stw,2    %r1, 0x0, %g16
}
.L25:
{
  nop 5
  return  %ctpr3; ipd 2
}
{
  ct      %ctpr3
}
.L6:
{
  nop 2
  disp   %ctpr1, .L25; ipd 2
  ldw,0  %r1, 0x0, %g16
}
{
  nop 3
  muls,0 %g16, %g16, %g16
}
{
  ct      %ctpr1
  stw,2   %r1, 0x0, %g16
}
}

```

В приведенном коде есть новые операции:

cmpl операция сравнения двух аргументов. Результат операции сравнения записывается в предикатный регистр `%pred<N>`, где `N` может быть от 0 до 31. Это отличает архитектуру Эльбрус от многих других архитектур, в которых сравнение вырабатывает специальный регистр флагов. Предикатные регистры хранят значения 0 или 1, и могут использоваться для:

- передачи управления,
- управления исполнением операции,
- вычисления других предикатных регистров.

disp операция подготовки перехода. Как мы уже видели, операция может подготовить адрес для вызова процедуры, но в данном примере с ее помощью подготавливается переход по локальным меткам `.L6` и `.L25`.

ct передача управления. С ее помощью производится не только возврат из процедуры, но и заранее подготовленный переход по локальной метке.

Листинг 10. Пример 5, оптимизация -03

```
f:
{
  nop 2
  setwd      wsz = 0x4, nfx = 0x1
  return    %ctpr3; ipd 2
  ldw,0,sm  %r1, 0x0, %g17
  ldw,2,sm  %r1, 0x0, %g16
}
{
  nop 1
  muls,0,sm %g16, %g16, %g16
  subs,1,sm 0x0, %g17, %g17
}
{
  nop 1
  cmplesb,0 %r0, 0x0, %pred0
}
{
  ct      %ctpr3
  stw,2   %r1, 0x0, %g16 ? ~%pred0
  stw,5   %r1, 0x0, %g17 ? %pred0
}
```

Здесь можно увидеть другой способ использования предикатных регистров - управление исполнением операций.

```
stw,2 %r1, 0x0, %g16 ? ~%pred0
```

После операции записи указывается символ `?`, за которым следует предикатный регистр. Операция записи будет исполнена, если значение регистра равно `1`, и не будет исполнена, если значение равно `0`. Если перед предикатным регистром указать символ отрицания `~`, то значение управляющего предикатного регистра будет использовано инвертированным образом. Управляющий предикат также иногда называют *квалифицирующим* (Qualifying Predicate).

Отсюда вытекает логика оптимизированного листинга: в зависимости от условия `c>0` выполнится операция записи с необходимым значением, сформированном в регистрах `%g16` либо `%g17` соответственно.

Код, в котором передача управления заменена на управляющие предикаты, называется *предикатным*. Техника преобразования кода от условного к предикатному описана в разделе *Слияние альтернатив условий*.

2.2.7 Переходы и вызовы по косвенности

Пример 6. Конструкция switch

```
1 void f( int v, float *arr)
2 {
3
4   switch(v) {
5     case 10:
6     case 20:
7       arr[0] *= 2.0;
8       break;
```

(continues on next page)

(продолжение с предыдущей страницы)

```

9  case 11:
10 case 21:
11     arr[1] *= 3.0;
12     break;
13
14 case 12:
15 case 22:
16     arr[2] *= 4.0;
17     break;
18
19 case 13:
20 case 23:
21     arr[3] *= 5.0;
22     break;
23
24 case 14:
25 case 24:
26     arr[4] *= 6.0;
27     break;
28
29 case 15:
30 case 25:
31     arr[5] /= 7.0;
32     break;
33
34 default:
35     arr[6] += 1.0;
36 }
37 return;
38 }

```

Для этого примера код на *-O1* и *-O3* не будет существенно отличаться. Приведем ассемблер для уровня оптимизации *-O3*.

Листинг 10. Пример 6, оптимизация *-O3*

```

f:
{
    setwd    wsz = 0x4, nfx = 0x1
    disp    %ctpr3, .L72; ipd 2
    subs,3  %r0, 0xa, %g16
}
{
    cmpbesb,3 %g16, 0xf, %pred0
    sxt,4,sm 0x2, %g16, %g16
}
{
    shld,3,sm %g16, 0x3, %g16
}
{
    ldd,3,sm %g16, [ _f64,_lts0 .T.1 ], %r0
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```

{
    ct      %ctpr3 ? ~%pred0
}
{
    nop 2
}
{
    nop 7
    movtd,0,sm      %r0, %ctpr1; ipd 2
}
nop
{
    ct      %ctpr1
}
.LSC.1:
{
    nop 2
    return  %ctpr3; ipd 2
    ldw,0   %r1, 0x0, %g16
}
{
    nop 3
    fmul,0  %g16, _f32s,_lts0 0x40000000, %g16
}
{
    ct      %ctpr3
    stw,2   %r1, 0x0, %g16
}
.LSC.2:
{
    nop 2
    return  %ctpr3; ipd 2
    ldw,0   %r1, 0x4, %g16
}
{
    nop 3
    fmul,0  %g16, _f32s,_lts0 0x40400000, %g16
}
{
    ct      %ctpr3
    stw,2   %r1, 0x4, %g16
}
.LSC.3:
{
    nop 2
    return  %ctpr3; ipd 2
    ldw,0   %r1, 0x8, %g16
}
{
    nop 3
    fmul,0  %g16, _f32s,_lts0 0x40800000, %g16
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```
{
  ct      %ctpr3
  stw,2   %r1, 0x8, %g16
}
.LSC.4:
{
  nop 2
  return  %ctpr3; ipd 2
  ldw,0   %r1, 0xc, %g16
}
{
  nop 3
  fmul,0  %g16, _f32s,_lts0 0x40a00000, %g16
}
{
  ct      %ctpr3
  stw,2   %r1, 0xc, %g16
}
.LSC.5:
{
  nop 2
  return  %ctpr3; ipd 2
  ldw,0   0x10, %r1, %g16
}
{
  nop 3
  fmul,0  %g16, _f32s,_lts0 0x40c00000, %g16
}
{
  ct      %ctpr3
  stw,2   0x10, %r1, %g16
}
.LSC.6:
{
  nop 2
  return  %ctpr3; ipd 2
  ldw,3   0x14, %r1, %g16
}
{
  nop 3
  fstofd,3 %g16, %g16
}
{
  nop 7
  fdivd,5  %g16, _f64,_lts0 0x401c000000000000, %g16
}
{
  nop 5
}
{
  nop 3
  fdtofs,3 %g16, %g16
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```

}
{
    ct        %ctpr3
    stw,5     0x14, %r1, %g16
}
.L72:
.LSC.7:
{
    nop 2
    return   %ctpr3; ipd 2
    ldw,0    0x18, %r1, %g16
}
{
    nop 3
    fstofd,0 %g16, %g16
}
{
    nop 3
    fadd,0   %g16, _f64,_lts0 0x3ff0000000000000, %g16
}
{
    nop 3
    fdtofs,0 %g16, %g16
}
{
    ct        %ctpr3
    stw,2     0x18, %r1, %g16
}
.T.1:
.dword     .LSC.1
.dword     .LSC.2
.dword     .LSC.3
.dword     .LSC.4
.dword     .LSC.5
.dword     .LSC.6
.dword     .LSC.7
.dword     .LSC.7
.dword     .LSC.7
.dword     .LSC.7
.dword     .LSC.1
.dword     .LSC.2
.dword     .LSC.3
.dword     .LSC.4
.dword     .LSC.5
.dword     .LSC.6

```

В ассемблере можно видеть следующее:

- различные альтернативы конструкции switch начинаются с меток *.LSC.<N>*
- метки собраны в секции данных в таблицу *.T.1*
- в начале процедуры из таблицы производится чтение по смещению:

```
ldd,3,sm    %g16, [ _f64,_lts0 .T.1 ], %r0
```

- результат чтения преобразуется в регистр подготовленного перехода с помощью команды:

```
movtd,0,sm  %r0, %ctpr1; ipd 2
```

- по регистру %ctpr1 выполняется переход:

```
ct    %ctpr1
```

Этот переход осуществит передачу управления на метку, соответствующую требуемой альтернативе конструкции switch.

Команда movtd выполняет передачу управления по косвенности: она подготавливает переход по значению регистра. Данная операция может также использоваться для вызовов функций. Продемонстрируем это на примере с вызовом виртуального метода C++.

Пример 7. Вызов виртуального метода

```

1  class base {
2  protected:
3      int val;
4
5  public:
6      virtual int getval();
7  };
8
9  int f(base *p)
10 {
11     return p->getval();
12 }
```

Листинг 11. Пример 7, оптимизация -O3

```

_Z1fP4base:
.cfi_startproc
{
    nop 2
    setwd    wsz = 0x8, nfx = 0x1
    setbn    rsz = 0x3, rbs = 0x4, rcur = 0x0
    getsp,0  _f32s,_lts1 0xffffffff0, %r2
    ldd,3    %r0, 0x0, %r3
}
{
    nop 1
    ldd,3    %r3, 0x0, %r3
}
{
    nop 1
    addd,0,sm 0x0, %r0, %b[0]
}
{
    nop 7
    movtd,0,sm    %r3, %ctpr1; ipd 2
```

(continues on next page)

(продолжение с предыдущей страницы)

```

}
nop
.LCS.1:
{
    call    %ctpr1, wbs = 0x4
}
{
    nop 5
    return %ctpr3; ipd 2
    sxt,3  0x2, %b[0], %r0
}
{
    ct     %ctpr3
}

```

В функции `f` (манглированное имя `_Z1fP4base`) в нулевом такте производится чтение с нулевым смещением по указателю `p`:

```
ldd,3 %r0, 0x0, %r3
```

По этому адресу лежит указатель на таблицу виртуальных методов класса `base`. Далее из полученного указателя на таблицу производится еще одно чтение:

```
ldd,3 %r3, 0x0, %r3
```

Полученный адрес является адресом входа в виртуальный метод `getval()`. Этот адрес записывается в регистр подготовки перехода:

```
movtd,0,sm %r3, %ctpr1; ipd 2
```

И наконец, по полученному регистру `%ctpr1` производится вызов:

```
call %ctpr1, wbs = 0x4
```

Необходимо заметить, что процесс подготовки перехода по числовому регистру является весьма длительной операцией: она требует 9 тактов. По этой причине конструкции `switch` и вызовы по косвенности являются для архитектуры Эльбрус не эффективными с точки зрения производительности. Этот вопрос дополнительно рассматривается в разделе *Рекомендации по оптимизации программ под архитектуру Эльбрус*.

2.3 Работа в gdb

Рассмотрим особенности отладки для платформы Эльбрус в gdb на примере.

2.3.1 SIGILL как сигнал об ошибках

```

1  #include <stdlib.h>
2
3  int *p;
4  int *q;
5  int g;
6
7  int main(int argc, char* argv[])
8  {
9      p=&g;
10     q=&g;
11
12     q=q+0x100000;
13
14     if (argc>1)
15         if (atoi(argv[1])==1)
16             *p = *q;
17     return 0;
18 }
```

В данном коде происходит попытка чтения по некорректному адресу, хранящемуся в q . Пример скомпилируем с опциями `-O0 -g` и `-O2 -g`.

```
gcc ./t.c -o t_00 -O0 -g
gcc ./t.c -o t_02 -O2 -g
```

При выполнении без параметров падения нет. При запуске с параметром «1» и оптимизации `-O0` увидим «Segmentation fault» после некорректного чтения из $*q$. При запуске с оптимизацией `-O2` будет выведена ошибка «Illegal instruction». Она вызвана тем же некорректным чтением, однако приложение получило сигнал SIGILL вместо SIGSEGV.

Это произошло потому, что при оптимизациях чтение из $*q$ было в спекулятивном режиме. Вместо падения исполнение программы продолжилось. При попытке записи некорректного (диагностического) значения в $*p$ происходит слом, который по системе команд вызывает сигнал SIGILL. Данный сигнал говорит о попытке работы с диагностическим значением в неспекулятивной операции, в данном случае в записи в память.

SIGILL необходимо трактовать как сигнал о программной ошибке, наряду с SIGSEGV или SIGBUS.

2.3.2 Отладка оптимизированного кода

Отображение исходного кода при исполнении программы в `gdb` возможно только при её сборке с опциями `-O0 -g`. В этом режиме работа отладчика ничем не отличается от поведения на других архитектурах.

В режиме с оптимизациями (`-O1` или выше) отображение исходного кода при исполнении не поддерживается. Чтобы пошагово выполнять программу, нужно пользоваться командами `nexti`, `stepi` вместо их аналогов `next`, `step`.

Чтобы видеть пошагово исполняемые широкие команды Эльбруса, можно выполнить:

```
(gdb) display /i $pc
```

После этого в каждой точке останова будет печататься следующая широкая команда.

Операции внутри одной широкой команды выполняются параллельно на уровне аппаратуры, поэтому в отладчике нет возможности выполнить эти операции пошагово.

Для просмотра данных выполняются команды `info registers` и печать содержимого памяти `x /x 0x<adress>`.

```
(gdb) info registers r0 r1
```

Выведет содержимое регистров `r0`, `r1`.

```
(gdb) x /4xw 0x1009a
```

Вывести в 16-ричном формате 4 одинарных слова (32-битных) из локальной памяти по адресу `'0x1009a'`

2.3.3 Пример сессии gdb

Запустим в `gdb` программу, приведённую выше в первой секции раздела об отладке.

```
gcc ./t.c -o t_02 -O2 -g
gdb ./t_02
```

Запуск программы с параметром, чтобы получить ошибку.

```
(gdb) set args 1
(gdb) r
```

```
Starting program: /home/test/t_02 1
Program received signal SIGILL, Illegal instruction
exc_diag_operand at 0x10738 ALS2
0x00000000000010738 in main ()
```

В диагностическом выводе указан адрес падения и слог в широкой команде, на котором произошёл слом.

Просмотрим дизассемблер всей функции:

```
(gdb) disassemble
```

```

Dump of assembler code for function main:
   0x0000000000010628 <+0>:
:
   ipd 2
   getsp,0 _f32s,_lts1 0xfffffe0, %dr3
   addd,1 0x0, _f64,_lts2 0x11c28, %dr4
   disp %ctpr2, M_10428
   setwd wsz = 0x8, nfx = 0x1
   setbn rsz = 0x3, rbs = 0x4, rcur = 0x0

   0x0000000000010658 <+48>:
:
   ipd 2
   cmplesb,0 %r0, 0x1, %pred0
   addd,1 0x0, _f64,_lts0 0x411c28, %dr5
   std,2 %dr4, [ _f64,_lts2 0x11c20 ]
   return %ctpr3

   0x0000000000010680 <+88>:
:
   std,2 %dr5, [ _f64,_lts0 0x11c30 ]
   ldd,5,sm [ %dr1 + 0x8 ], %dr1

   0x0000000000010698 <+112>:
:
   addd,0 0x0, 0x0, %dr0 ? %pred0
   addd,1,sm 0xa, 0x0, %db[2] ? ~ %pred0
   addd,2,sm 0x0, 0x0, %db[1] ? ~ %pred0
   addd,3 0x0, 0x0, %dr0 ? ~ %pred0
   rlp,cd00 %pred0, ~>alc2, ~>alc1, >alc0
   rlp,cd01 %pred0, ~>alc3

   0x00000000000106b0 <+136>:
:
   ct %ctpr3 ? %pred0
   ipd 3

   0x00000000000106b8 <+144>:
:
   ldd,0,sm [ _f64,_lts0 0x11c30 ], mas = 0x4, %dr1
   addd,4,sm 0x0, %dr1, %db[0] ? ~ %pred0

   rlp,cd00 %pred0, ~>alc4

   0x00000000000106d8 <+176>:
:
   ipd 3
   call %ctpr2, wbs = 0x4 ? ~ %pred0

   0x00000000000106e8 <+192>:

```

(continues on next page)

(продолжение с предыдущей страницы)

```

:
    ---Type <return> to continue, or q <return> to quit---
ipd 2
ldd,0,sm [ _f64,_lts0 0x11c20 ], %dg17
add,1,sm 0x0, %db[0], %dg16 ? ~ %pred0
return %ctpr3

rlp,cd00 %pred0, ~>alc1

    0x0000000000010708 <+224>:
:
cmpesb,0,sm %g16, 0x1, %pred1

    0x0000000000010710 <+232>:
:nop 1
pass %pred0, @p0
pass %pred1, @p1
landp ~@p0, @p1, @p4
pass @p4, %pred1

    0x0000000000010718 <+240>:
:
ldd,2 [ _f64,_lts0 0x11c30 ], mas = 0x3, %dr1 ? %pred1

rlp,cd00 %pred1, >alc2

    0x0000000000010730 <+264>:
:nop 3
ldw,0,sm [ %dr1 + 0x0 ], %g16
=> 0x0000000000010738 <+272>:
:
ct %ctpr3 ? ~ %pred0
ipd 3
stw,2 %g16, [ %dg17 + 0x0 ] ? %pred1
rlp,cd00 %pred1, >alc2

End of assembler dump.

```

В листинге команда с адресом `0x0000000000010738 <+272>` отмечена стрелочкой `=>`. Это говорит о том, что данная команда вызвала ошибку.

Чтобы подробнее разобраться, в чём она заключается, распечатаем значения регистров:

```
(gdb) info all-registers g16 g17
```

```

g16          <11> 0x4afafafa4afafafa      5402906655891061498
g17          <00> 0x11c28  72744

```

g16 содержит странное значение `0x4afafafa4afafafa`. Слева от значения регистра расположены теги диагностики, выставленные в две единицы.

Таким образом инициализируется регистр при ошибочной операции, которая произошла в спекулятивном режиме.

g17 при этом содержит корректное значение.

2.4 Прочее

2.4.1 Отладка ядра

Чтобы сбросить стек ядра, можно нажать комбинацию клавиш:

```
ctrl-prtscr-?  
ctrl-prtscr-'t'  
ctrl-prtscr-'l'  
ctrl-prtscr-'r'
```

Расширенные возможности для отладки ОС выведены в `/proc/sys/debug/`:

```
echo 1 > /proc/sys/debug/sigdebug  
echo 1 > /proc/sys/debug/datastack  
echo 1 > /proc/sys/debug/userstack  
echo 1 > /proc/sys/debug/coredump  
echo 1 > /proc/sys/debug/pagefault
```

Чтобы сохранять дампы памяти в пользовательской директории:

```
mkdir -p /export/mycore  
sysctl -w kernel.core_pattern=/export/mycore/core-%e-%s-%u-%g-%p-%t
```

2.4.2 Модификация запуска задач

В случае, если требуется привязать выполнение задачи к какому-то конкретному ядру или набору ядер, следует использовать команду `taskset`. Пример:

```
taskset -c 1,2,3,4 <команда с аргументами>
```

Глава описывает любые отличия в поведении системных компонент на Эльбрусе, из-за которых может понадобиться адаптация программ для их корректной работы.

3.1 Совместимость с компиляторами

Компилятор `lcc` стремится к максимальной совместимости с `gcc`. Версия `lcc-1.24` соответствует `gcc-7.3.0`.

3.1.1 Конструкции языка

3.1.1.1 Variable length array inside a struct

Не поддерживается недокументированная функция `gcc`: Variable length arrays (VLA) в середине структуры.

3.1.1.2 Nested functions

Не поддерживается элемент диалекта GNU C: nested functions.

3.1.2 gcc builtins

Ряд билтинов (builtin) `gcc` поддерживается с ограничениями. Некоторые из них вызваны особенностями реализации компилятора, другие отсутствием практической необходимости. С выходом новых версий `lcc` расширяется состав поддерживаемых билтинов.

Ограничения перечислены в разделе документации на компилятор:

`/opt/mcst/doc/builtin_gnu.html`.

3.1.3 Прагмы

Информация о поддерживаемых прагмах в документации компилятора:

`/opt/mcst/doc/pragma.html`

При сборке ПО иногда встречается особенность с неподдерживаемыми выражениями в прагмах. Для архитектуры Эльбрус следует использовать только константы и переменные.

Вместо

```
#pragma omp parallel for if(!singleThreaded) schedule(dynamic)
```

нужно написать

```
bool multiThreaded = !singleThreaded;
```

```
#pragma omp parallel for if(multiThreaded) schedule(dynamic)
```

3.1.4 OpenMP

3.1.4.1 Возможности

- Поддержан стандарт OpenMP 3.1.
- Доступны языки C, C++, Fortran.

3.1.4.2 Ограничения

- Для e2k не поддерживается в режиме `-m128`.
- Nested параллелизм не поддерживается. Если при выполнении уже распараллеленного цикла встречаются циклы, которые нужно распараллелить, то эти (вложенные) циклы будут исполняться последовательно.
- Не поддерживается `clause collapse`.
- Для C/C++ после директивы `#pragma omp` всегда должен следовать `statement` языка. Проблемы могут возникнуть для `#pragma omp barrier` и `#pragma omp flush`, если за ними нет `statement`'а. Для обхода проблемы рекомендуется в следующей строке поставить пустой `statement`, например `“0;”` или `“;”`
- Переменные, перечисленные в `clause`'ax `private`, `lastprivate`, `firstprivate` и `threadprivate` должны иметь скалярный базовый тип или массив скалярного базового типа. В противном случае результат программы неопределен.
- Директива `#pragma omp for` не поддерживается для итераторов C++.
- Для C/C++ `clause`'ы `if` и `num_threads` своими параметрами могут иметь только константы и переменные целого типа, выражения не допускаются.

3.1.4.3 Справочный файл

В компиляторе информация об OpenMP хранится здесь:

`/opt/mcst/doc/openmp.html`

3.2 Системные интерфейсы

3.2.1 `makecontext`

Функция `makecontext()` для управления контекстом пользователя реализована на Эльбрусе с другой семантикой.

Вместо вызова `makecontext()` необходимо вызывать `makecontext_e2k()`. Функции дано другое название, чтобы отображать несовместимость с реализацией на других архитектурах.

`makecontext_e2k()` - в отличие от `makecontext()`, дополнительно выделяет аппаратные стеки, из-за чего является более тяжеловесным.

Можно вызвать `makecontext_e2k()` дважды на один и тот же контекст, не выполняя `freecontext_e2k()`. Для ядер ≥ 5.4 контекст будет переиспользован напрямую, что гораздо быстрее, чем освобождать его через `freecontext_e2k()` и выделять новый. Перед этим следует убедиться, что контекст не используется. При повторном использовании `makecontext` можно подменять стек, т.е. с точки зрения пользователя вызов «`makecontext + makecontext`» полностью эквивалентен вызову «`makecontext + freecontext + makecontext`», за исключением производительности.

`makecontext_e2k()` возвращает значение `int`, а не `void`. Значение вызова необходимо проверять на статус ошибки (< 0).

Возможные ошибки:

- EFAULT - не удалось обратиться по указателю `ucp` или `ucp->uc_stack`;
- ENOMEM - в системе недостаточно памяти;
- EINVAL - подозрительное содержимое `ucontext_t`;
- EBUSY - `ucp->uc_stack` уже отдан под другой контекст.

3.2.2 `freecontext`

При использовании `makecontext_e2k()` в конце области видимости необходимо вызвать `freecontext_e2k()`.

`freecontext_e2k()` - освобождает связанные с контекстом аппаратные стеки. Если контекст в данный момент занят, то реальное освобождение произойдет, только когда он перестанет использоваться. Попытка освободить текущий контекст (используемый текущим потоком) игнорируется.

3.2.3 swapcontext/setcontext

swapcontext()/setcontext() - при ошибке возвращают не только ENOMEM.

Возможные ошибки:

- EFAULT - подан плохой указатель или в системе недостаточно памяти;
- ENOMEM - в системе недостаточно памяти;
- ESRCH - некорректный usr;
- EBUSY - контекст usr уже используется.

3.2.3.1 Особенность переключения контекстов

Для архитектуры Эльбрус, помимо доступного пользовательскому приложению контекста (%r, %b], содержимое стека данных), существует также и привилегированный контекст, доступный на запись лишь ядру ОС, например:

- указатели на стеки (%pcsp, %psp, %usd, %sbr);
- содержимое стека связующей информации (он же «стек возвратов»);
- при исполнении в обработчике сигнала: указатель на стек из прерванных сигналами контекстов (%tir, trap cellar, %ctpr, %g16-%g31, ...)

Это означает, что каждый сохранённый пользовательский контекст состоит из двух половин: ровно одной привилегированной, хранящейся в ядре ОС; и ≥ 1 непривилегированной, хранящейся в памяти приложения (struct ucontext).

Ввиду этого, во-первых, makecontext помимо struct ucontext дополнительно инициализирует привилегированную половинку нового контекста (в основном это два стека - связующей и процедурный), и во избежание утечки памяти при освобождении struct ucontext нужно дополнительно освободить привилегированную половинку вызовом freecontext_e2k(). Во-вторых, для связи этих двух половинок заводится хэш-таблица, и swapcontext должен найти два элемента этой таблицы.

В итоге: примерно равный вклад в замедление вносят: организация хэш-таблицы; необходимость полноценного системного вызова; большой размер контекста.

3.2.4 sys_get_backtrace/sys_set_backtrace

Функции sys_get_backtrace() и sys_set_backtrace() используются для чтения и модификации адресов возврата в стеке связующей информации. sys_get_backtrace() считывает адреса в буфер, sys_set_backtrace() - записывает.

Если при записи в буфер положить специальное значение -1UL, то адрес возврата будет подменен на пустую функцию, просто выполняющую return, это полезно для пропуска функций, у которых нет обработки исключений.

```
long sys_get_backtrace(unsigned long *buf, size_t count,
size_t skip, unsigned long flags);

long sys_set_backtrace(const unsigned long *buf, size_t count,
size_t skip, unsigned long flags);
```

Здесь:

count - размер буфера;

skip - сколько пропустить фреймов в стеке связующей;

flags - должен быть равен 0.

При успехе возвращают число прочитанных/записанных адресов (даже если это число меньше чем count). При неудаче возвращают код ошибки.

В случае, если удалось записать не все адреса, код ошибки можно получить, повторив вызов sys_set_backtrace со skip увеличенным и count уменьшенным на число успешно записанных адресов.

sys_get_backtrace/sys_set_backtrace возвращают ошибку:

- EFAULT В случае если буфер недоступен;
- EINVAL В случае нулевого flags.

sys_set_backtrace возвращает ошибку:

- ESRCH Если один из IP указывает на незамапированную область.
- **EPERM В случае если сработала проверка:**

- 1) Либо старый и новый IP имеют различные разрешения r/w/x;
- 2) Либо старый и новый IP соответствуют различным исполняемым файлам.

Если требуется просто заменить ip возврата текущий функции, то sys_set_backtrace() может оказаться слишком тяжеловесным, тогда можно воспользоваться fast_sys_set_return() :

```
#if __ptr32__
# define FAST_SYS_ENTRY "5"
#elif __ptr64__
# define FAST_SYS_ENTRY "6"
#else
# error
#endif

#include <asm/unistd.h>

unsigned long ip = <return ip>;
int ret;

asm ("{"sdisp %%ctpr1, " FAST_SYS_ENTRY "}\n" \
     "{adds 0, %1, %%b[0]\n" \
     "add 0, %2, %%db[1]\n" \
     "adds 0, 0, %%b[2]\n" \
     "call %%ctpr1, wbs=%#\n" \
     "adds 0, %%b[0], %0\n" \
     : "=r" (ret)
     : "i" (__NR_fast_sys_set_return), "r" (ip)
     : "ctpr1", "ctpr2", "ctpr3", "memory",
     "b[0]", "b[1]", "b[2]", "b[3]", "b[4]", "b[5]", "b[6]", "b[7]",
     "g16", "g17", "g18", "g19", "g20", "g21", "g22", "g23",
     "g24", "g25", "g26", "g27", "g28", "g29", "g30", "g31");
```

3.2.5 `sys_access_hw_stacks`

Функция `sys_access_hw_stacks()` используется для чтения и записи аппаратных стеков текущего потока в памяти, предполагается такое использование:

- получаем размер через `E2K_GET_*_STACK_SIZE`
- выделяем буфер при необходимости
- читаем в буфер сколько нужно/откуда нужно через `E2K_READ_*_STACK_EX`
- правим в буфере
- пишем обратно только изменённую часть буфера через `E2K_WRITE_*_STACK_EX`

Для подключения нужных определений следует использовать:

```
#include <asm/e2k_syswork.h>
```

1) `E2K_GET_CHAIN_STACK_SIZE` и `E2K_GET_PROCEDURE_STACK_SIZE` используются для получения размера аппаратных стеков.

```
__u64 chain_size, procedure_size;

sys_access_hw_stacks(E2K_GET_CHAIN_STACK_SIZE, NULL, NULL, 0,
&chain_size)
sys_access_hw_stacks(E2K_GET_PROCEDURE_STACK_SIZE, NULL, NULL, 0,
&procedure_size)
```

2) Для чтения/записи аппаратных стеков по смещениям относительно их реальной базы (реальная база может быть меньше прописанного в регистре значения `%pcsp.base`). Предполагается выравнивание `offset` и `size` на 8 для процедурного стека и на 32 для связующей (в связующую пишем только фреймы целиком).

```
__u64 offset;
void *buffer;
unsigned long size;

sys_access_hw_stacks(mode, &offset, buffer, size, NULL)
```

Здесь:

`mode` - один из `E2K_READ_CHAIN_STACK_EX`, `E2K_READ_PROCEDURE_STACK_EX`, `E2K_WRITE_PROCEDURE_STACK_EX`, `E2K_WRITE_CHAIN_STACK_EX`.

`offset` - смещение в стеке, откуда читать или куда писать

`buffer` - буфер в пользователе, в который/из которого копировать

`size` - сколько копировать

При попытке считать/записать больше, чем есть в стеке, функция вернёт ошибку - `EINVAL`.

Введение в архитектуру «Эльбрус»

4.1 Введение

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

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 операции 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, тщательно планируя отдельные операции, исходя из ресурсов широкой команды. В результате аппаратра процессора может быть проще и экономичнее.

Компилятор способен анализировать исходный код гораздо тщательнее, чем аппаратра RISC/CISC процессора, и находить независимые операции в существенно большем диапазоне. Поэтому в архитектуре Эльбрус больше параллельно работающих исполнительных устройств, чем в традиционных архитектурах.

4.2 Характеристики микропроцессоров «Эльбрус»

Ниже представлены характеристики актуальных моделей микропроцессоров «Эльбрус».

Таблица 1: 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

Таблица 2: 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

Таблица 3: Характеристики микропроцессора «Эльбрус-8СВ»

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

4.3 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*.

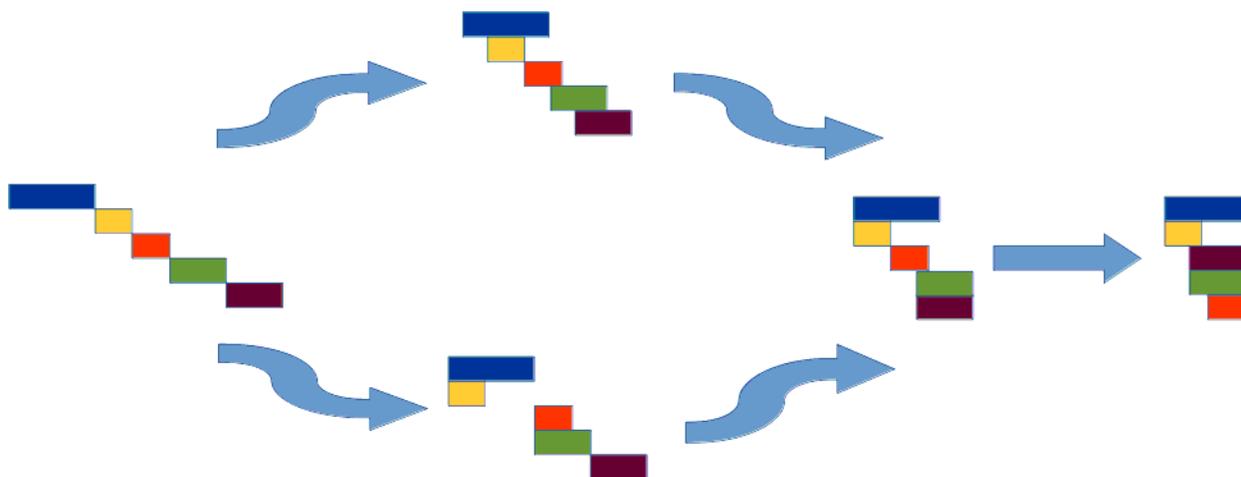


Рис. 1: Переход от скалярных процессоров к суперскалярным с возможностью перестановки инструкций

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».

4.4 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 (см. рис. *Широкая команда «Эльбрус», парк 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).

Int, FP, Vect, LD, Cmp		Int, FP, Vect, LD, Cmp			
Int, FP, Vect, Cmp		Int, FP, Vect, Cmp			
Int, LD, ST, FP*		Int, LD, ST, Div/Sqrt, FP*			
CT					
PL		PL		PL	
QP	QP	QP	QP	QP	QP
APB		APB		APB	
LIT32		LIT32		LIT32	

Рис. 2: Широкая команда «Эльбрус», парк устройств

* В каналах 2 и 5 операции над вещественными числами поддерживаются 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.

4.5 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-конвейеризованном 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.

4.6 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. $e=y*y$

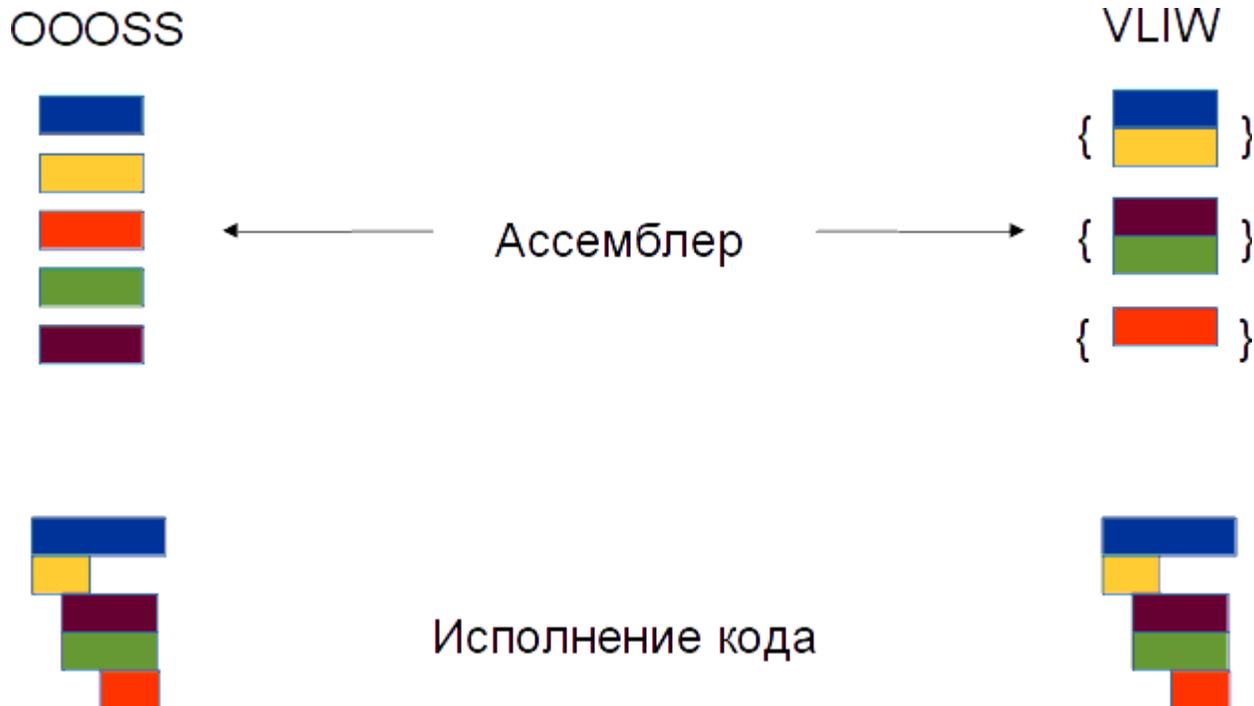


Рис. 3: Сравнение исполнения кода в OOOSS и VLIW

На рисунке *Сравнение исполнения кода в OOOSS и VLIW* 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 аппаратра 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 выполнении 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.*

Анализ производительности программ

В этом разделе рассматриваются инструменты для анализа производительности кода на платформе «Эльбрус».

Важнейшие утилиты для работы с производительностью - *профилировщики*. Под *профилем* понимается информация о времени и частоте исполнения различных участков программы. Возможна различная детализация профиля:

- с точностью до процедуры;
- с точностью до линейных участков;
- с точностью до отдельных команд;
- с точностью до состояния конвейера при исполнении отдельных команд.

Профиль можно получать с помощью:

- симулятора / модели процессора;
- инструментующих профилировщиков;
- сэмпинговых профилировщиков;

Рассмотрим две утилиты для сбора сэмплов в ОС «Эльбрус Линукс»:

- **dprof** (разработан в АО «МЦСТ»);
- **perf** (поддержана утилита и подсистема в ядре ОС).

Сэмпл - информация о состоянии программы в момент регулярно-случайной остановки. Минимальная информация в сэмпле - IP адрес текущей команды.

Профилировщик **dprof** запускает программу в качестве *ptrace*-наблюдаемой и перехватывает прерывания для сбора/сброса сэмпла. При этом каждое прерывание приводит к смене контекста «программа»/**dprof** для того, чтобы **dprof** сбросил сэмпл. Профилировщик **perf** имеет поддержку в ядре ОС, и смены контекста в этом случае не происходит - сбор и сброс сэмпла выполняются при обработке прерывания в ядре ОС.

Рассмотрим примеры использования профилировщиков. В обзоре будут представлены не все возможности, а лишь активно используемые на платформе Эльбрус для сбора информации о потреблении ресурсов.

5.1 Утилита perf

Для записи сэмплов надо использовать команду:

```
perf record
```

Следом за ней указывается тип события, по которому будет происходить сбор. Оптимальным для платформы Эльбрус является тип *task-clock*. Потом через параметр *-F* указывается необходимая частота срабатывания: сколько раз в секунду программа останавливается для сбора информации о выполняемой функции. Рекомендации по выбору числа для *-F* могут быть следующими:

- при времени выполнения программы несколько минут и более: *-F10 -F100*
- при выполнении программы менее одной минуты: *-F1000*
- при выполнении программы менее одной секунды: *-F10000*.

Так можно собрать сэмплы для программы а.ОЗ:

```
perf record -e task-clock -F10000 ./a.O3
```

Выдача:

```
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.340 MB perf.data (8818 samples)
```

Стоит следить за тем, чтобы количество собранных сэмплов не было слишком маленьким.

Чтобы посмотреть собранный профиль, необходимо выполнить команду:

```
perf report
```

Профиль имеет вид:

```
Samples: 8K of event 'task-clock', Event count (approx.): 881800000
Overhead      Samples Command Shared Object      Symbol
99,65%        8787 a.O3      a.O3              [.] sample
0,09%          8 a.O3      a.O3              [.] main
0,02%          2 a.O3      [kernel.kallsyms] [k] page_add_file_rmap
0,02%          2 a.O3      [kernel.kallsyms] [k] release_pages
0,02%          2 a.O3      [kernel.kallsyms] [k] zap_pte_range
0,01%          1 a.O3      [kernel.kallsyms] [k] __flush_icache_pag
```

Два раза нажимая *Enter* на имени конкретной функции, мы можем посмотреть дизассемблер. Для возврата к профилю нужно нажать “q”. В данном примере видно, что дольше всех работает функция *sample*, соответственно ее и стоит анализировать для повышения производительности.

5.2 Утилита dprof

Рассмотрим профилировщик dprof.

```
dprof -o IP_raw ${exe} params...
```

В файле *IP_raw* будет находиться список адресов Instruction Pointer-ов (IP), полученный с интервалами 10ms. Частоту можно менять опцией *-d*.

```
[14684] 0000455555786910
[14684] 000000000036d38
[14684] 000000000007c968
```

По опции *-b* можно сбрасывать не только IP, но и весь стек адресов возврата. Это позволит в дальнейшем получить профиль с учетом графа вызовов.

```
[14716] 000000000011400
[14716] DPROF-BACKTRACE EVENT timer:10 (depth=12): 000000000011400 000000000063898
↪000000000038d00 000000000039258 00000000003b778 00000000002faa8 000000000030f80
↪0000000000071298 0000000000070100 000045555560e758
[14716] 0000000000059408
[14716] DPROF-BACKTRACE EVENT timer:10 (depth=9): 0000000000059408 00000000003b818
↪00000000002faa8 000000000030f80 0000000000071298 0000000000070100 000045555560e758
```

Для получения текстового профиля из *IP_raw* можно использовать команды:

```
dprof2ptrace.sh IP_raw >${exe}.ptrace
ldis -P ${exe}
```

Пример вывода от команд выше:

```
.symtab:
%time      %summ      time      calls     time/call  name
25.81%     25.81%     24        1         24         mrglist
12.90%     38.71%     12        1         12         addlist
6.45%      45.16%     6         1         6          getefflibs
3.23%      48.39%     3         1         3          lupdate
```

Для визуализации профиля с использованием информации о графе вызовов можно использовать формат *callgrind/cachegrind*:

```
dprof2callgrind.sh ${exe} IP_raw >callgrind.out
```

С помощью сэмпинговых профилировщиков можно получать доступ к информации о блокировках и их причинах.

Во всех современных микропроцессорах встроены отлаочно-диагностические мониторные регистры для профилирования различных событий. Можно использовать эти мониторные регистры для выработки прерываний, которые перехватывает сэмпинговый профилировщик.

Примеры:

- TICKS – все такты
- EXEC – исполненные команды
- BUB_V – блокировки конвейера на стадии В

- BUB_E2 – блокировки конвейера на стадии E2
- IB_NO_COMMAND – блокировки конвейера на стадии подкачки кода
- L2_QUERY – запросы в L2\$
- L2_HIT – попадания в L2\$

Для сбора профиля по событиям нужно задать событие и частоту выработки прерываний SAV (Sample After Value):

```
dprof -o IP_raw_exec -e EXEC:10000000 ${exe} params...
```

Прерывания будут вырабатываться аппаратурой через каждые 10^7 выполненных команд.

Можно задать до 4 событий одновременно, каждое из них будет вырабатывать прерывания с собственным отличительным маркером.

Рассмотрим пример работы программы `bc`, которая вычисляет математическое выражение.

```
hostname /export/minis/Dprof # time -p echo 2^218839%2 | bc
```

```
0
real 1.00
user 1.00
sys 0.00
```

Для удобства числа подобраны так, чтобы время выполнения равнялось одной секунде. Так как машина, на которой выполняется выражение, работает с тактовой частотой 1200MHz, это значит, что за одну секунду будет выполнено миллиард двести миллионов тактов, часть из которых пойдет на полезные вычисления, часть на задержки.

Рассмотрим профиль выполнения примера:

```
echo 2^218839%2 | dprof -o IP_raw -b -f /usr/bin/bc
```

```
-- child 9530 attached
0
-- child 9530 exited with code
```

```
dprof2ptrace.sh IP_raw >bc.ptrace
cp /usr/bin/bc .
ldis -P ./bc
./bc:
```

```
.symtab:
%time    %summ    time     calls    time/call  name
46.00%   46.00%   46        1         46        _bc_simp_mul
37.00%   83.00%   37        1         37        _bc_shift_addsub
 7.00%   90.00%   7         1          7        (*) external unknown
 4.00%   94.00%   4         1          4        _bc_do_sub
 2.00%   96.00%   2         1          2        _bc_rec_mul
 1.00%   97.00%   1         1          1        bc_new_num
 1.00%   98.00%   1         1          1        bc_free_num
 1.00%   99.00%   1         1          1        free@plt
 1.00%  100.00%   1         1          1        memset@plt
```

Если мы сравним этот профиль с полученным командой perf:

45,73%	4749	bc	bc	[.]	_bc_shift_addsub
42,43%	4407	bc	bc	[.]	_bc_simp_mul
3,76%	390	bc	bc	[.]	_bc_do_sub
1,65%	171	bc	bc	[.]	_bc_rec_mul
1,45%	151	bc	libc-2.29.so	[.]	cfree@GLIBC_2.2
1,15%	119	bc	bc	[.]	bc_free_num
0,87%	90	bc	libc-2.29.so	[.]	malloc
0,51%	53	bc	libc-2.29.so	[.]	memset

то видим, что профили различаются, но не существенно. Две самые тяжелые функции в обоих профилировщиках одни и те же.

Посмотрим, сколько тактов ушло на различные события при выполнении примера. Узнаем общее количество потраченных тактов:

```
echo 2^218839%2 | /opt/mcst/bin/dprof -m TICKS /usr/bin/bc
```

```
-- child 9566 attached
0
-- child 9566 exited with code 0
[9566] event TICKS (2): 1195620487
```

Рассмотрим, какую долю этих тактов выполнялся код:

```
echo 2^218839%2 | /opt/mcst/bin/dprof -m EXEC /usr/bin/bc
```

```
-- child 9569 attached
0
-- child 9569 exited with code 0
[9569] event EXEC (2): 82864377
```

Т.е. $1195620487 - 828643773 = 366976714$ тактов потрачено на обслуживание вычислений – отсутствие кода и другие задержки.

Рассмотрим другие счётчики. На отсутствие команд пришлось порядка 5 миллионов тактов.

```
echo 2^218839%2 | /opt/mcst/bin/dprof -m IB_NO_COMMAND /usr/bin/bc
```

```
-- child 9582 attached
0
-- child 9582 exited with code 0
[9582] event IB_NO_COMMAND (2): 5252756
```

На задержки от BUB_E2 ушло порядка 55 миллионов тактов:

```
echo 2^218839%2 | /opt/mcst/bin/dprof -m BUB_E2 /usr/bin/bc
```

```
-- child 9588 attached
0
-- child 9588 exited with code 0
[9588] event BUB_E2 (2): 55073991
```

Таким образом можно просмотреть большое количество событий и проанализировать, что больше всего может повлиять на производительность. Полный список доступных событий для анализа можно посмотреть, выполнив команду:

```
dprof -mlist
```

5.3 Просмотр привязки к исходнику

Тестовый пример умножения векторов:

```
//mul_vector_eml.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <eml/eml.h>

#define N 100000
#define M 1000000

int main()
{
    int i,j;
    double s=0.0;
    double *A;
    double *B;
    double *C;
    A = (double*)malloc(N * sizeof(double));
    B = (double*)malloc(N * sizeof(double));
    C = (double*)malloc(N * sizeof(double));

    srand(time(NULL));
    for (i = 0; i < N; i++)
    {
        A[i] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
        B[i] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
    }

    for (j=0; j<M; j++)
        eml_Vector_Mul_64F(A,B,C,N);

    s+=C[0];
    free(A);
    free(B);
    free(C);

    return (int)s;
}
```

Чтобы посмотреть привязку в ассемблерном файле, нужно использовать дополнительную опцию -fverbose-asm:

```
$ lcc mul_vector_eml.c -fverbose-asm -S -O3
$ cat mul_vector_eml.s
```

```
.file "mul_vector_eml.c"
.ignore ld_st_style
.ignore strict_delay
! -----
.text
! t.c : 11
.global main
.type main, #function
.align 8
main:
! <0000>
{
setwd wsz = 0xc, nfx = 0x1, dbl = 0x0 ! t.c : 11
setbn rsz = 0x3, rbs = 0x8, rcur = 0x0 ! t.c : 11
disp %ctpr2, malloc ! t.c : 17
getsp,0 _f32s,_lts1 0xffffffffe0, %dr2 ! t.c : 11
add,1 0x0, _f32s,_lts2 0xc3500, %dr11 ! t.c : 17
adds,2 0x0, _f32s,_lts3 0x68db8bad, %r9 ! t.c : 24
add,3 0x1f, 0x1, %dr7 ! t.c : 24
adds,4 0x1f, 0x0, %r8 ! t.c : 24
add,5,sm 0x0, 0x0, %dr0 ! t.c : 22
}
! <0001>
{
nop 3
add,0,sm 0x0, %dr11, %db[0] ! t.c : 17
add,1 0x0, _f64,_lts0 0x40c3880000000000, %dr10 ! t.c : 24
adds,2 0x0, 0x0, %r6 ! t.c : 22
}
...

```

Перед каждой функцией будет помещена строка с привязкой к исходнику, а затем на каждой команде, где известна привязка к исходнику, будет напечатано имя файла и номер строки. Чтобы удобнее было искать соответствие между ассемблером и дизассемблером, для архитектуры e2k дополнительно в угловых скобках будет напечатан номер такта.

Чтобы посмотреть привязку в кодовом файле, нужно использовать дополнительную опцию `-gline`:

```
$ lcc -O3 mul_vector_eml.c -gline -c
$ ldis mul_vector_eml.o
```

```
! function 'main', entry = 9, value = 0x000000, size = 0x420, sect = ELF_TEXT num = 1
0000<00000000000000> main:
getsp,0 _f32s,_lts1 0xffffffffe0, %dr2 ! t.c : 11
add,1 0x0, _f32s,_lts2 0xc3500, %dr11 ! t.c : 17
adds,2 0x0, _f32s,_lts3 0x68db8bad, %r9 ! t.c : 24
add,3 0x1f, 0x1, %dr7 ! t.c : 24
adds,4 0x1f, 0x0, %r8 ! t.c : 24
add,5,sm 0x0, 0x0, %dr0 ! t.c : 22

```

(continues on next page)

(продолжение с предыдущей страницы)

```
disp %ctpr2, $malloc ! t.c : 17
setwd wsz = 0xc, nfx = 0x1, dbl = 0x0
setbn rsz = 0x3, rbs = 0x8, rcur = 0x0

0001<0000000000038> :nop 3
add,0,sm 0x0, %dr11, %db[0] ! t.c : 17
add,1 0x0, _f64,_lts0 0x40c3880000000000, %dr10 ! t.c : 24
adds,2 0x0, 0x0, %r6 ! t.c : 22
...
```

В каждой команде, для которой имеется соответствие, будет напечатано имя файла и номер строки. Таким простым способом удобно просматривать, какая часть кода отвечает за конкретные строки программы.

В компиляторе информация про привязки к исходникам находится здесь:

`/opt/mcst/doc/code_to_source.html`

Использование библиотеки `eml` подробно описано в разделе *Использование оптимизированных библиотек*

Повышение производительности

Один из известных ресурсов повышения производительности - выявление и использование параллелизма на уровне элементарных операций (устоявшийся термин ILP = Instruction-Level Parallelism). Для АПП «Эльбрус» параллелизм выражается в машинном коде явным образом в виде Широких Команд (ШК), а задача обнаружения и эффективного использования ILP возлагается на компилятор - этот принцип характерен для всех известных процессоров VLIW. Далее будет показан ряд решений, реализованных в АПП Эльбрус, направленных на выражение, обнаружение и увеличение ILP. Ознакомившись с материалом, читатель сможет видеть в коде результаты применения этих решений и регулировать их работу с помощью опций оптимизирующего компилятора и/или модификации исходных текстов оптимизируемых программ.

6.1 Планирование кода

Рассмотрим простейший пример функции на языке C и соответствующий ей ассемблерный код:

```
int f(int a, int b, int c)
{
    int x = a * b + c<<3;
    int y = b * c + a<<4;

    return x + y;
}
```

```
f:
    muls    %r0, %r1, %r7    !0    ! _t0 = a * b
    shls    %r2, 0x3, %r8    !1    ! _t1 = c<<3
    adds    %r7, %r8, %r5    !2    ! x = _t0 + _t1
    muls    %r1, %r2, %r9    !3    ! _t2 = b * c
    shls    %r0, 0x4, %r10   !4    ! _t3 = a<<4
    adds    %r9, %r10, %r6   !5    ! y = _t2 + _t3
    adds    %r5, %r6, %r0    !6    ! _retval = x + y
```

(continues on next page)

(продолжение с предыдущей страницы)

return	%ctpr3	!7	! prepare return
ct	%ctpr3	!8	! return

В примере для простоты оставлены только арифметические операции и операции передачи управления.

Если попробовать собрать из приведенного ассемблера машинный код и исполнить его на потактовом симуляторе процессора, то можно наблюдать следующее:

- все операции заняли по одной широкой команде, и как следствие по одному такту, несмотря на наличие независимых вычислений, которые можно было бы запустить одновременно, пользуясь ресурсами широкой команды;
- некоторые широкие команды выполняются с задержками в несколько тактов; это вызвано тем, что некоторые операции требуют нескольких тактов для выработки результата, и конвейер процессорного ядра, пытаясь исполнить команду с неподготовленным аргументом, пропустит несколько тактов.

```
f:
    muls    %r0, %r1, %r7    !0    ! _t0 = a * b
    shls    %r2, 0x3, %r8    !9    ! _t1 = c<<3
+4 такта
    adds    %r7, %r8, %r5    !10   ! x = _t0 + _t1
    muls    %r1, %r2, %r9    !11   ! _t2 = b * c
    shls    %r0, 0x4, %r10   !20   ! _t3 = a<<4
+4 такта
    adds    %r9, %r10, %r6   !21   ! y = _t2 + _t3
    adds    %r5, %r6, %r0    !22   ! _retval = x + y
    return  %ctpr3          !23   ! prepare return
+6 тактов
    ct      %ctpr3          !30   ! return
```

Как видим, оба фактора отрицательно сказываются на производительности при исполнении этого кода. Перед компонентой компилятора, занимающейся планированием кода, стоит задача обнаружить и выразить в широких командах параллельность на уровне операций в пределах линейных участков кода, и одновременно с этим при размещении очередной операции в широкой команде выдержать длительности операций-предшественников, которые вырабатывают ее аргументы.

```
f:
{ ! T=0
    return    %ctpr3          !0    ! prepare return
    muls,0    %r0, %r1, %r7    !0    ! _t0 = a * b
    muls,1    %r1, %r2, %r9    !0    ! _t2 = b * c
}
{ ! T=1
    nop 4
    shls,0    %r2, 0x3, %r8    !1    ! явная задержка
    shls,1    %r0, 0x4, %r10   !1    ! _t1 = c<<3
    shls,1    %r0, 0x4, %r10   !1    ! _t3 = a<<4
}
{ ! T=6
    adds,0    %r7, %r8, %r5    !6    ! x = _t0 + _t1
    adds,1    %r9, %r10, %r6   !6    ! y = _t2 + _t3
}
{! T=7
    adds,0    %r5, %r6, %r0    !7    ! _retval = x + y
```

(continues on next page)

(продолжение с предыдущей страницы)

```

ct          %ctpr3          !7    ! return
}
    
```

Ниже приведена таблица длительностей некоторых классов операций на платформе «Эльбрус».

Операция	Задержка
Int, Bitwise	1
Int_combined	2
Fp	4
Fp_combined	8
Fdivs/d	11/14
Ld L1 hit	3
Ld L2 hit	11
Ld L3 hit	40
Ld mem	~100
Cmp->logic	1
logic->logic	0.5
Cmp,logic->qual	2
Cmp,logic->ct	3
Disp->ct	5
Return->ct	6
Movtd->ct	9
fp->int	+2
int->fp	+0

Int, bitwise целочисленные операции **add**, **sub**, побитовые **and**, **or**, **xor**, сдвиги **shl**, **shr**, **sar**, расширение знака **sxt** и некоторые другие. Время выполнения - 1 такт: если операция стоит в такте N, ее результат готов к использованию в такте N+1.

Int_combined комбинированные (двухэтажные) целочисленные операции: **shl_add**, **xor_and** и т.д.

Fp вещественные операции сложения и умножения **fadd**, **fmul**, а также целочисленные операции **mul**.

Fp_combined комбинированные вещественные операции **fmul_fadd**, **fadd_sub** и т.д.

Fdiv операция вещественного деления. Длительность операции зависит от используемой разрядности (для 32 бит - это 11 тактов, для 64 бит - это 14 тактов).

Операция чтения из памяти имеет различную длительность в зависимости от уровня кэш-памяти, в которой нашлась копия читаемых данных. Самое быстрое чтение из L1 - 3 такта, самое долгое чтение - из памяти, оно может длиться от 100 до 200 тактов в зависимости от конкретной архитектуры процессора «Эльбрус». Планировщик всегда будет планировать чтение из L1, если только от программиста не было подсказки не обращаться по чтению в L1.

Для операций сравнения **Cmp** и логических операций **andp** длительность определяется тем, какая операция и каким образом потребляет результат.

- для потребителей вида **andp** задержка составляет 1 такт, причем в некоторых случаях источник **andp** и потребитель **andp** можно спланировать в одном такте (но не более двух зависимых), поэтому длина задержки **andp - andp** трактуется как 0.5 такта.
- для потребителей - арифметических операций, исполнение которых управляется (отменяется) вычисленным предикатом, задержка равна 2 тактам;
- для потребителей - операций передачи управления **ct**, **call**, **branch**, задержка равна 3 тактам.

Для различных операций, устанавливающих регистр подготовки передачи управления, величина задержки до операции передачи управления различается:

- для статической подготовки перехода или вызова процедуры - 5 тактов;
- для статической подготовки возврата из процедуры - 6 тактов;
- для динамической подготовки перехода или вызова процедуры по значению числового регистра (вызовы по указателю, конструкции switch-case) - 9 тактов.

Наконец, есть дополнительный «штраф» за передачу данных между целочисленными и вещественными операциями - он составляет +2 такта к длительности операции в случае, когда операция-источник является вещественной, а потребитель - целочисленной, и +1 такт в симметричном случае: источник - целочисленная операция, потребитель - вещественная.

Нужно отметить, что дополнительный «штраф» в классе fp получают упакованные (векторные) целочисленные операции, а также операции целочисленного умножения и деления.

Таким образом, возвращаясь к рассмотренному примеру, из приведенной краткой таблицы можно увидеть, что задержка от операции mul до операции add составляет 6 тактов: 4 такта собственной длительности mul, и +2 такта передачи результата от вещественной операции до целочисленной.

6.2 Inline - подстановки

Под инлайном функции понимается подстановка тела функции в другую функцию. При этом происходит связь фактических параметров с подставляемыми. В результате такой подстановки исчезают затраты на время вызова процедуры, передачи параметров и возврата значения, а также возникает контекст для дополнительных оптимизаций. Иллюстрация inline-подстановки представлена на примере.

```
double f(double x, double y)
{
    double r;

    if ( x > y )
        r = ( x - y );
    else
        r = ( x + y );

    return r;
}

void g(double *p, int n)
{
    int i;

    for (i=0; i<n; i++)
        p[i]=f(p[i],p[i+1]);

    return;
}
```

После inline-подстановки функция **f** осталась бы прежней. Она не исчезает, так как она может быть вызвана и из других функций. А функция **g** приобрела бы следующий вид:

```
void g(double *p, int n)
{
```

(continues on next page)

(продолжение с предыдущей страницы)

```

int i;

for (i=0; i<n; i++)
    if (p[i] > p[i+1])
        p[i] = ( p[i] - p[i+1] );
    else
        p[i] = ( p[i] + p[i+1] );

return;
}

```

Можно выделить как позитивный, так и негативный эффект от inline-подстановки.

Позитив:

- удаление операций вызова, возврата, передачи параметров;
- в некоторых случаях контекст для других оптимизаций: упрощение вычислений, протаскивание констант, упрощение управления;
- увеличение степени параллельности операций.

Негатив:

- усложнение кода для отладки;
- увеличение времени компиляции;
- снижение hit rate для кэша команд.

Вызов функции для компилятора является барьером для оптимизации: компилятор в поиске ИЛР ограничен процедурой, и к тому же часто не имеет возможности переставлять операции с вызовами функций. Инлайн вызовов существенно помогает в поиске ИЛР. Однако для успешных inline-подстановок требуется выполнение ряда условий.

Первое — вызываемая процедура должна находиться в том же самом модуле, где и вызывающая ее процедура. Для решения этой проблемы предусмотрена возможность сборки проекта в режиме межмодульной оптимизации. Для этого на вход компилятору следует подать опцию *-fwhole*.

Второй проблемой является вызов по косвенности. Компилятор может бороться с косвенностью:

- либо с помощью межмодульного анализа, определяющего множество функций, которые могут быть вызваны в каждой точке вызова по косвенности; этот анализ также доступен по опции *-fwhole*;
- либо с помощью профилирования значений переменных. В этом случае необходимо использовать двухфазную компиляцию с профилированием значений указателей на процедуры, дополнительно к *-fprofile-generate* добавив опцию *-fprofile-values*. Информация о профилировании есть в блоке о *двухфазной компиляции*.

Для управления inline-подстановкой предусмотрены следующие опции:

- ключевое слово `inline`;
- атрибуты процедур:
 - `__attribute__((noinline))`;
 - `__attribute__((always_inline))`;

- опции оптимизации `-O1`, `-O2`, `-O3`;
- опция отключения `-fno-inline`;
- опции тонкой настройки:
 - `-finline-level=1.0` – коэффициент k [0.1-20.0];
 - `-finline-scale=1.0` – коэффициент увеличения основных ресурсных ограничений [0.1-5.0].

Рассмотрим пример, который на ВК «Эльбрус» будет выполняться быстрее, чем на ВК с архитектурой `x86_64`. Все примеры в этом методическом пособии будут проводиться в сравнении между следующими машинами:

- ВК «Эльбрус 801-PC» с процессором «Эльбрус-8С» (8 ядер):

```
# uname -a
Linux blabla 4.19.0-0.1-e8c #1 SMP Fri Aug 16 19:53:19 GMT 2019 e2k E8C
MBE8C-PC v.2 GNU/Linux
# lcc -v
lcc:1.24.04:Jul-2-2019:e2k-v4-linux
Thread model: posix
gcc version 7.3.0 compatible.
# cat /etc/mcst_version
4.0-rc3
```

- ВК с процессором Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (8 ядер):

```
# uname -a
Linux debion 4.9.0-3.6-amd64 #1 SMP Tue Jun 4 14:10:20 GMT 2019 x86_64
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz GenuineIntel GNU/Linux
# gcc -v
Используются внутренние спецификации.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5.5.0/lto-wrapper
Целевая архитектура: x86_64-linux-gnu
Параметры конфигурации: ../configure --build=x86_64-linux-gnu --prefix=/usr
--libexecdir=/usr/lib --enable-shared --enable-threads=posix --disable-bootstrap
--enable-__cxa_atexit --enable-multilib --enable-clocale=gnu
--enable-languages=c,c++,fortran
Модель многопоточности: posix
gcc версия 5.5.0 (GCC)
# cat /etc/mcst_version
4.0-rc4
```

Пример:

```
#include <stdio.h>

#define REP 100000
#define N 10000

void f0(int *p)
{
    *p = (*p) * (*p) + 1;
}

void f1(int *p)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
{
    *p = (*p) / ((*p) * (*p) + 1);
}
void f2(int *p)
{
    *p = (*p)<<((*) & 31);
}
void f3(int *p)
{
    *p = (*p) + (*p)<<1 + (*p)<<7 + (*p)<<4;
}
void f4(int *p)
{
    *p = (*p) ^ (int)( (*p) * 0.42 );
}
void f5(int *p)
{
    *p = (*p) % 17;
}

int r[N];

#ifdef NOINLINE
#define NOINLINE __attribute__((noinline))
#endif
void NOINLINE sample(int i)
{
    f0(&(r[i]));
    f1(&(r[i+1]));
    f2(&(r[i+2]));
    f3(&(r[i+3]));
    f4(&(r[i+4]));
    f5(&(r[i+5]));
}

int main()
{
    int i,k;

    for (i=0; i<N; i++)
        r[i]=i;

    for (k=0; k<REP; k++)
    {
        for (i=0; i<(N-5); i+=6)
        {
            sample(i);
        }
    }
    printf("%10i\n",r[0]);
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```

return (int)r[0];
}

```

Компиляция под платформу «Эльбрус»:

```

/opt/mcst/bin/lcc -O3 ./t.c -o a.elb.noinline -fno-inline
/opt/mcst/bin/lcc -O3 ./t.c -o a.elb
/opt/mcst/bin/lcc -O3 ./t.c -o a.elb.aggr -DNOINLINE=

```

Таблица 1: Время выполнения программы

Эльбрус	Intel	Опции сборки
real 5.15	real 1.83	
user 5.14	user 1.83	
sys 0.01	sys 0.00	
real 18.37	real 2.73	-O3 -fno-inline
user 18.36	user 2.73	
sys 0.00	sys 0.00	
real 0.71	real 1.70	-O3 -DNOINLINE=
user 0.71	user 1.70	
sys 0.00	sys 0.00	

Опция `-fno-inline` запрещает инлайн-подстановку, и в таком режиме программа на Эльбрусе существенно уступает в производительности. Опция `-DNOINLINE=` специфична для приведенного примера, она управляет возможностью подстановки функции `sample()` в тело цикла в функции `main()` за счет выключения атрибута `__attribute__((noinline))`. В случае инлайна всех функций `f()` в `sample()` и `sample()` в `main()` компилятор смог дополнительно воспользоваться параллельностью итераций цикла `main()`. Об этой технике подробно рассказывается в следующем разделе.

6.3 Программная конвейеризация

Под конвейеризацией принято понимать метод организации длительной циклической работы. В основе конвейеризации лежит разбиение одного цикла работы на несколько стадий, при котором каждая стадия принимает на вход результат предыдущей стадии и передает собственный результат следующей стадии; при этом для достижения желаемого эффекта время работы стадий должно быть существенно меньше исходного цикла.

В качестве примеров конвейеризированной работы можно привести:

- тушение пожара методом передачи ведер по цепочке:
 - исходный цикл:** принести полное ведро от реки до пожара и добежать от пожара до реки с пустым ведром;
 - стадии:** работа отдельных людей в цепочке по передаче полного ведра от предыдущего к следующему человеку в направлении пожара, и пустого в обратном направлении;
- автоматизированную сборку автомобиля:
 - исходный цикл:** полная сборка автомобиля (~сутки);
 - стадии:** работа отдельного комплекса роботов с полуготовым автомобилем на конвейерной ленте (~час).

В обоих случаях эффект заключается в увеличении темпа: растет количество принесенных ведер в минуту, количество собранных автомобилей в сутки.

Для циклов программы конвейеризация означает сокращение суммарного времени выполнения всех итераций цикла за счет большего темпа запуска итераций цикла в работу. Принцип работы конвейеризации цикла проиллюстрирован на рис. *Пример работы конвейера*.

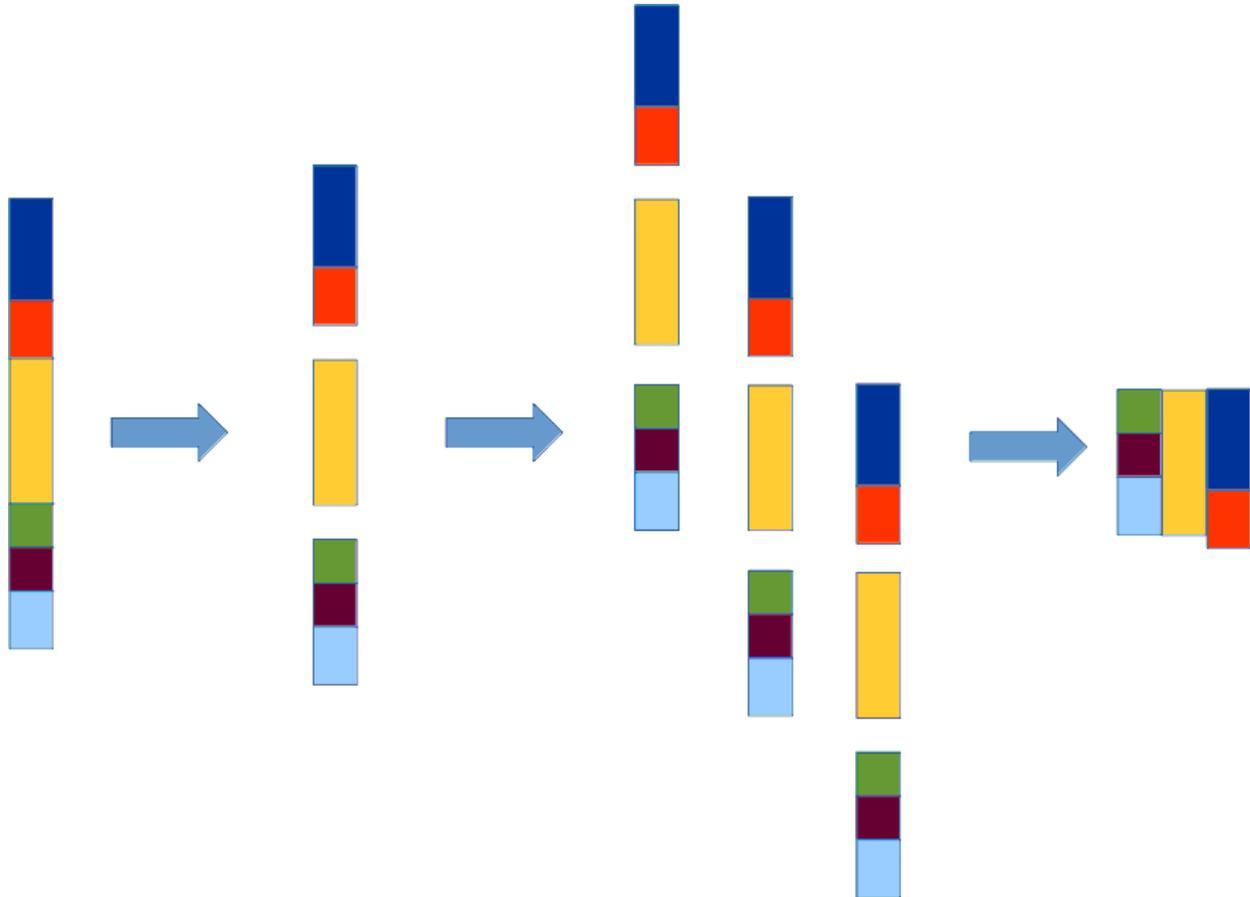


Рис. 1: Пример работы конвейера

6.3.1 Важность конвейеризации для VLIW и OOOSS

VLIW:

- Без конвейеризации все итерации цикла будут исполняться строго последовательно. Конвейеризация дает многократное ускорение циклов. Заметим, что эффект, схожий с конвейеризацией, предоставляет распространенная компиляторная оптимизация «раскрутка цикла» (loop unrolling). В этой оптимизации в одном «туловище» раскрученного цикла выполняется несколько рядом стоящих итераций.

OOOS:

- Для OOOSS конвейеризация происходит за счёт аппаратного планирования. В OOOSS предсказатель переходов предсказывает переход по обратной дуге, и это позволяет начать исполнять операции со следующей итерации еще до конца текущей. Эффективность конвейеризации цикла в OOOSS ограничена объемами ROB (буфер переупорядочения) и количеством операций на одной итерации цикла.

6.3.2 Пример ручной конвейеризации цикла

Рассмотрим цикл со слабой зависимостью между итерациями:

```
do
{
  s += 1.0/(a[i]*a[i] + 1.0);
}
while (i++ < N);
```

Ассемблер такого цикла будет следующим:

```
ldd          [%r_a  %r_i_], %r0
fmuld        %r0,    %r0,    %r1
fadd         %r1,    1.0,    %r2
fdivd        1.0,    %r2,    %r3
fadd         %r_s,   %r3,    %r_s
add          %r_i,   1,      %r_i
add          %r_i_,  8,      %r_i_
cmpl        %r_i,   %r_N,   %pred0
ct           %ctpr1 ,    %pred0
```

Операцию подготовки перехода

```
disp :loop_head, %cptr1
```

В данном случае вынесли в качестве инварианта и спланировали перед входом в цикл.

Для лучшей наглядности изменены названия регистров. Мнемоники команд сохранены. Действия описаны так, как если бы они выполнялись последовательно.

В примере использованы следующие операции:

- **ldd** — чтение;
- **fmuld** — умножение;
- **fadd** — сложение;
- **fdivd** — деление;
- **cmpl** — сравнение;
- **ct** — переход.

Из этих операций состоит цикл. Их можно выполнить в каком-то порядке, некоторые операции могут смещаться относительно других. Заметим, что большую часть операций с соседних итераций цикла можно исполнять независимо. Зависимыми от предыдущей итерации являются операции, помеченные красным цветом.

Операция «**i++**»

Операция «s+=»

Операция сложения для «i_»

Каждая из этих операций вырабатывает результат, который используется этой же операцией на следующей итерации.

Цикл написан в предположении, что мы работаем с типом double. Чтобы не делать сдвига переменной i в качестве аргумента для чтения, мы увеличиваем i_ для продвижения по массиву (i_ определяет смещение).

Для большего понимания выпишем всю последовательность элементарных действий на одной итерации цикла:

- читается a[i] по смещению i_ и кладется в %r0;
- %r0 умножается на %r0 и записывается в %r1;
- к %r1 прибавляется единица и происходит запись в %r2;
- единицу делим на %r2 и кладем результат в %r3;
- %r3 прибавляем к S и кладем результат в регистр для S;
- производим инкремент i;
- i_ увеличиваем на 8;
- сравниваем i и N;
- делаем переход в начало цикла.

В разделе *Планирование кода* уже упоминалось, что операции имеют различную длительность. Планирование этого цикла с учётом длительностей операций для процессора «Эльбрус-4С» выглядит следующим образом:

0	ldd	[%r_a	%r_i_],	%r0
0	addd	%r_i,	1,	%r_i
0	addd	%r_i_,	8,	%r_i_
1	cmpl	%r_i,	%r_N,	%pred0
2				
3	fmuld	%r0,	%r0,	%r1
4				
5				
6				
7	faddd	%r1,	1.0,	%r2
8				
9				
10				
11	fdivd	1.0,	%r2,	%r3
...				
22	faddd	%r_s,	%r3,	%r_s
22	ct	:loop_head,	%pred0	

В данном примере подняты максимально вверх те операции, на которые другие операции не оказывают давления (не зависят по регистрам или данным). Поднялись вверх операции i++ и i_+8. От операции чтения a[i] до умножения **fmuld** три такта, это обусловлено длительностью операции чтения из L1 кэша. Операция **fmuld** работает 4 такта, за него зацеплена операция сложения, которая будет работать в 7-м такте, так как ей требуется результат умножения. Операция **faddd** также работает 4 такта, соответственно операция **fdivd**, зависящая от неё по %r2, будет выполняться в 11-м такте.

Операция деления является длительной операцией и выполняется 11 тактов (или более, в зависимости от используемой разрядности), поэтому операция сложения с *S* будет производиться только в 22 такте и вместе с ней сможет выполняться операция перехода. Таким образом, одна итерация цикла требует 23 такта для выполнения.

Начнем процесс конвейеризации цикла с выполнения переноса операций с критического пути по обратной дуге. Первой из таких операций является операция чтения в 0 такте, которая отодвигала умножение в третий такт. Перенесем эту операцию по всем переходам в голову цикла - это переход по обратной дуге цикла, и вход в голову цикла извне. После переноса по обратной дуге операция чтения поднимется и окажется в такте $T=1$. Операция чтения не может подняться в нулевой такт, так как перед выполнением чтения должно произойти изменение *i_*. После переноса чтения умножение может подняться в нулевой такт, так как теперь на него ничего не давит и все, что зависело от умножения, также поднимается вверх. Но теперь умножение, пришедшее в нулевой такт, давит на сложение в 4-м такте. Также перенесем, как и **ldd**, операцию **fmuld** по обратной дуге.

```

0   addd          %r_i,  1,      %r_i
0   addd          %r_i_, 8,      %r_i_
0   fmuld         %r0,   %r0,    %r1
1   cmpl         %r_i,   %r_N,   %pred0
1   ldd  .s      [%r_a   %r_i_], %r0
3
3
4   faddd        %r1,   1.0,    %r2
5
6
7
8   fdivd        1.0,   %r2,    %r3
9
10
11
...
19  faddd        %r_s,   %r3,    %r_s
19  ct           :loop_head, %pred0

```

После переноса операции **fmuld** зависящая от нее операция **faddd** поднимется в нулевой такт. Операция **fmuld** встанет в четвертом такте, так как она зависима от операции чтения в первом такте.

```

0   addd          %r_i,  1,      %r_i

```

```

0   addd      %r_i_, 8,      %r_i_
0   faddd     %r1,  1.0,    %r2
1   cmpl     %r_i, %r_N,    %pred0
1   ldd      [%r_a %r_i_], %r0
3
3
4   fdivd    1.0,  %r2,    %r3
4   fmuld    %r0,  %r0,    %r1
6
7
8
9
10
11
...
15  faddd    %r_s,  %r3,    %r_s
15  ct      :loop_head, %pred0

```

Таким образом, цикл сократился еще на 4 такта. Следующим шагом перенесем fadd из нулевого такта:

```

0   addd     %r_i,  1,      %r_i
0   addd     %r_i_, 8,      %r_i_
0   fdivd    1.0,  %r2,    %r3
1   ldd      [%r_a %r_i_], %r0
1   cmpl     %r_i, %r_N,    %pred0
2
3   fmuld    %r0,  %r0,    %r1
3
4
5

```

```

6
7   fadd      %r1, 1.0, %r2
8
9
10
11  fadd      %r_s, %r3, %r_s
11  ct       :loop_head, %pred0
    
```

Если теперь попытаться сделать следующий шаг и перенести через обратную дугу операцию деления, поднявшуюся в 0 такт, то мы столкнемся с проблемой т.н. антизависимости операций. Термин антизависимость, или зависимость типа Write-After-Read, обозначает невозможность переставить местами две операции, из которых:

- первая читает значение из некоторого ранее определенного регистра;
- вторая записывает в этот же регистр новое значение, после чего старое значение регистра пропадает.

В нашем примере проблема заключается в антизависимости `fadd <- %r3` и `fdivd -> %r3` со следующей итерации.

Для борьбы с антизависимостью используется техника переименования регистров: нижней операции нужно назначить другой регистр для записи результата (и запомнить этот факт для дальнейших операций, использующих результат нижней операции). В OOOSS переименование регистров производится аппаратно с помощью скрытого регистрового файла. Для архитектуры Эльбрус используется механизм явного переименования регистров - базированные регистры. Они описаны в разделе *Пространство регистров подвижной базы. Программные соглашения использования базированных регистров*.

Ассемблерная мнемоника базированных регистров — `%b[]`. Циклическое переименование (вращение окна) базированных регистров с помощью операции `abn` показано на рисунке *Циклическое переименование базированных регистров операцией abn*.

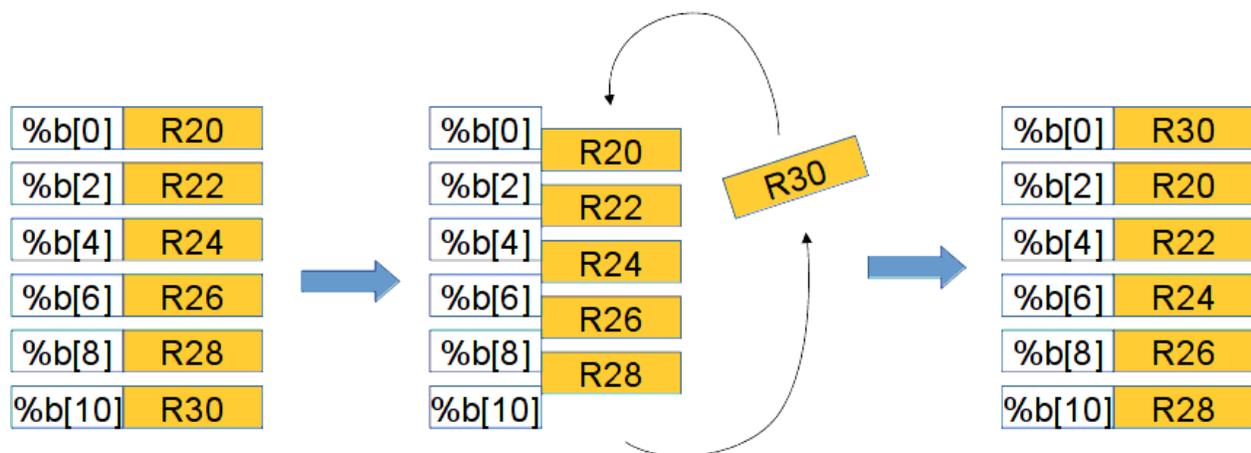


Рис. 2: Циклическое переименование базированных регистров операцией `abn`

Теперь проблема антизависимости для длинных операций на соседних итерациях цикла может быть

решена с помощью операции вращения регистров `abn`, поставленной в последнем такте цикла вместе с операцией перехода по обратной дуге. Механизм вращения позволяет длинной операции (в нашем случае операции деления `fdiv`) писать в различные физические регистры, обращаясь к ним с помощью одной и той же мнемоники `%b[X]`. Это позволит продолжить перенос по обратной дуге с той лишь разницей, что регистр, с которым будет работать длинная операция `fdiv`, будет заменен на базированный.

Приведем следующий шаг процесса переноса, сокращающий цикл до 8 тактов:

```

0   addd          %r_i,  1,      %r_i
0   faddd         %r1,   1.0,    %r2
1   addd          %r_i_,  8,      %r_i_
1   ldd           [%r_a  %r_i_], %r0
1   cmpl         %r_i,   %r_N,   %pred0
2
4   fmuld        %r0,    %r0,    %r1
4   fdivd        1.0,   %r2,    %b[0]
5
6
7   faddd        %r_s,   %b[2],   %r_s
7   abn
7   ct           :loop_head, %pred0

```

Здесь произошло сразу несколько событий:

- операции `ldd` и `fmuld` перенесены еще на одну итерацию вверх, они теперь помечены светло-зеленым;
- ради операции `ldd` перенесена по обратной дуге операция `addd`, инкрементирующая регистр `%r_i_`;
- операция `fdivd` перенесена по обратной дуге и помечена зеленым;
- добавлена операция `abn` для циклического переименования регистров `%b[0] -> %b[2]`;
- операции `fdivd` назначен базированный регистр `%b[0]`. Для использования ее значения на следующей итерации операции `faddd` назначен регистр `%b[2]` - в нем окажется нужное значение после исполнения операции `abn`.

На данном этапе длинная цепочка вычислений на одной итерации цикла разбита уже на три стадии:

```

ldd -> fmuld ->
-> faddd -> fdivd ->
-> faddd

```

Если продолжить процесс, можно довести планирование до 4-х тактов.

```

0   addd          %r_i,  1,      %r_i
0   cmpl         %r_i,   %r_N,   %pred0

```

```

1   addd          %r_i_, 8,      %r_i_
2   fmuld        %r0,  %r0,    %r1
2   faddd        %r1,  1.0,    %r2
2   fdivd        1.0,  %r2,    %b[0]
3   ldd          [%r_a  %r_i_], %r0
3   faddd        %r_s,  %b[6],  %r_s
3   abn
3   ct           :loop_head,  %pred0

```

Обратим внимание, что `fdivd` пишет в регистр `%b[0]`, а `faddd` потребляет регистр `%b[6]` - такая дистанция образовалась из-за того, что между заброшенным по обратной дуге `fdiv` и использующим его результат `fadd` успевают исполниться три операции `abn`. Сделать так потребовалось для того, чтобы выдержать задержку в 11 тактов за несколько итераций цикла длиной 4 такта.

6.3.3 Логическая и физическая итерации

Для конвейеризированных циклов приняты термины **Логическая** и **Физическая** итерация.

Логическая итерация последовательность операций одной итерации исходного цикла.

Стадия одна из n последовательных частей логической итерации, которая будет исполняться последовательно.

Физическая итерация набор совмещенных стадий различных соседних логических итераций, по составу операций совпадающий с логической итерацией, а по длине планирования - с самой длинной стадией.

Пролог разгон цикла, первые $n-1$ физических итераций цикла, предшествующих завершению исполнения первой логической итерации.

Эпилог торможение цикла, последние $n-1$ физических итераций цикла, завершающие последнюю логическую итерацию.

Для рассмотренного нами примера ручной конвейеризации длина логической итерации составляет 23 такта, длина физической итерации 4 такта, число стадий равно 6. Если у исходного цикла количество итераций было равно 100, то у конвейеризованного цикла оно составит $100 + (6-1) = 105$, т.к. к ним добавится 5 итераций пролога.

Длина физической итерации ограничена снизу:

- количеством ресурсов ШК (например, количеством АЛУ);
- длиной рекуррентности;
- длиной последней стадии, содержащей операции с побочным эффектом (например, записи или условные переопределения).

Для того, чтобы цикл смог стать конвейеризованным, необходимо выполнить ряд условий. Требуется:

- отсутствие операций вызова в теле цикла;

- наличие единственной обратной дуги и единственного выхода как альтернативы обратной дуге;
- отсутствие ветвлений в теле цикла.

Также можно выделить аспекты, которые делают конвейеризированный цикл малоэффективным:

- число итераций меньше либо сравнимо по величине с числом стадий;
- длина рекуррентности сравнима с длиной исходной итерации.

6.3.4 Примеры рекуррентности

```
i++;
```

```
0      addd    %r_i,    1,    %r_i
```

Рекуррентность ограничивает длину стадии снизу. В случае `i++` длина рекуррентности составляет 1 такт в соответствии с длительностью операции. Операция читает из регистра `r_i` и пишет в регистр `r_i`.

```
s += sqrt(t);
```

```
0      fsqrts  %r_t,    %r0
```

```
15     fadds   %r_s,    %r0,    %r_s
```

В этом примере, выполняющемся 16 тактов, длинная операция `sqrt` не участвует в рекуррентности, которая состоит из единственной операции `fadds`.

```
a[i] = a[j] + 4;
```

```
0      ldd     [%r_a_m16 %r_j_], %r0
```

```
3      addd   %r0,    4,    %r1
```

```
4      std    %r1,    [%r_a %r_i_]
```

В этом примере показана рекуррентность между чтением и записью, так как нет уверенности, пересекаются ли области памяти:

- та, из которой мы читаем со смещением `j`;
- та, в которую пишем со смещением `i`.

В распоряжении программиста есть аппаратно-программные приемы, чтобы избавиться от рекуррентности. Рекуррентность через единственную ассоциативную операцию может быть уменьшена за счет:

- раскрутки цикла:

```
s += (sqrt(t0) + sqrt(t1));
```

```
0      fsqrts    %r_t0, %r0
```

```
2      fsqrts    %r_t1, %r1
```

```
17     fadds     %r0,  %r1,  %r2
```

```
21     fadds     %r_s,  %r2,  %r_s
```

Считаем два корня, складываем первый со вторым, и их сумма входит в рекуррентную операцию `fadds`; рекуррентность осталась равна 4 тактам, но теперь на одной итерации выполняется две итерации исходного цикла. Заметим, что порядок вещественных операций при этом изменился, это может (и в большинстве случаев будет) приводить к изменениям точности результата. Для разрешения такого преобразования необходимо подать компилятору опцию `-fassociated-math` или более общую опцию `-ffast-math`.

- редукции с помощью вращаемых регистров:

```
s1 = s0 + sqrt(t);
s0 = s1;
...
// после цикла
...
s = s0 + s1;
```

```
0    fsqrts        %r_t,          %r0
15   fadds        %b[4], %r0,    %b[0]
...
// после цикла
...
fadds        %b[2], %b[0],    %r_s
```

Принцип тот же, но вместо раскрутки используется базированный регистр, заменяющий одно суммируемое значение на два, расположенных в соседних регистрах, в одном из которых соберется сумма `sqrt(t)` с операций с четным номером, а в другом - с нечетным. После цикла эти регистры необходимо сложить для получения полной суммы. Это также существенно меняет порядок вещественных операций и требует опции `-ffast-math` или `-fassociative-math`.

Отдельно нужно отметить, что операции, которые не могут выполняться спекулятивно, должны размещаться на последней стадии цикла. К таким операциям относятся, например, операции записи в память. Если при этом на итерации цикла есть операции чтения, которые зависят от операции записи, то они тоже вынуждены размещаться на последней стадии. Пример:

```
a[rnd]++;
s+=a[i];
```

```
0    ldw          [%r_a %r_rnd_], %r0
3    adds        %r0, 1,        %r1
4    stw          %r1, [%r_a %r_rnd_]
5    ldw          [%r_a %r_i_],  %r2
```

```
8   adds      %r_s, %r2, %r2
```

Это показывает важность разрыва зависимости по памяти между операциями чтения и записи в конвейеризируемых циклах. Для этого можно пользоваться различными приемами, они описаны в последующих разделах.

6.3.5 Управление конвейеризацией

Для управления конвейеризацией предусмотрены следующие опции и прагмы:

```
#pragma loop count(N)
```

подсказка компилятору о среднем количестве итераций цикла. Позволяет компилятору принять положительное решение о конвейеризации в тех случаях, когда предсказанное самим компилятором количество итераций цикла было небольшим. Эту же подсказку можно использовать для того, чтобы отключить конвейеризацию с негативным эффектом для циклов с малым количеством итераций.

```
#pragma ivdep
```

разрыв зависимостей между операциями чтения и записи, расположенными на разных итерациях цикла; позволяет уменьшить длину последней стадии и увеличить эффективность конвейеризации.

```
-fswp-maxopers=N
```

максимальное количество операций для цикла, чтобы его можно было рассматривать для конвейера. Конвейеризация полностью отключается при значении `swp-maxopers = 0`. Для значений порядка 1000 нужно бояться сильного роста времени компиляции проекта. Значение по умолчанию - 300.

```
-fforce-swp
```

отключает оценки качества конвейеризации в пользу безусловного применения. Использовать можно для экспериментов.

6.3.6 Пример с конвейеризацией цикла

```
#include <stdio.h>

#define REP 100000
#define N 30000

double const c1=1.0, c2=0.01, c3=0.99, c4=0.01, c5=1.01, c6=0.01, c7=0.98, c8=-0.01;

double a[N];

void sample(double *d)
{
    int i;
    for (i=0; i<N; i+=4)
    {
        d[i]=(((d[i]*c1+c2)*c3+c4)*c5+c6)*c7+c8;
    }
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```

}

int main()
{
    int i,k;
    for (i=0; i<N; i++)
    {
        a[i]=i/N;
    }

    for (k=0; k<REP; k++)
    {
        sample(a);
    }
    printf("%10.7f\n",a[N-4]);
    return a[N-4];
}

```

Компиляция под платформу «Эльбрус»:

```

/opt/mcst/bin/lcc -O3 ./t.c -o a.03 -fno-inline
/opt/mcst/bin/lcc -O3 -ffast ./t.c -o a.03f -fno-inline
/opt/mcst/bin/lcc -O3 -fswp-maxopers=0 -ffast ./t.c -o a.03noswp -fno-inline

```

Компиляция под платформу Intel:

```

gcc ./t.c -O3 -fno-inline

```

Таблица 2: Время выполнения программы

Эльбрус	Intel	Опции сборки
0.9700975	0.9700975	-O3 -fno-inline
real 1.05	real 1.32	
user 1.05	user 1.32	
sys 0.00	sys 0.00	
0.9700975	Опция -ffast не поддерживается	-O3 -ffast -fno-inline
real 0.69		
user 0.69		
sys 0.00		
0.9700975	Опция -fswp-maxopers=0 не поддерживается	-O3 -fswp-maxopers=0 -ffast -fno-inline
real 4.12		
user 4.11		
sys 0.00		

Первым полем перед временами исполнения в таблице приведено значение результата программы.

При этом на последних версиях компилятора lcc-1.24 даже с опцией -fswp-maxopers=0 сработает аппаратная подкачка массива с помощью **apb**. В результате может получиться очень быстрое вычисление, сбивающее с толку, так как результат ожидался более медленный.

```

/opt/mcst/bin/lcc -O3 -fswp-maxopers=0 -ffast ./t.c -o a.03noswp -fno-inline

hostname /export/minis/01_swp # time -p ./a.03noswp

```

```
0.9700975
real 0.58
user 0.58
sys 0.00
```

6.4 Слияние альтернатив условий

Рассмотрим простейшую условную конструкцию:

```
int f(int c, int *p)
{
    int r=0;

    if (c>0)
        r += *p;
    return r;
}
```

При компиляции с выключенной оптимизацией мы увидим следующий код:

```
f:
    {
        setwd wsz = 0x4, nfx = 0x1
    }
    {
        nop 4
        adds,0,sm    0x0, 0x0, %r4
        cmplesb,1    %r0, 0x0, %pred0
        disp %ctpr1, .L5; ipd 2
    }
    {
        ct    %ctpr1 ? %pred0
    }
.L8:
    {
        nop 2
        ldw,0 %r1, 0x0, %r5
    }
    {
        adds,0      %r4, %r5, %r4
    }
.L5:
    {
        nop 5
        sxt,0,sm    0x2, %r4, %r0
        return     %ctpr3; ipd 2
    }
    {
        ct    %ctpr3
    }
}
```

Видим, что код получился очень рыхлым - на 6 операций приходится 14 тактов, если условие не выполняется, и 18 тактов, если условие выполняется. Во всех трех участках кода, разделенных метками, количество тактов определяют операции подготовки перехода **disp** и **return**.

Архитектура Эльбрус предоставляет возможность сделать этот код более компактным за счет замены операций условного перехода на предикатный (условный) режим исполнения отдельных операций.

```
f:
{
  setwd wsz = 0x4, nfx = 0x1
  return      %ctpr3; ipd 2
  adds,0      0x0, 0x0, %g16
}
{
  nop 1
  cmplesb,0   %r0, 0x0, %pred0
}
{
  nop 3
  ldw,0 %r1, 0x0, %g16 ? ~%pred0
}
{
  ct   %ctpr3
  sxt,3 0x2, %g16, %r0
}
```

Рассмотрим, как работает приведенный код. Видим, что меток в коде не осталось, есть только одна операция подготовки возврата из процедуры и одна операция возврата. Операция сравнения **cmplesb** все так же записывает результат в предикатный регистр **%pred0**, но теперь он используется не для перехода, а для управления исполнением операции **ldw**. Это задается с помощью мнемоники:

```
oper arg1, arg2, res ? pred
```

Если значение предикатного регистра равно *0*, операция не будет выполнена. Можно задать управление с помощью инвертированного предиката:

```
oper arg1, arg2, res ? ~pred
```

В этом случае операция не будет выполнена при значении предикатного регистра *1*. Таким образом, в регистр **%g16** есть две записи:

- безусловная запись значения *0* операцией **adds**;
- условная запись прочитанного из памяти значения **r* операцией **ldw**.

Время спланированного исполнения такого кода составляет 8 тактов, что заметно лучше исходного времени 14-18 тактов.

Показанный прием называется слиянием альтернатив условного кода, или предикацией. Далее будем называть его «*слиянием кода*». Его можно применять не только к элементарным условным конструкциям, но и к более сложным. При этом условия, управляющие исполнением отдельных операций, будут вычисляться посредством разных операций сравнения. Для этого в архитектуре Эльбрус предусмотрены логические операции над предикатными регистрами.

Доступны следующие операции над предикатами:

- пересылка предиката в локальный предикат и обратно;

```
pass      %pred0, @p0
pass      @p1,    %pred2
```

- логическая функция & с возможностью инверсии аргументов;

```
andp      @p2,    ~@p3,    @p4
```

- пересылка предиката под условием;

```
movp      @p2,    ~@p3,    @p4
```

В одной ШК могут выполняться одновременно до трех логических операций **andp/movp**, при этом две из них могут быть зацеплены зависимостью следующим образом:

```
%pred3 = %pred0 & ~%pred1
%pred4 = %pred3 & %pred 2
```

В ассемблере предикатные операции выглядят так:

```
pass      %pred0, @p0
pass      %pred1, @p1
pass      %pred2, @p2
andp      @p0,    ~@p1,    @p3
pass      @p3,    %pred3
andp      @p3,    @p2,    @p4
pass      @p4,    %pred4
```

6.4.1 Важность слияния кода для VLIW и OOSS

VLIW:

- среднее количество операций на линейном участке слишком мало для выявления ILP;
- перемешивание операций с соседних участков позволяет повысить ILP;
- объединение линейных участков позволяет уменьшить долю дорогостоящих операций перехода.

OOSS:

- ввиду наличия аппаратного предсказания траекторий исполнения кода слияние требуется в редких, плохо обнаруживаемых случаях непредсказуемых ветвлений;
- поддержка условных операций в системе команд довольно слабая, зачастую реализованы только условные пересылки.

6.4.2 If-Conversion

If-Conversion — это слияние нескольких линейных участков, связанных друг с другом в подграф графа потока управления, в один участок кода, с использованием предикатного режима исполнения отдельных операций. Цель слияния - увеличение контекста поиска и выражения параллелизма уровня операций, а также удаление операций перехода.

Рассмотрим немного более сложный пример:

```

#define UL unsigned long

void adds(UL as, UL am, UL bs, UL bm, UL *ress, UL *resm, UL s)
{
    if (as == bs) {
        *resm = am + bm;
        *ress = as;
    }
    else if (am >= bm) {
        *ress = as;
        if (!s)
            (*resm)++;
    }
    else {
        *ress = bs;
        if (!s)
            (*resm)--;
    }
}

```

На рисунке ниже изображен граф потока управления, соответствующий этой процедуре. Под таким графом традиционно понимается граф, узлами которого являются линейные участки кода, завершающиеся единственной операцией передачи управления (перехода), а дугами соединяются линейные участки, между которыми возможны переходы. При этом переход может выполняться в одно, два или множество мест (см. рис. *Граф потока управления и траектории исполнения кода*).

Теперь рассмотрим ассемблерное содержимое линейных участков (см. рис. *Разветвленный код*). Узлы представлены номерами N1, . . . N8. Каждому узлу в соответствие приведено количество тактов, за которое он выполняется. Внутри каждого узла представлен набор ассемблерных операций, реализующих данный узел.

Рассмотрим маленький участок этого графа между первым и вторым узлом. Из первого узла есть два выхода в узел N2 и в узел N7. Из второго также есть два выхода в узлы N3 и N5. Суммарное время выполнения узлов - 12 тактов. В обоих узлах используется операция подготовки перехода **disp**, операция сравнения **cmp** и операция перехода **ct**.

Перенесем операцию **cmpb** из узла 2 вверх в узел 1. На рис. *Перенос операций* показано, что в таком случае произойдет с узлами.

В узле 2 остались операции подготовки и перехода, и на данном этапе планирование не уменьшилось. Эти операции также можно перенести в узел 1, тем самым превратив его в //гиперузел// и изменив планирование. В гиперузле получим три выхода — два перехода и один провал (см. рис. *Гиперузел*).

Имеем две операции подготовки, две операции сравнения, две операции перехода и новую операцию **landp** - логическую операцию *И* над двумя предикатными регистрами. Она необходима для определения сложного условия перехода в узел N5,. Время выполнения гиперузла станет порядка 6,5 тактов: иногда мы будем выходить через 6 тактов в узел N7, а иногда через 7 тактов в узлы N3 и N5.

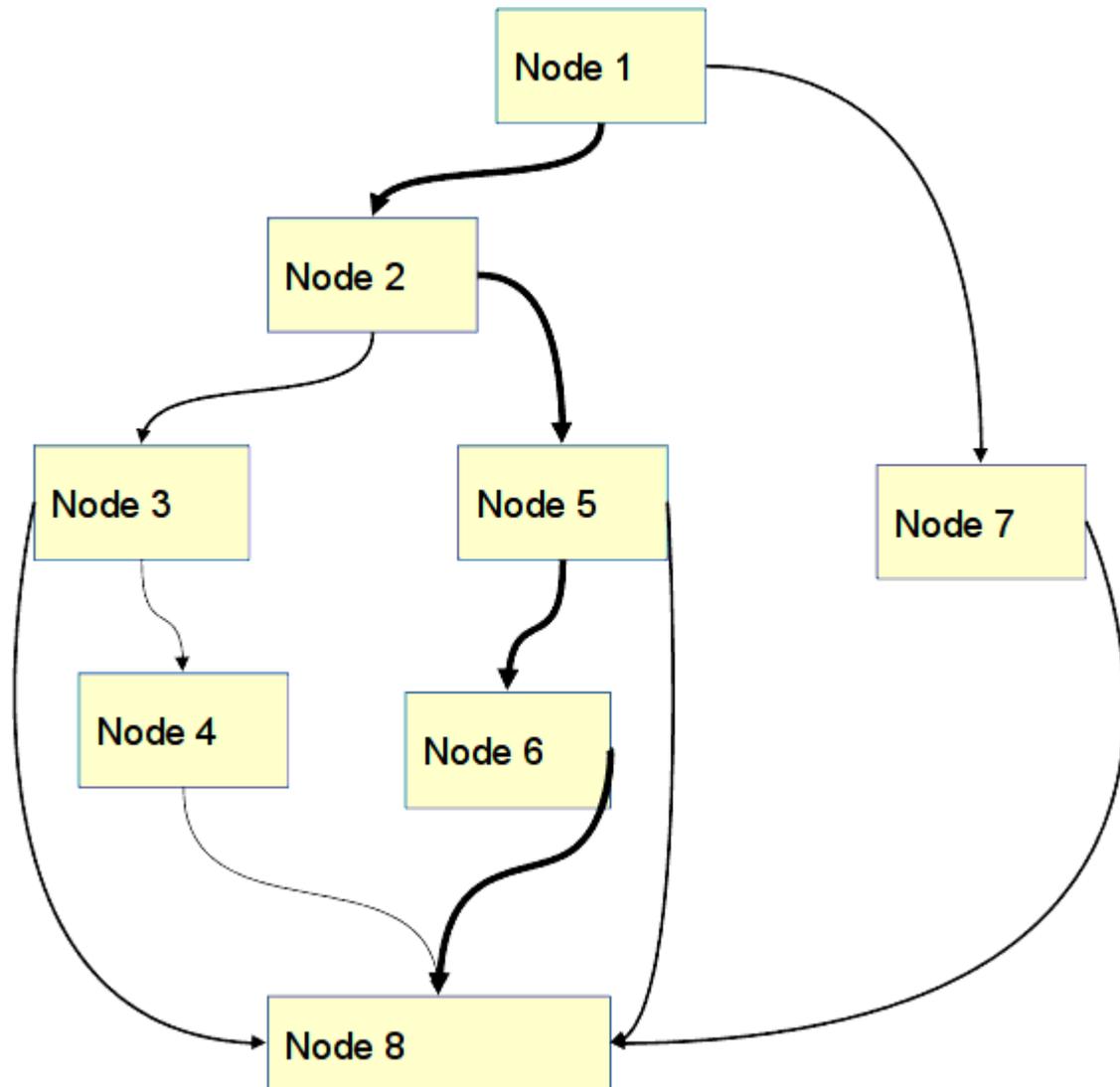


Рис. 3: Граф потока управления и траектории исполнения кода

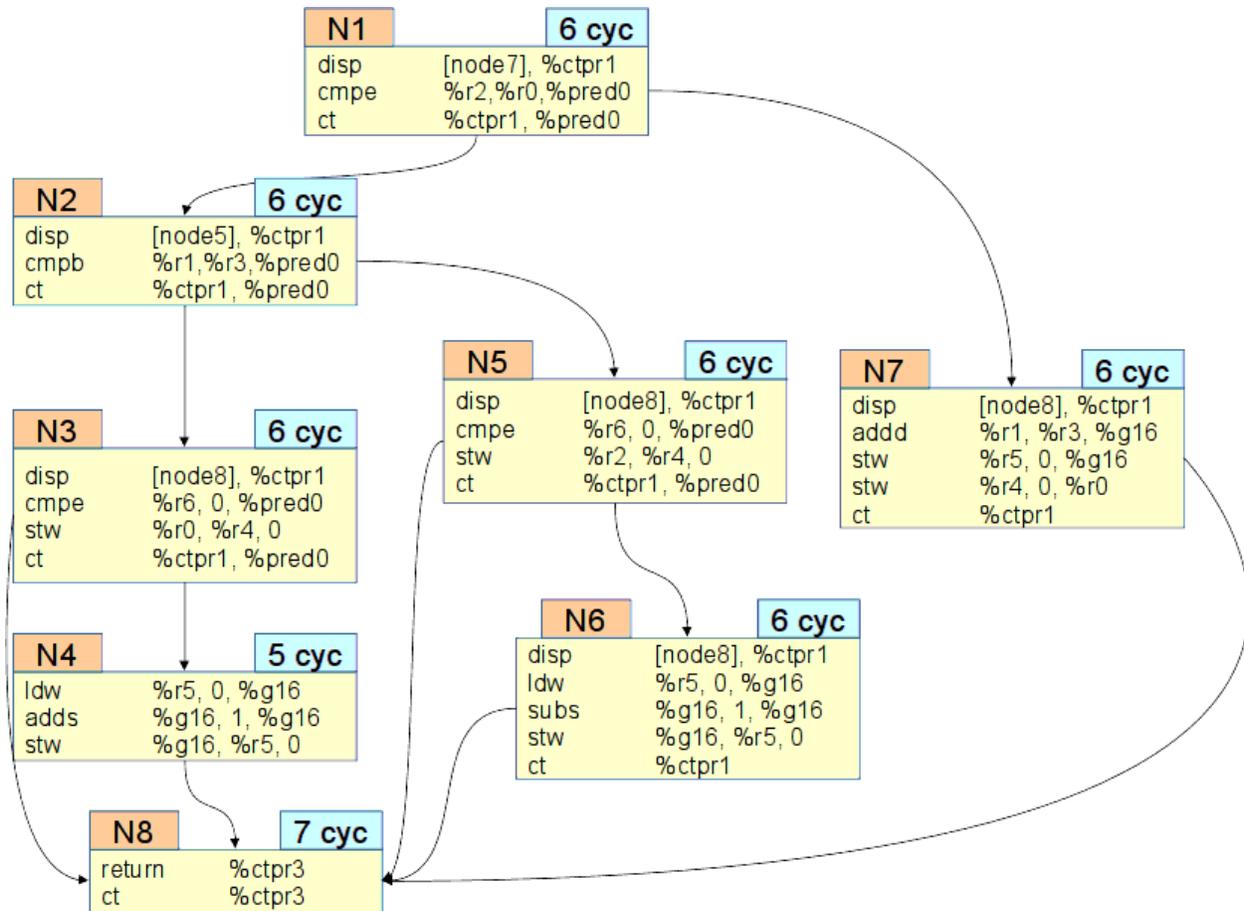


Рис. 4: Разветвленный код

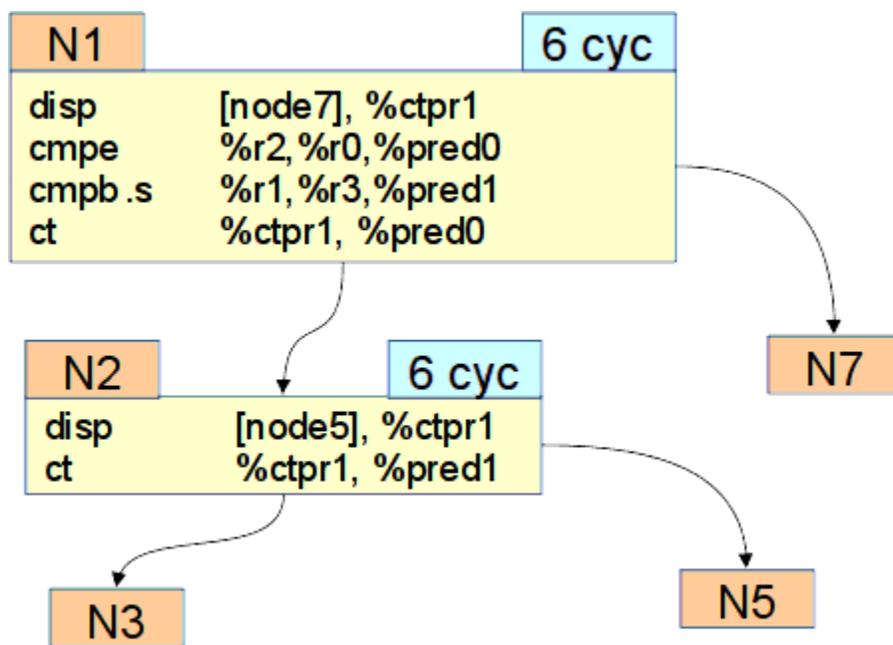


Рис. 5: Перенос операций

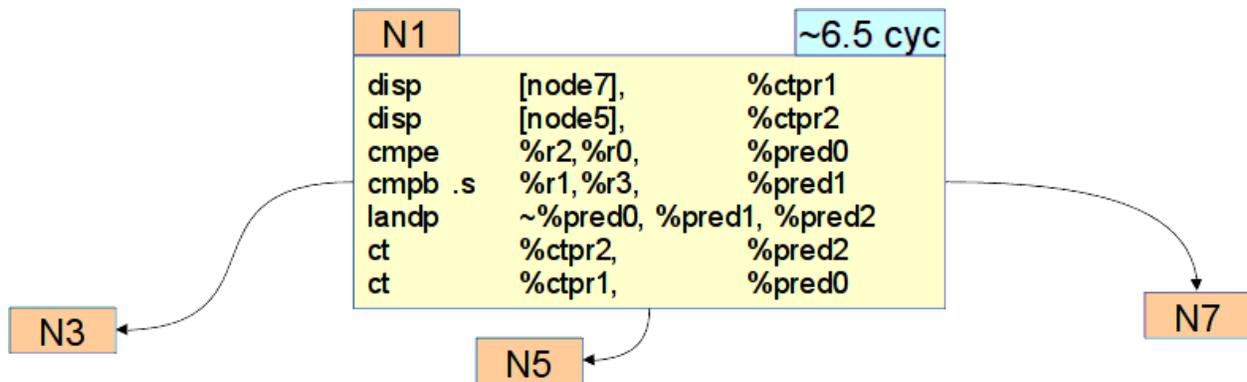


Рис. 6: Гиперузел

6.4.3 Спекулятивный режим

Для повышения параллелизма на уровне операций часто бывает полезно начать выполнять операцию до того, как станет известно, нужно ли было выполнять эту операцию. В случае, если операция может привести к прерыванию (например, чтение из памяти по недопустимому адресу, или деление на 0), такое преждевременное исполнение может привести к некорректному аварийному выходу из программы. Для того, чтобы решить эту проблему, в архитектуре «Эльбрус» допускается выполнение операций в режиме отложенного прерывания, также называемом спекулятивным режимом исполнения операции. Для хранения информации об отложенном прерывании ячейки памяти, числовые и предикатные регистры снабжены дополнительным битом (тэгом), по одному биту на одно слово (1 слово = 4 байта = 32 бита) в памяти, по два на 64-битный числовой регистр (по одному на старшую и младшую части), и по одному на каждый предикатный регистр. 1 в бите тэга означает, что значение регистра указывает на отложенное прерывание (диагностическое значение).

Примерная схема работы такова:

- вместо выработки прерывания операция, имеющая спекулятивный признак (везде далее «спекулятивная операция»), записывает 1 в диагностический бит результата;
- диагностический операнд у спекулятивной операции приводит к выработке в ней диагностического результата;
- диагностический операнд у несспекулятивной операции приводит к выработке отложенного прерывания.

Возможности оптимизации, предоставляемые спекулятивным режимом исполнения, таковы: в спекулятивном режиме операция может быть исполнена раньше, чем условный переход на линейный участок, содержащий эту операцию, и логика исполнения программы при этом сохраняется.

Пример:

```
int* a;
...
if (a!=NULL)
{
  *a++;
}
...
```

Такой код может быть выполнен при помощи следующих операций:

```

0   disp      label    => ctp1    ! подготовка обходной метки
0   cmpe      Ra,      0 => p0    ! сравнение указателя на равенство 0
4   ct        ctp1     ? p0     ! условный переход на обходную метку
5   ld        Ra,      0 => Rx    ! чтение из указателя a
8   add       Rx,      1 => Ry    ! инкремент результата чтения
9   st        Ra,      0 <= Ry   ! запись в a результата инкремента
label:

```

время работы — 10 тактов.

Спекулятивный режим позволяет спланировать код с большей параллельностью:

```

0   disp      label    => ctp1    ! подготовка обходной метки
0   cmpe      Ra,      0 => p0    ! сравнение на равенство
0   ld.s      Ra,      0 => Rx    ! чтение из указателя a
3   add.s     Rx,      1 => Ry    ! инкремент результата чтения
4   ct        ctp1     ? p0     ! условный переход на обходную метку
5   st        Ra,      0 <= Ry   ! запись в a результата инкремента
label:

```

время работы — 6 тактов.

Обратим внимание, что операции **ld** и **add**, поставленные в спекулятивный режим, будут исполнены независимо от результата сравнения, причем при **a=NULL** результат чтения будет диагностическим (чтение по адресу NULL вызывает прерывание, которое будет отложено). При этом результат операции **add**, содержащий диагностическое значение, не будет использован, т.к. операция **st** исполнится только при **a!=NULL**.

Скомбинировав предикатный и спекулятивный режимы, можно получить еще более быстрый код:

```

0   ld.s      Ra,      0 => Rx    ! чтение из указателя a
0   cmpe      Ra,      0 => p0    ! сравнение на равенство
3   add.s     Rx,      1 => Ry    ! инкремент результата чтения
4   st        Ra,      0 <= Ry ? ~p0 ! запись под предикатом не-p0

```

время работы — 5 тактов.

Отметим, что приведенный код является более параллельным, чем в чистом предикатном режиме, поскольку цепочка операций **ld.s** и **add.s** не зацеплена за операцию сравнения, вырабатывающую предикат.

Также отметим, что некоторые операции нельзя исполнять в спекулятивном режиме, такие как передача управления или запись в память.

6.4.4 Итог слияния примера со сложным условным кодом

Используя спекулятивный и предикатный режим, показанный в начале этого раздела, граф потока управления в виде ассемблерных операций можно слить в один гиперузел с временем выполнения 7 тактов. Такой узел выглядел бы следующим образом:

```

return      %ctpr3
ldw .s      [%r5 + 0x0], %g16
adds .s     %r1, %r3, %g18
cmpe .s     %r6, 0x0, %pred0
cmpb .s     %r1, %r3, %pred1

```

(continues on next page)

(продолжение с предыдущей страницы)

cmpe	%r0,	%r2,	%pred2
landp	~%pred2,	~%pred1,	%pred3
landp	%pred3,	%pred0,	%pred4
landp	~%pred2,	%pred1,	%pred5
landp	%pred5,	%pred0,	%pred6
subs .s	%g16,	0x1,	%g17
adds .s	%g16,	0x1,	%g16
stw	%r0,	[%r4 + 0x0]	? %pred2
stw	%g18	[%r5 + 0x0]	? %pred2
stw	%r0,	[%r4 + 0x0]	? %pred3
stw	%r2,	[%r4 + 0x0]	? %pred4
stw	%g16,	[%r5 + 0x0]	? %pred5
stw	%g17,	[%r5 + 0x0]	? %pred6
ct	%ctpr3		

6.4.5 Управление слиянием кода

При слиянии кода часть одного узла сливается с другим. Однако другие узлы могут иметь свои дуги на данный слитый узел. Ввиду этого происходит дублирование кода.

Влияние опций и других факторов на работу слияния:

- `-01` – слияние не производится;
- `-02` – слияние работает, дублирование сильно ограничено;
- `-03`, `-04` – слияние работает, дублирование разумное;
- закрытые ассемблерные вставки препятствуют набору регионов;
- профиль от двухфазной компиляции помогает весьма сильно;
- без профиля помогают подсказки `__builtin_expect()`.

6.4.6 Пример на слияние кода

```
#include <stdio.h>
#include <stdlib.h>

#define REP 100000
#define N 10000

int rn[N+63];
int a,b,c;

void sample(int ind)
{
    int S;
    S = rn[ind];
    if ((S&1)&&(S<(1<<30)))
        a++;
    if (((S>>3)&1)&&(S<(1<<29)))
        b++;
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```

    if (((S>>5)&1)&&(S<(1<<28)))
        c++;
}

int main()
{
    int i,k,S;

    for (i=0; i<N; i++)
        rn[i]=rand();

    for (k=0; k<REP; k++)
        for (i=0; i<N; i++)
        {
            sample(i+(k&63));
        }

    printf("%d %d %d\n",a,b,c);
    return 0;
}

```

Компиляция под платформу «Эльбрус»:

```

/opt/mcst/bin/lcc -O3 ./t.c -o a.03 -fno-inline
/opt/mcst/bin/lcc -O1 ./t.c -o a.01 -fno-inline

```

Компиляция под платформу Intel:

```

gcc ./t.c -O3 -fno-inline

```

Время выполнения программы:

Таблица 3: Время выполнения программы

Эльбрус	Intel	Опции сборки
244176687 123914132		-O1 -fno-inline
63020334		
real 42.64		
user 42.62		
sys 0.00		
244176687 123914132	244176687 123914132	-O3 -fno-inline
63020334	63020334	
real 19.21	real 14.30	
user 19.19	user 14.30	
sys 0.01	sys 0.00	

Для наглядности рекомендуется на платформе «Эльбрус» получить ассемблерный код и увидеть разницу между функцией **sample** с оптимизацией *-O1* и *-O3*. Будет заметна разница в наполненности ШК, а также в большом количестве дорогостоящих операций **disp** в неоптимизированном варианте.

6.4.7 Последствия явного спекулятивного режима

Некоторые последствия спекулятивного режима были ранее представлены в книге в разделе SIGILL как сигнал о ошибках (см. *SIGILL как сигнал об ошибках*)

Рассмотрим последствия использования явного спекулятивного режима на примере:

```
#include <stdio.h>
#include <fenv.h>
#include <math.h>

double zero = 0;

double
own_sin (double x)
{
    fenv_t env;
    feholdexcept (&env);
    return sin(x);
}

double
own_fma (double x, double y, double z)
{
    if (!__builtin_isfinite (x) || !__builtin_isfinite (y))
        return x * y + z;
    else if (!__builtin_isfinite (z))
        return (z + x) + y;

    if ((x == 0 || y == 0) && z == 0)
    {
        x = ({ __typeof (x) __x = (x); __asm (" : "+m" (__x)); __x; });
        return x * y + z;
    }

    fenv_t env;
    feholdexcept (&env);

    long double x1 = (long double) x * ((1ULL << (64 + 1) / 2) + 1);
    long double y1 = (long double) y * ((1ULL << (64 + 1) / 2) + 1);
    long double m1 = (long double) x * y;
    x1 = (x - x1) + x1;
    y1 = (y - y1) + y1;
    long double x2 = x - x1;
    long double y2 = y - y1;
    long double m2 = (((x1 * y1 - m1) + x1 * y2) + x2 * y1) + x2 * y2;

    long double a1 = z + m1;
    long double t1 = a1 - z;
    long double t2 = a1 - t1;
    t1 = m1 - t1;
    t2 = z - t2;
    long double a2 = t1 + t2;
```

(continues on next page)

(продолжение с предыдущей страницы)

```

    a2 = a2 + m2;
    a1 = a1 + a2;
    return a1;
}

int main ()
{
    double res = own_fma(1.23, 7.8, 9)/zero;

    // double res = own_sin(0.5);
    printf ("\nres=%e sticky=%x", res, fetestexcept(FE_ALL_EXCEPT));
    return 0;
}

```

own_sin() - это вызов синуса с обнуленными исключениями или sticky битам.

own_fma() - это часть глобального fma. Там много лишнего, но это не позволяет компилятору все это свернуть и подставить готовый результат. Функция также очищает sticky выработывает inexact (или precision) exception. Деление на 0 после вызова own_fma() дает еще divide by zero exception.

В варианте запуска own_fma():

на x86:

```
gcc ~/t.c -lm -static; ./a.out
res=inf sticky=24
```

на e2k:

```
gcc ~/t.c -lm -static -O3; ./a.out
Illegal instruction
```

С запретом спекулятивных плавающих операций:

```
gcc -O3 t.c -static -lm -fno-spec-fp ; ./a.out
res=inf sticky=24
```

Вариант с запуском own_sin (надо откомментировать вызов own_sin и закомментировать вызов own_fma):

на x86:

```
gcc ~/t.c -lm -static; ./a.out
res=4.794255e-01 sticky=20
```

на e2k:

```
gcc -O3 t.c -static -lm -fno-spec-fp ; ./a.out
Illegal instruction
```

Тут и опция отказа от спекулятивных операций не спасает, поскольку падение происходит в библиотечном синусе.

Существующие проблемы:

в режиме закрытых масок и нулевыми sticky-битами спекулятивная операция, налетевшая на fr-исключение, выработает диагностику; из-за этого будет падение из-за замаскированного fr-исключения, которое формально будет выглядеть как sigill

в режиме открытых масок спекулятивная операция выдаст диагностику, на операции, использующей ее результат будет пойман sigill. Поэтому не стоит, рассчитывать на fr-исключения, так как написанный перехватчик для них, не сможет поймать и обработать этот sigill

Одним из решений может быть использование опции `-fno-fr-spec` для запрета спекулятивности fr-операций

Другое решение это сборка в этом режиме некоторых библиотечных функций, в частности, тех, которые управляют fr-исключениями

6.5 Конфликты операций работы с памятью. Анализ указателей. Разрыв зависимостей

6.5.1 Виды зависимостей между операциями

Операции внутри некоторого участка кода могут быть связаны зависимостями. Чем меньше зависимость между операциями, тем большего параллелизма можно достигнуть.

Существует 4 типа зависимостей между операциями (см. рис. *Виды зависимостей между операциями*):

Flow истинная зависимость, отражает поток данных. Одна операция пишет в регистр, следующая читает. Операция умножения пишет в `%r1`, операция сложения читает из `%r1`.

Output редкая зависимость, её пример: два раза производится запись в один и тот же регистр `%r1` под разными условиями. Разрешается несовместными предикатами - теми, которые в логическом выражении оба не равны единице, т.е. их произведение равно нулю.

Anti зависимость решается с помощью переименования регистров. На рис. *Виды зависимостей между операциями* регистр `%r0` надо успеть прочесть в операции `faddd`, так как впоследствии в этот регистр будет помещено новое значение операцией `fsubd`.

Memory конфликт доступа в память. Разноплановая зависимость, требует множества подходов. В примере на рис. *Виды зависимостей между операциями* есть операция записи и чтения, каждая работает со своим адресом. Зачастую на этапе компиляции про такие операции нельзя сказать, конфликтуют они или нет.

Типичный связанный фрагмент потока данных выглядит таким образом, что первично выполняются операции чтения из памяти параметров или результатов вызовов, после этого происходят некоторые арифметические операции, завершающиеся записью в память, переходом или возвратом. На практике не возникает необходимость поднимать операции записи над чтением. Последняя стадия конвейера, как правило, включает в себя запись. Если мы поднимем чтение над записью, то это чтение также останется на последней стадии конвейера. Это нежелательный результат.

Flow

$y = x * x + 1.0;$

fmuld	%r0, %r0,	%r1
faddd	%r1, _lts 1.0,	%r2

Output

$y = x + 1.0;$
 $\text{if (c) } y = x * 4.0;$

faddd	%r0, _lts 1.0	%r1
fmuld	%r0, _lts 4.0,	%r1

Anti

$y = x + 1.0;$
 $x = z - 2.0;$

faddd	%r0, _lts 1.0	%r1
fsubd	%r2, _lts 2.0,	%r0

Memory

$*p = a;$
 $s += *q;$

std	%r2,	[%r0 + 0]
ldd	[%r1 + 0],	%r3

Рис. 7: Виды зависимостей между операциями

6.5.2 Зависимости по памяти

Можно выделить 4 типа зависимости по памяти:

- 1) Зависимость отсутствует, чтения не влияют друг на друга.

```
...
ldd      [%r0 + 0],      %r2
ldd      [%r1 + 0],      %r3
...
```

- 2) Разрыв позволяет переставлять записи, но это лишь немного улучшает параллельность операций.

```
...
std      %r2,            [%r0 + 0]
std      %r3,            [%r1 + 0]
...
```

- 3) Бессмысленная. Необходимость поднимать записи выше чтений на практике не возникает.

```
...
ldd      [%r1 + 0],      %r3
std      %r2,            [%r0 + 0]
...
```

- 4) Самая важная. Разрыв зависимости позволяет потенциально совместить большие связанные группы операций.

...

```
std          %r2,          [%r0 + 0]
```

```
ldd          [%r1 + 0],    %r3
```

...

6.5.3 Важность разрешения зависимостей по памяти

VLIW:

- операции с потенциальной зависимостью уменьшают ILP;
- наличие мемогу-зависимостей часто снижает эффективность конвейеризации циклов;
- некоторые оптимизации, в том числе универсальные, не могут применяться из-за мемогу-зависимостей.

OOOSS:

- в момент исполнения операций становятся известны значения адресов, это позволяет устанавливать зависимость операций точно, а не потенциально;
- некоторые универсальные оптимизации не могут применяться из-за мемогу-зависимостей.

6.5.4 Способы разрыва зависимостей по памяти

Выделяют два глобальных типа разрыва зависимостей по памяти:

Compile-time:

- анализ указателей в пределах процедуры;
- межпроцедурный анализ указателей;
- анализ на основе типов данных;
- на основании подсказок, опций, атрибутов.

Run-time:

- скалярный разрыв, основанный на сравнении адресов;
- цикловой разрыв, основанный на сравнении адресов;
- разрыв с использованием аппаратной поддержки.

6.5.4.1 Статический анализ конфликтов по памяти

Анализ указателей в пределах процедуры определяет источник записи адреса в каждый указатель. Варианты, откуда он может прийти:

- локальная переменная;
- глобальная переменная;
- адрес памяти, выделенной аллокатором;
- значение адреса, пришедшее из параметра.

Выделенная аллокатором память для разных `malloc` внутри одной процедуры является гарантированно независимой. Также память, выделенная аллокатором, не конфликтует с памятью для локалов и глобалов. Поэтому для таких выделений памяти в рамках одной процедуры операции `ldd` можно будет спокойно поднять над `std`. Но если мы анализируем чтение из указателя, полученного в качестве параметра, про который мы не можем сказать, откуда он выделен, мы не можем поднять чтение над записью.

Межпроцедурный анализ является расширением статического. В нем производится анализ мест, откуда пришел указатель между процедурами — производится межпроцедурная пропация списков локаций. Точки вызова процедур рассматриваются как присваивания фактических параметров в формальные.

Рассмотрим пример:

```
...
char *p0, *p1;
p0=&local_var;
p1=malloc(sz);

f(p0,p1);
...

void f(char* param_p0, char* param_p1)
{
    (*param_p0)++;
    (*param_p1)++;
    return;
}
```

В этом примере без межпроцедурного анализа в функции `f` нельзя сказать, перестановочны ли чтения с записями. Проведя межпроцедурный анализ, станет ясно, что в `*param_p0` содержится адрес из локалов, а в `*param_p1` - из кучи. Тем самым будет обеспечена возможность перестановки чтения и записи.

Существующая проблема межпроцедурного анализа - в точках вызова по косвенности происходит массовая склейка формальных параметров различных процедур, анализ, соответственно, слабеет.

Еще одним вариантом разрыва `memoгу`-конфликтов является спецификатор `__restrict`. Пример его применения можно увидеть в объявлении функции `memcpy()` в файле `string.h`.

Спецификатор `__restrict` для указателя `<p>` говорит, что к памяти, на которую указывает `<p>`, можно обращаться только через этот указатель, разыменовывая его. Т.е. `<p>` будет указывать на уникальную память. Для компилятора это важная информация, которая означает, что операции чтения/записи с адресом, формируемым из `__restrict` переменной, зависимы только с другими обращениями по этой переменной. Ответственность за независимость адресов в случае использования `__restrict` перекладывается с компилятора на программиста.

В примере ниже операцию чтения из `*bp` можно поднять над записью ввиду того, что указатель `*rp` объявлен со спецификатором `__restrict`.

```
{
  double * __restrict rp = param0;
  Double * bp = param1;
  ...
  *rp = a/(a*a+1);
  b = *bp;
  ...
}
```

Следующим вариантом разрыва memopy-конфликтов является использование «strict aliasing rule».

`strict aliasing rule` запрещает обращаться к одной памяти с помощью указателей несовместимых типов:

`int*` и `float*` – нет

`short*` и `long*` – нет

`signed int*` и `unsigned int*` – да

`int*` и `int**` – нет

Исключение – `char*` и `void*`, можно обращаться к любой памяти.

Для компилятора это означает, что операции чтения/записи несовместимых типов независимы. В примере ниже операцию чтения из `*ip` можно поднять над операцией записи в `*dp` ввиду того, что `*dp` имеет тип `double`, а `*ip` имеет тип `int`:

```
{
  double * dp = param0;
  int * ip = param1;
  ...
  *dp = sqrt(a);
  b = *ip;
  ...
}
```

Для использования `strict aliasing rule` необходимо скомпилировать ваш проект с опцией `-fstrict-aliasing` или с опцией `-ffast`.

6.5.4.2 Динамический разрыв конфликтов по памяти

Рассмотрим run-time разрыв зависимостей или динамический разрыв зависимостей, при котором компилятор строит дополнительный код, чтобы в динамике разрешать конфликты по памяти, подобно OOOSS.

Первый метод динамического разрыва основан на сравнении адресов. Он применим только для выровненных указателей одинакового размера. Для использования возможности такого динамического разрыва проект или файл должен быть скомпилирован с использованием опции `-faligned`.

Рассмотрим пример:

```
{
  int * p0 = param0;
  int * p1 = param1;

  ...
  *p0 = a;
  b = *p1;
  ...
}
```

В примере неизвестно, пересекаются ли адреса для указателей `p0` и `p1`. Поэтому операцию чтения из `p1` нельзя выполнить раньше записи в `p0`. Однако, если собрать проект с опцией `-faligned`, компилятор сможет построить дополнительный код. В языке Си такой код представлял бы из себя преждевременное чтение из `p1` в некоторую переменную `_b0`, после происходила запись «`a`» в `*p0` с последующим использованием тернарного оператора, который бы проверял равенство адресов `p1` и `p0` и в зависимости от результата записывал бы в `b` либо значение «`a`», либо значение «`_b0`».

Пример:

```
{
  ...
  _b0 = *p1;
  *p0 = a;
  b = (p1 == p0)? a : _b0;
  ...
}
```

На языке ассемблерного кода для платформы «Эльбрус» этот пример выглядел бы так:

```
...
ldw      [%r1 + 0],    %r4
cmpe     %r1,    %r0,    %pred0
stw      %r2,        [%r0 + 0]
merges   %r4,    %r2,    %pred0,    %r3
...
```

В зависимости от времени выполнения `ldw` мы можем выиграть 1-2 такта.

Подобным методом разрыва зависимостей в динамике является сравнение адресов — RTMD. Оптимизация RTMD применима для циклов с базовой индуктивностью и линейно изменяющимися адресами конфликтующих переменных.

Рассмотрим простой цикл с двумя конфликтующими адресами:

```

{
  int * p0 = param0;
  int * p1 = param1;

  for (i=0; i<N; i++)
    p0[i] = p1[i] + 1;
}

```

Компилятор может построить дополнительный код, превратив этот цикл в два цикла под разными условиями. Первый цикл будет закладываться на то, что `p0` и `p1` не конфликтуют. Второй же, напротив, будет закладываться на то, что эти адреса нельзя считать неконфликтующими. Выглядеть это будет следующим образом:

```

{
  if ((p0+N) < p1) || ((p1+N) < p0)
  {
    int * __restrict rp0 = p0;
    int * __restrict rp1 = p1;
    for (i=0; i<N; i++)
      rp0[i] = rp1[i] + 1;
  }
  else
    for (i=0; i<N; i++)
      p0[i] = p1[i] + 1;
}

```

Важно, что первично будет происходить сравнение адресов на предмет того, что они не пересекаются — правая граница `p0` меньше левой границы `p1` и правая граница `p1` меньше `p0`. Если условие этого выражения истинно, то компилятор создаст новые указатели со спецификатором `__restrict`, которым присвоит старые указатели `p0` и `p1` и будет выполнять цикл с ними. Это позволит забрасывать чтения из `rp1` на всем диапазоне адресов цикла над записями в `rp0`. В случае, если проверяемое выражение оказалось ложью, то будет выполняться обычный первоначальный цикл.

Расчет здесь сделан на то, что чаще адреса будут независимы, и компилятор не зря подготовил дополнительный код и делает дополнительную проверку. Если снять профиль со счётчиками при выполнении такой процедуры и в профиле будет видно, что мы часто попадаем в вариант, где есть конфликт адресов, то рекомендуется выключить режим RTMD с помощью опции `-fno-loop-rtmd`. Минусом этого подхода является необходимость попарного сравнения всех конфликтующих адресов, например, для 8 чтений и 6 записей потребуется построить 96 сравнений.

Последний вариант разрыва конфликтов в динамике является аппаратно-программным. Он уже упоминался при описании спекулятивности по данным. Механизм построен на использовании аппаратной таблицы DAM. Рассмотрим пример:

```

{
  int * p0 = param0;
  int * p1 = param1;

  ...
  *p0 = a;
  b = *p1 * 14;
  ...
}

```

В этом примере нельзя поднять операцию чтения из `*p1` и зацепленную за неё операцию умноже-

ния выше записи в `*r0`. Но с использованием аппаратной таблицы DAM это становится возможно. Компилятор может выполнить спекулятивно операцию чтения и умножения. При этом чтение будет выполняться с тегом `mas=0x4`, это запирающее чтение. Оно поместит адрес чтения в аппаратную таблицу.

После чтения выполнится запись в `*r0`, над которой мы спекулятивно выполнили чтение. При этом запись будет выбивать из аппаратной таблицы, куда был занесен адрес при запирающем чтении, любые адреса, пересекающиеся с этой записью, даже пересекающиеся частично. После записи компилятор поместит проверочное чтение с тегом `mas=0x3`. Это чтение будет проверять, остался ли адрес, внесенный в таблицу, запирающим чтением. Если адрес остался, то чтение и запись никак не пересекались, и перенос был правомерным. Если же записи не осталось в аппаратной таблице, будет произведен переход на компенсирующий код, в котором придется заново производить умножение. При этом появятся дополнительные штрафы при переходе на компенсирующий код и возврат из него. Также операция проверочного чтения удалит из таблицы проверенную строку.

В этой технике, как и в RTMD, сделан расчет на то, что конфликта адресов между чтением и записью при исполнении чаще всего нет, и в компенсирующий код мы будем попадать редко. Если в профиле со счётчиками видно, что мы часто попадаем в компенсирующий код, то рекомендуется отключить спекулятивность по данным опцией `-fno-dam`.

Для лучшего понимания пример, описанный выше, проиллюстрирован на рис. *Аппаратная поддержка, таблица DAM*.

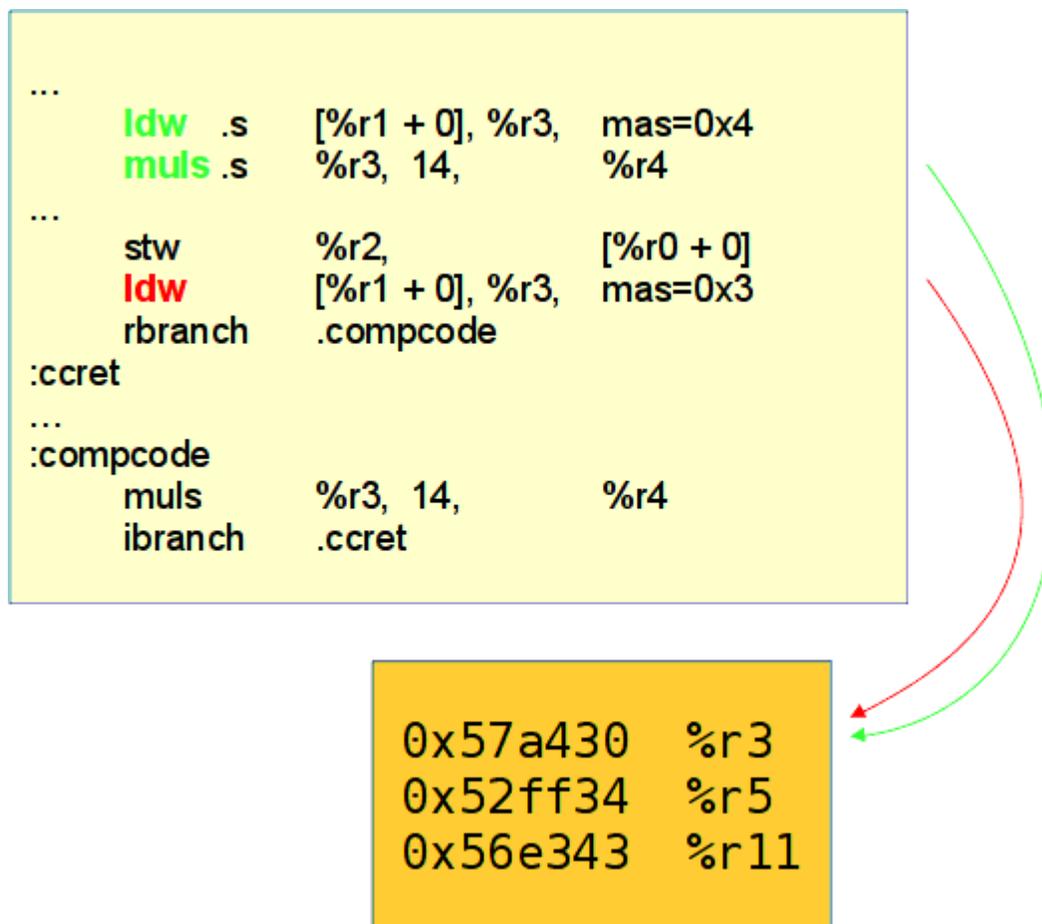


Рис. 8: Аппаратная поддержка, таблица DAM

Подытоживая раздел с разрывом конфликтов по памяти, можно еще выделить несколько опций, ко-

торые могут пригодиться программистам.

Это опции:

- `-fwhole` - уощняет межпроцедурный анализ;
- `-frestrict-all` автоматически проставляет спецификатор `__restrict` на все указатели;
- `-frestrict-params` автоматически проставляет спецификатор `__restrict` на указатели-параметры процедур.

6.6 Предварительная подкачка данных

Чтение из памяти - операция, имеющая различную длительность. Разница обусловлена наличием аппаратных кэш-памятей, ускоряющих этот доступ. Без них время доступа было бы одинаково долгим, порядка 100-200 тактов. Кэши-памяти, как правило, устроены иерархически: от небольших кэшей с быстрым доступом до больших кэшей с медленным доступом. Маленький кэш первого уровня L1 расположен ближе всего к вычислительным ресурсам и обеспечивает наименьшее время доступа. При планировании операций компилятор по-умолчанию считает, что операция чтения всегда попадает в кэш данных первого уровня. Проблема в том, что кэш L1 удерживает далеко не все данные, за которыми обращается программа. При каждом кэш-промахе возникает блокировка. Её длительность варьируется:

~7-9 тактов, если данные находятся в L2 кэше;

~40 тактов, если данные находятся в L3 кэше;

100-200 тактов, если данные находятся в оперативной памяти.

Такты во время блокировки не тратятся на что-либо полезное, в результате получается низкая скорость исполнения операций за такт. Такие блокировки для VLIW архитектур являются очень критичными. Для OOOSS блокировки являются менее критичными, так как они могут совмещаться в рамках окна переупорядочивания операций для нескольких промахнувшихся операций чтения. Также для современных OOOSS реализован механизм автоматической подкачки линейных данных в кэш-память, основанный на обнаружении линейных паттернов доступа.

Для решения проблемы с блокировками конвейера, вызванными операциями чтения, в архитектуре «Эльбрус» предусмотрено несколько методов:

- Ациклические участки кода:
 - совмещение блокировок;
 - ограничение на простановку маловероятных чтений в спекулятивный режим.
- Цикловые участки кода:
 - совмещение блокировок в конвейеризированных циклах;
 - выявление регулярных чтений, предподкачка с помощью `prefetch (ld->empty)`;
 - использование аппаратно-программного механизма для подкачки линейных данных.

Так как это пособие является практическим, описание совмещения блокировок компилятором и простановки маловероятных чтений в спекулятивный режим будет пропущено.

Встроенная функция `__builtin_prefetch()` позволяет заблаговременно подкачать данные в кэш-память. Подкачивается целая порция кэш-строк - 64 байта.

Синтаксис:

```
__builtin_prefetch(addr, rw, locality)
```

addr - адрес памяти для предподкачки.

rw (0..1) - подкачка для чтения либо записи.

0 - (умолчание) подкачка для чтения.

1 - подкачка записи, означает простановку признака эксклюзивности подкачанной кэш-строке (зарезервировано для версий системы команд >5).

locality (0..3) - уровень темпоральной локальности (ожидание переиспользования).

3 (умолчание) означает сохранение во всех кэшах.

0 - можно выкидывать из кэша сразу после использования.

Locality фактически используется для указания уровня кэш-памяти подкачки.

При использовании `__builtin_prefetch()` в ассемблере можно увидеть спекулятивное чтение, но не в регистр назначения, а в `empty`:

```
ld .s [addr, offset] →%empty, mas=...
```

Уровень кэш-памяти регулируется параметром `mas`:

0x0 (умолчание) - заводить везде

0x20 - не заводить в L1\$

0x40 - не заводить в L1\$ и L2\$

0x60 - не заводить в кэшах (для подкачки не используется)

В обычных циклах компилятор умеет находить регулярно зависящие от индуктивности чтения и добавлять опережающие на `dist` итераций операции `ld → empty`. Когда компилятор не смог или побоялся префетчить данные, это стоит сделать вручную.

```
for (i=0; i<N; i++)
{
  { // block }
  s += a[i];
  { // block }
}
```

Например, так:

```
for (i=0; i<N; i++)
{
  { // block }
  s += a[i];
  __builtin_prefetch(&a[i+dist]);
  { // block }
}
```

Для расчета `dist` компилятор использует округленное в большую сторону отношение времени доступа в память и оценки времени планирования итерации цикла. `dist` вычисляется следующим образом:

```
dist = Lat (ld ← mem) / T (iter)
```

Для использования предподкачки в циклах следует использовать комбинацию префетчей.

Пусть

```
a[ i ]           - 0-линейное чтение
b[ a[ i ] ]     - 1-линейное чтение
c[ b[ a[ i ] ] ] - 2-линейное чтение
```

Компилятор умеет обнаруживать и строить предподкачку для n-линейно-регулярных чтений (на практике $n \leq 3$). Общий принцип - строится n подкачек:

```
a[ i+(n+1)*dist ]
b[ a[ i+(n)*dist ] ]
c[ b [ a[ i+(n-1)*dist ] ] ]
...
```

В коде это может выглядеть следующим образом:

```
for (i=0; i<N; i++)
{
  { // block }
  s += p->values->coords->x;
  p++;
  { // block }
}
```

С использованием префетча:

```
for (i=0; i<N; i++)
{
  { // block }
  s += p->values->x;
  p++;
  __builtin_prefetch(&((p+3*dist)->values));
  __builtin_prefetch(&((p+2*dist)->values->coords));
  __builtin_prefetch(&((p+ dist)->values->coords->x));
  { // block }
}
```

Для увеличения производительности в случае чтения в цикле по линейно изменяющимся адресам ($a[i]$, $b[k*i]$, $(p++)$ и т.п.) в архитектуре ``Эльбрус`` реализован механизм асинхронного доступа к элементам массива. Суть его состоит в следующем: доступ к массивам описывается особым образом в виде кода асинхронной программы. Она состоит только из операций ***farb**. Операции **farb** запускаются по циклу в независимом конвейере, пополняя буферы упреждающих данных для разных массивов. При этом основной поток исполнения забирает данные из этого буфера операциями **movb** (вместо запуска операций чтения).

Преимущества, предоставляемые механизмом асинхронного доступа к массивам:

- вместо операции чтения, занимающей ALU, используется операция **movb**, занимающая отдельный канал, что освобождает в широкой команде место под арифметическую операцию;
- блокировки из-за отсутствия данных существенно уменьшаются, т.к. операции доступа к памяти, имеющие непредсказуемую длительность, выполняются асинхронно и не блокируют основной поток исполнения операций. При этом операции **movb** имеют фиксированную длительность.

В одной широкой команде можно исполнить до 4 операций чтения из буфера **АРВ**. Программисту в редком случае может понадобиться разбираться в тонкостях работы **АРВ**. В этом методическом

пособии данный вопрос пропущен. Достаточно понимать, что механизм **ARB** приносит существенную пользу. **ARB** можно применять:

- при отсутствии вызовов функций в цикле - механизм не допускает сохранения и восстановления при вызове;
- при выровненности адресов; требование снято для форматов 2, 4 и 8 в версии v5 системы команд; снято для формата 16 в версии v6 системы команд.

Также отдельно стоит выделить случаи, когда механизм **ARB** эффективен:

- при достаточно большом числе итераций;
- при отсутствии зависимостей с записями между итерациями, либо при достаточно большой дистанции зависимости.
- при инкременте адреса $<32b$ **arb** более эффективен, чем при больших инкрементах;

На основании соображений эффективности компилятор может отказываться от применения **arb** для конкретных циклов.

Существуют также ресурсные ограничения применения **arb** в цикле:

- число различных массивов не может превышать 32;
- число различных инкрементов адреса не может превышать 8;
- порция данных, читаемая из одного массива в рамках одной итерации, не может превышать 32 байта; если требуется большее количество данных (например, читается массив структур, либо к циклу применена раскрутка), компилятор разбивает их на порции $<32b$ и считает различными массивами

Для управления предподкачкой в арсенале программиста есть следующие опции компилятора:

- **-fprefetch** - включает применение предподкачки в циклах;
- **-fcache-opt** - включает применение совмещения блокировок в конвейеризированных циклах;
- **-fno-arb** - выключает применение **arb**;
- **-faligned** (также входит в состав **-ffast**) - разрешение компилятору предполагать выровненность операций обращения к памяти по формату. Является фактически необходимой для возможности применения **arb** для чтений формата больше 1 байта, в версиях системы команд V1-V4. Опция **-faligned** также влияет на работу других оптимизаций, и при этом возлагает на пользователя ответственность за выровненность обращений: компиляция с опцией **-faligned** программ с нарушением выровненности может приводить к ошибкам. Для проверки нарушения выровненности рекомендуется использовать опцию **-faligned-check**, по которой включается аппаратный контроль выравнивания обращений к памяти - невыровненное обращение к памяти вызовет сигнал SIGBUS.

Использование оптимизированных библиотек

7.1 Общие сведения

Все производители современных процессоров разрабатывают и поставляют пользователям высокопроизводительные библиотеки, обеспечивающие скорость работы, близкую к максимальной для данного типа процессоров. Примерами таких библиотек являются библиотеки IPP/МКЛ фирмы Intel, библиотеки mediaLib/Perflib фирмы Oracle, библиотеки ACML/APL фирмы AMD.

Для процессоров линии «Эльбрус» также была разработана библиотека EML - высокопроизводительная математическая и мультимедийная библиотека, представляющая из себя набор разнообразных функций для обработки сигналов, изображений, видео, математических вычислений.

7.2 Состав

Библиотека eml состоит из следующих разделов:

Ядро (Core) выделение и освобождение памяти, номер версии и статус.

Вектор (Vector) различные операции над векторами: арифметические, логические, преобразование типов, математические функции, статистика.

Сигналы (Signal) цифровая обработка сигналов: конволюция, фильтрация, усиление, генерация, быстрые преобразования Фурье и Хартли.

Изображение (Image) создание и заполнение изображений, арифметические операции, фильтрация, геометрические и цветовые преобразования, ДПФ.

Линейная Алгебра (Algebra) стандартные пакеты работы с матрицами и векторами BLAS 1/2/3, LAPACK.

Видео (Video) обработка видео: интерполяция, усреднение, оценка движения, цветовые преобразования, ДКП, квантизация.

Графика (Graphics) рисование/закрашивание точек/линий/треугольников/прямоугольников/ полигонов/дуг/окружностей/эллипсов, закрашивание/перекрашивание области.

Объем (Volume) бросание параллельных/произвольных лучей с интерполяцией, линейное масштабирование вокселей, поиск максимальных значений на луче.

7.3 Информационная система

Полная документация поставляется вместе с самой библиотекой. Она находится в файле `/opt/mcst/doc/eml/index.html`.

Также начинать поиск справочной информации можно в `/opt/mcst/doc/eml/annotatid.html`.

7.4 Примеры использования

Рассмотрим простейшие варианты применения библиотеки `eml`.

7.4.1 Умножение векторов

Самое простое перемножение без `eml`. Компилируем пример с оптимизацией `-O3`. Размер вектора и количество повторений выбраны достаточно большими, чтобы показать вариант, когда расчёты не помещаются в кэш-память. Пример ниже стоит скомпилировать на ВК «Эльбрус» и на машине с архитектурой `x86`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define N 100000
#define M 1000000

int main()
{
    int i,j;
    double s=0.0;
    double *A;
    double *B;
    double *C;
    A = (double*)malloc(N * sizeof(double));
    B = (double*)malloc(N * sizeof(double));
    C = (double*)malloc(N * sizeof(double));

    srand(time(NULL));
    for (i = 0; i < N; i++)
    {
        A[i] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
        B[i] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
    }

    for (j=0; j<M; j++)
        for(i = 0; i < N; i++)
            C[i] = A[i] * B[i];
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```

s+=C[0];
free(A);
free(B);
free(C);

return (int)s;
}

```

Пример с использованием библиотеки eml будет отличаться двумя строчками, помеченными звездочкой (* после комментариев //). Чтобы скомпилировать пример, надо добавить в сборку линковку библиотеки eml:

```
gcc -O3 -o eml mul_vector_eml.c -lem1
```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <eml/eml.h>

#define N 100000
#define M 1000000

int main()
{
    int i,j;
    double s=0.0;
    double *A;
    double *B;
    double *C;
    A = (double*)malloc(N * sizeof(double));
    B = (double*)malloc(N * sizeof(double));
    C = (double*)malloc(N * sizeof(double));

    srand(time(NULL));
    for (i = 0; i < N; i++)
    {
        A[i] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
        B[i] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
    }

    for (j=0; j<M; j++)
        eml_Vector_Mul_64F(A,B,C,N);

    s+=C[0];
    free(A);
    free(B);
    free(C);

    return (int)s;
}

```

Проверить корректность работы можно путем вывода вектора и результата на печать:

```
printf("vector A\n");
for (i = 0; i < N; i++)
{
    printf("%F ", A[i]);
}
printf("\nvector B\n");
for (i = 0; i < N; i++)
{
    printf("%f ", B[i]);
}
printf("\nthe result\n");

for (i = 0; i < N; i++)
{
    printf("%f ", C[i]);
}
printf("\n");
```

Результаты приведены в таблице *Время выполнения программы умножения векторов*.

Таблица 1: Время выполнения программы умножения векторов

Интел	Эльбрус	Эльбрус + eml
real 64.90	real 163.65	real 63.71
user 64.89	user 163.54	user 63.66
sys 0.00	sys 0.01	sys 0.01

7.4.2 Умножение матриц

Преимущество использования библиотеки `eml` более явно проявляется на примере умножения матриц. Для `eml` критично, чтобы матрица была объявлена как линейный массив, через `A[i*N+j]`. Если удобнее обращаться через `A[i][j]`, тогда нужно объявлять массив как `A[M][N]`. Объявление `double **A` приведет к нелинеаризованному представлению.

Линеаризованный массив выглядит следующим образом:

A11	A12	A13	A14	A21	A22	A23	A24	A31	...
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$$C_{ij} = \text{alpha} * \sum_k (A_{ik} * B_{kj}) + \text{beta} * C_{ij}$$

Рис. 1: Общая функция умножения матриц `dgemm`

Параметры:

Order	признак расположения матрицы в памяти по строкам или по столбцам.
TransA	признак транспонирования матрицы A.
TransB	признак транспонирования матрицы B.
M	число строк матрицы A.

(continues on next page)

(продолжение с предыдущей страницы)

N	число столбцов матрицы B.
K	число столбцов матрицы A.
alpha, beta	входные константы.
A	входная матрица общего вида.
lda	leading Array Dimension - ведущая размерность массива A.
B	входная матрица общего вида.
ldb	leading Array Dimension - ведущая размерность массива B.
C	выходная матрица общего вида.
ldc	leading Array Dimension - ведущая размерность массива C.

В EML, Blas, Lapack:

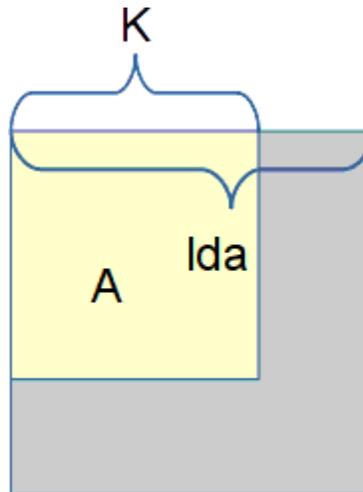


Рис. 2: Матрица A

Пример умножения матриц без использования eml:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1000
#define M 100
int main()
{
    int i,j,k,l;
    double s = 0.0;
    double *A = (double*)malloc(N * N * sizeof(double));
    double *B = (double*)malloc(N * N * sizeof(double));
    double *C = (double*)malloc(N * N * sizeof(double));

    srand(time(NULL));
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i * N + j] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
            B[i * N +j] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
        }
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    }
    for (l=0;l<M;l++)
    {
        for(i = 0; i < N; i++)
            for(j = 0; j < N; j++)
            {
                for(k = 0; k < N; k++)
                    C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
    }

    /*
    printf("matrix A\n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
            printf("%f ", A[i * N + j]);
        printf("\n");
    }
    printf("\nmatrix B\n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
            printf("%f ", B[i * N + j]);
        printf("\n");
    }
    printf("\nthe result of multiplying\n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
            printf("%3f ", C[i * N + j]);
        printf("\n");
    }
    */
    s+=C[0];

    free(A);
    free(B);
    free(C);

    return (int)s;
}

```

Пример умножения матриц с использованием eml:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <eml.h>

#define N 1000
#define M 100

```

(continues on next page)

(продолжение с предыдущей страницы)

```

int main()
{
    int i,j,k;
    double s=0.0;
    double *A = (double*)malloc(N * N * sizeof(double));
    double *B = (double*)malloc(N * N * sizeof(double));
    double *C = (double*)malloc(N * N * sizeof(double));

    srand(time(NULL));
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i * N + j] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
            B[i * N +j] = (double)(rand()%10000)/(double)10000+(double)(rand()%10000);
        }
    for (i=0; i<M;i++)
        eml_Algebra_GEMM_64F(EML_MATRIX_ROW_MAJOR, EML_MATRIX_NO_TRANS,
                            EML_MATRIX_NO_TRANS, N, N, N, 1, A, N, B, N, 0, C, N);

    s+=C[0];
    free(A);
    free(B);
    free(C);

    return (int)s;
}

```

Результаты приведены в таблице *Время выполнения программы умножения матриц*.

Таблица 2: Время выполнения программы умножения матриц

Интел	Эльбрус	Эльбрус + eml
real 316.88	real 1207.73	real 14.72
user 316.82	user 1206.92	user 14.69
sys 0.01	sys 0.03	sys 0.02

Рассмотрим шаги, которые могут улучшить результат умножения матриц. Данные приёмы уже реализованы в библиотеке eml для повышения производительности при умножении матриц.

Классическое гнездо циклов для перемножения матриц выглядит следующим образом:

```

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)
            c[i][j] += a[i][k] * b[k][j];

```

Для того, чтобы избавиться от лишних чтений из памяти, первое, что можно сделать – присваивать значение $a[i][k] * b[k][j]$ не в $c[i][j]$, а в переменную, хранящуюся на регистре. Тогда наше гнездо будет выглядеть следующим образом:

```

for (i = 0; i < M; i++)
{

```

(continues on next page)

(продолжение с предыдущей страницы)

```

for (j = 0; j < N; j++)
{
    s = 0.0;
    for (k = 0; k < K; k++)
        S += a[i][k] * b[k][j];
    c[i][j] = s;
}
}

```

Это улучшит время приведенного примера с теми же параметрами:

```

real 448.55
user 448.24
sys 0.02

```

Для расчета такого гнезда требуется $2 \cdot M \cdot N \cdot K$ вещественных операций, в предположении соизмеримости размеров матриц. Сложность такого алгоритма $O(N^3)$.

Для оптимизации внутреннего цикла применимы следующие оптимизации:

1. `k++` и `k < N` заменяются на аппаратный счетчик циклов **LSR**;
2. операции `ld` из массивов `a[i][k]` и `b[i][k]` заменяются на `mova`.

После у нас есть два подхода для оптимизации внутреннего цикла:

- **Конвейеризация:**

- ресурсы: 2 `mova`, 1 `fmul`, 1 `fadd`;
- рекуррентность: 4 такта (**fadd** зависит от **fadd** на предыдущей итерации);
- результат: 0.5 `flop/cycle`;

- **loop unroll + балансировка рекуррентности (-ffast-math), конвейеризация:**

- ресурсы: 16 `mova`, 8 `fmul`, 8 `fadd`;
- рекуррентность: 4 тактов;
- результат: 4.0 `flop/cycle`.

Второй вариант выглядит намного лучше обычной конвейеризации. Время, полученное в результате такого выполнения, будет следующим:

```

real 167.63
user 167.52
sys 0.01

```

Цикл, приведенный в примере, будет выглядеть так:

```

for (l=0;l<M;l++)
{
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            {
                s0=s1=s2=s3=s4=s5=s6=s7=0.0;
                for(k = 0; k < N; k+=8)
                    {

```

(continues on next page)

(продолжение с предыдущей страницы)

```

s0 += A[i*N+k] * B[k*N+j];
s1 += A[i*N+k+1] * B[k+1*N+j];
s2 += A[i*N+k+2] * B[k+2*N+j];
s3 += A[i*N+k+3] * B[k+3*N+j];
s4 += A[i*N+k+4] * B[k+4*N+j];
s5 += A[i*N+k+5] * B[k+5*N+j];
s6 += A[i*N+k+6] * B[k+6*N+j];
s7 += A[i*N+k+7] * B[k+7*N+j];
}
C[i*N+j]=s0;
C[i*N+1+j]=s1;
C[i*N+2+j]=s2;
C[i*N+3+j]=s3;
C[i*N+4+j]=s4;
C[i*N+5+j]=s5;
C[i*N+6+j]=s6;
C[i*N+7+j]=s7;
}
}

```

После применения второго варианта появятся проблемы при дальнейшей оптимизации внутреннего цикла:

```

for (k = 0; k < N; k+=8)
{
  s0 += a[i][k ] * b[k ] [j];
  s1 += a[i][k+1] * b[k+1] [j];
  ...
  s7 += a[i][k+7] * b[k+7] [j];
}

```

Проблема 1: плохая локальность по памяти для массива **b**.

Решение: транспонирование матрицы **B**. Сложность $O(N^2)$ относительно мала.

Проблема 2: одно чтение происходит для одной флотовой операции.

Решение: переход от оптимизации цикла к оптимизации гнезда циклов.

После транспонирования цикл будет выглядеть так:

```

for (i = 0; i < M; i+=2)
{
  for (j = 0; j < N; j+=2)
  {
    s00 = s01 = s10 = s11 = 0.0;
    for (k = 0; k < K; k++)
    {
      s00 += a[i ] [k] * b[j ] [k];
      s01 += a[i+1] [k] * b[j ] [k];
      s10 += a[i ] [k] * b[j+1] [k];
      s11 += a[i+1] [k] * b[j+1] [k];
    }
    c[i][j ] = s00; c[i+1][j ] = s01;
  }
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    c[i][j+1] = s10; c[i+1][j+1] = s11;
  }
}

```

Обращаем внимание на тот факт, что чтение `a[i][k]` не зависит от `j`; применение `unroll&fuse` (`unroll&jam`) к циклу по `j` позволит сделать одно чтение `a[i][k]` для одной раскрученной итерации `j`.

Аналогично `unroll&fuse` по `i` позволит сделать одно чтение `b[j][k]` для одной раскрученной итерации `i`.

При оптимизации гнезда циклов важно не упираться в ресурсы аппаратуры: количество вещественных операций, которое мы можем выполнять в одной ШК, количество доступных регистров. Поэтому для архитектуры `e8c` было бы правильным подобрать параметры следующим образом:

Подбор параметров `unroll&fuse` по `i` на 8 и по `j` на 6 и дальнейшая конвейеризация:

- ресурсы: $8+6 = 14$ `mov`, $8*6 = 48$ `fmul_add`;
- рекуррентность: 8 тактов (длительность `fmul_add`).

Успешная конвейеризация в 8 тактов, и соотношение `счет/память` стало существенно лучше:

$12/14 = 0.857$ `flop/byte` (> 0.375).

Результат: 12.0 flop/cycle (!)

При этом гнездо будет выглядеть следующим образом:

```

for (i = 0; i < M; i+=8)
{
  for (j = 0; j < N; j+=6)
  {
    s00 = ... = s75 = 0.0;
    for (k = 0; k < K; k++)
    {
      s00 += a[i ][k] * b[j ][k];
      ...
      s75 += a[i+7][k] * b[j+5][k];
    }
    c[i][j] = s00; ... ; c[i+7][j+5] = s75;
  }
}

```

Рекомендации по оптимизации программ под архитектуру Эльбрус

8.1 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 **АРВ**; при регулярном обращении с большим шагом (>64b) **АРВ** все еще применим, но темп существенно падает (до 64 раз при побайтовой обработке);

- одномерные массивы структур.

При регулярной обработке применим **АРВ**, 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 $p=p \rightarrow next$, 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 **АРВ** 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 **АРВ** к перебору элементов хэша неприменим, тем не менее, возможна предварительная подкачка элементов хэша, считываемых на следующих итерациях.

8.2 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.

8.3 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.

8.3.1 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.

8.3.2 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.

8.3.3 Процедура с горячими простыми циклами/гнездами циклов

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

- плохая наполненность широких команд;
- не применен 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 вычисляемую 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.

8.3.4 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.

8.3.5 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 ручную выделение важных фрагментов в отдельные процедуры.

8.3.6 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.

8.3.7 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.

Интерфейсные программные соглашения

9.1 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.

9.1.1 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.

Таблица 1: 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
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	?	?

9.1.2 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 защищенность. В регулярную страницу возможен доступ по абсолютному адресу в виде целого числа, а также по дескриптору.

Архитектурное 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.

9.1.3 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.

9.1.4 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.

9.1.4.1 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.

Таблица 2: 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.

9.1.4.2 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.

9.1.4.3 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).

9.1.4.4 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.

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.

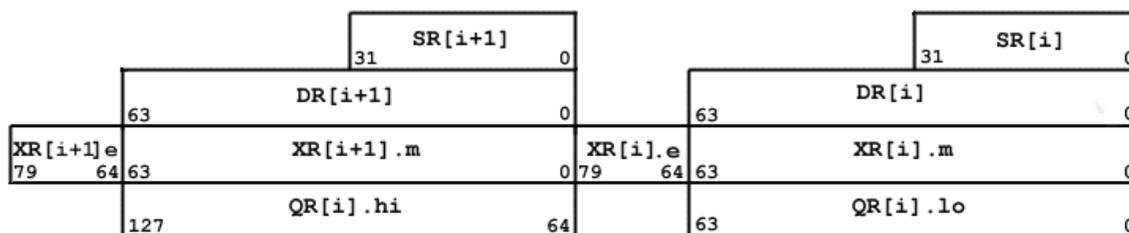


Рис. 2: Правила соответствия вложенности рабочих регистров

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.

9.2.1 Отображение целых типов

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

Таблица 3: 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.

Таблица 4: 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

9.2.2 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.

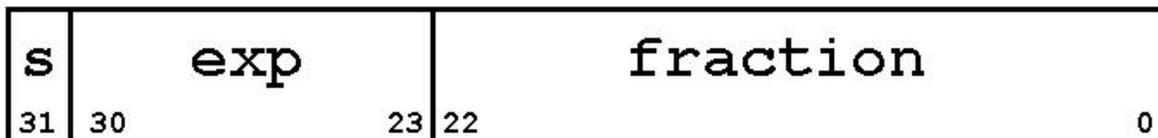


Рис. 3: Простой вещественный формат.

Рисунок D-14. Вещественный формат удвоенной точности.

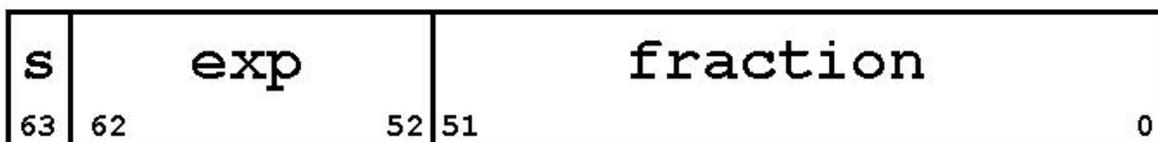


Рис. 4: Вещественный формат удвоенной точности

Рисунок D-15. Расширенный вещественный формат.

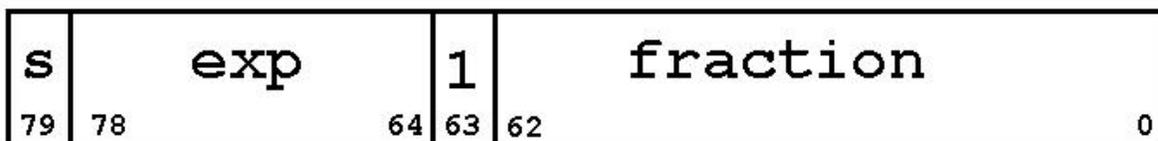


Рис. 5: Расширенный вещественный формат

Отображение вещественных типов для всех режимов приведено в таблице.

Таблица 5: 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

9.2.3 Отображение указательных типов

Отображение указательных типов для режима 32 приведено в таблице.

Таблица 6: 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.

Таблица 7: 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

9.2.4 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 (падинг). Значение полей падинга не определено.

9.2.4.1 Тип массива

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

9.2.4.2 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



Рис. 6: Простая маленькая структура

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

Рисунок D-4. Плотно упакованная структура без паддинга

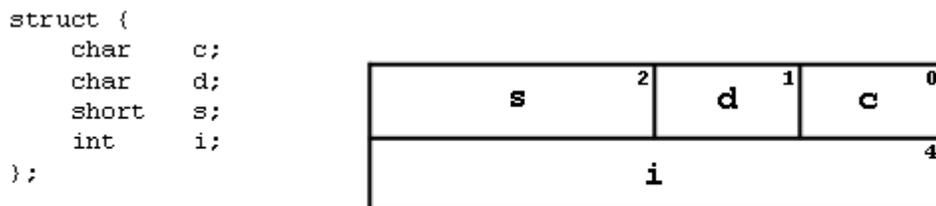


Рис. 7: Плотно упакованная структура без паддинга

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

Рисунок D-5. Структура с внутренним паддингом

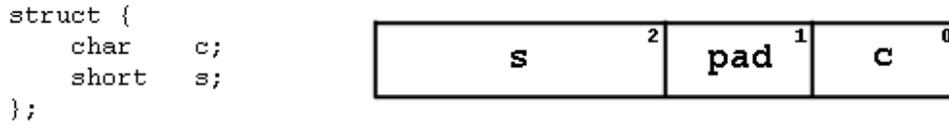


Рис. 8: Структура с внутренним паддингом

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

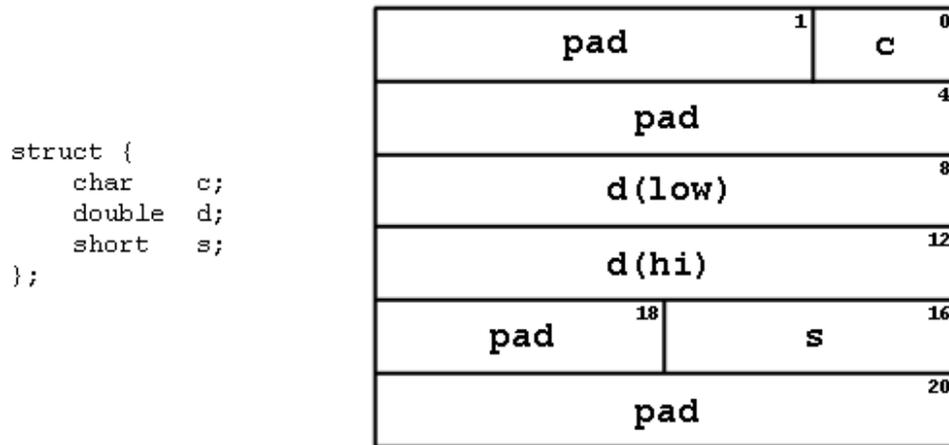


Рис. 9: Структура с внутренним и хвостовым паддингами

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.

9.2.4.3 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

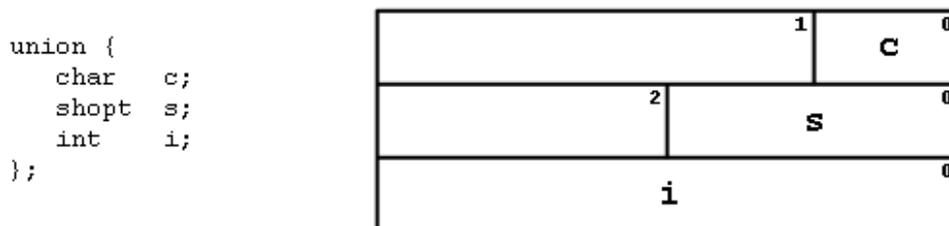


Рис. 10: Объединение

Размер объединения – 4 байта. Выравнивание – 4 байта.

Рисунок D-8. Объединение

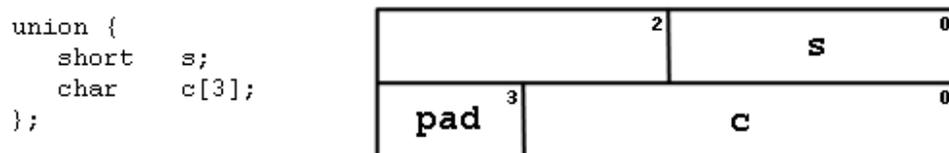


Рис. 11: Объединение с хвостовым паддингом

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.

9.2.4.4 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

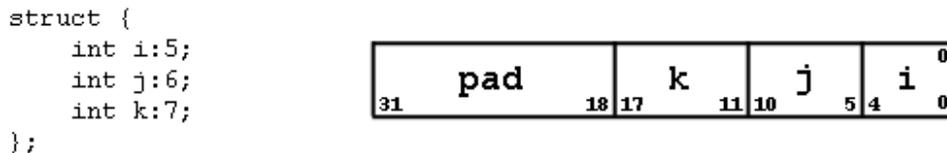


Рис. 12: Структура с битовыми полями

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

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 t 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

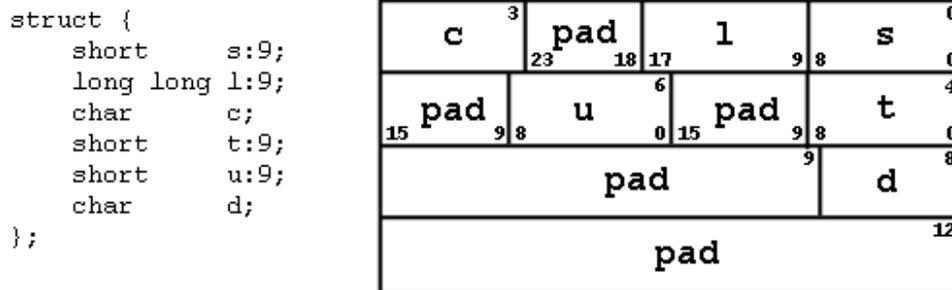


Рис. 13: Выравнивание в структуре с битовыми полями

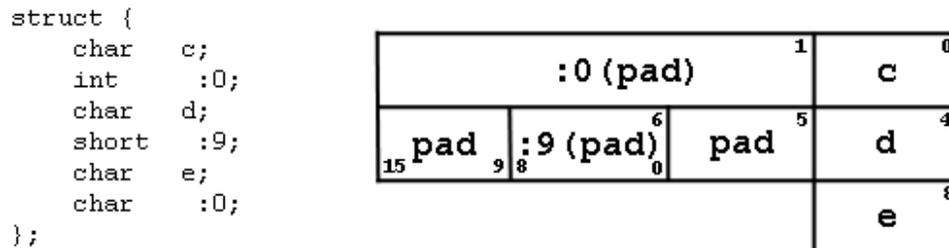


Рис. 14: Структура с неименованными битовыми полями нулевой длины

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

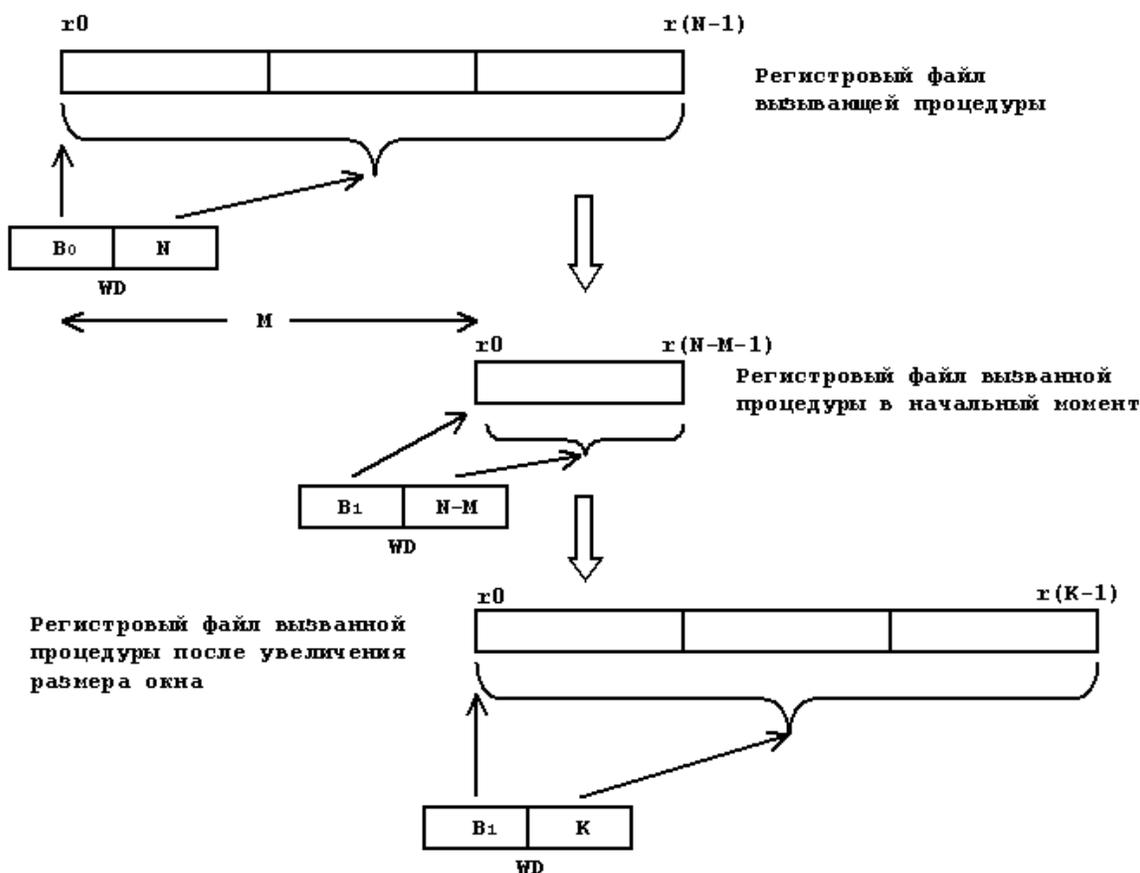


Рис. 15: Процедурное переключение окна

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 регистрационного 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 регистрационного 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 регистрационного окна (номер регистра больше размера, заданного в **WD**) возникает прерывание.

Максимально возможный размер регистрационного окна определяется размером стековой области регистрационного 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 покрывать все регистрационное окно.

Ассемблерная мнемоника обращения к регистрам текущего окна:

<code>%r [number]</code>	- SR регистр
<code>%dr [number]</code>	- DR регистр
<code>%qr [number]</code>	- 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:

<code>%b [number]</code>	- SR регистр
<code>%db [number]</code>	- DR регистр
<code>%qb [number]</code>	- 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.

Таблица 8: 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

continues on next page

Таблица 8 – продолжение с предыдущей страницы

Регистр	Описание	Доступ	Сохранение
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 равно 0. Устанавливается операцией SETTR.	R/-/M	Auto
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
PFPR	Статусный регистр вещественной арифметики упакованных значений. Содержит маски прерываний, накапливаемые флаги и правила вычислений. Накапливаемые флаги могут изменяться операциями вещественной арифметики упакованных значений.	R/W/M	Special
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
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



Рис. 16: 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 quadro 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 Р-1. Размещение списка параметров на физические ресурсы

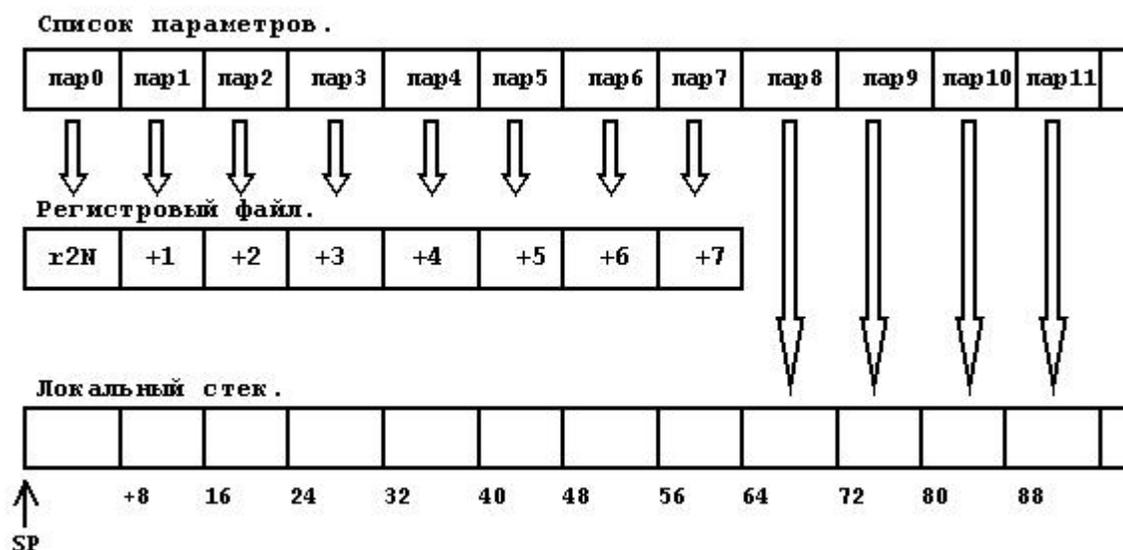


Рис. 17: Размещение списка параметров на физические ресурсы

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 Р-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.

Таблица 9: 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	$(\text{размер} + 7)/8$

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 (падинг).

При размещении целочисленного параметра размера меньше, 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 предписание, то тип возвращаемого значения определяется по правилам языка (обычно `int`). Особо отметим случай вызова процедур без предписания, 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 `ctpl1` (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 `ctpl1`.
- **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 `ctpl1`.

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 `ctprN`, и параметры. Если подготовленный переход имеет тип `ctpdw` или `ctprew`, 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 `IP`);
- 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 `ctprew` (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.

10.1 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 quadro формата	
		в регистровом файле	
		b - операнд или результат является предикатом	

(continues on next page)

(продолжение с предыдущей страницы)

		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.

10.2 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 1: таблица спекулятивного режима

режим/аргументы	допустимые	недопусти- мые	хотя бы один диагностический
обычный	результат	исключение	исключение
спекулятивный	результат	диагности- ческий	диагностический

10.3 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.

10.3.1 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

10.3.2 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, выработывая 32/64-разрядный результат.

Язык ассемблера:

<code>muls</code>	<code>src1, src2, dst</code>
<code>muld</code>	<code>src1, src2, dst</code>
<code>umulx</code>	<code>src1, src2, dst</code>
<code>smulx</code>	<code>src1, src2, dst</code>
<code>umulhd</code>	<code>src1, src2, dst</code>
<code>smulhd</code>	<code>src1, src2, dst</code>

10.3.3 Операции деления и вычисления остатка

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

<code>udivx</code>	<code>src1, src2, dst</code>
<code>umodx</code>	<code>src1, src2, dst</code>
<code>sdivx</code>	<code>src1, src2, dst</code>
<code>smodx</code>	<code>src1, src2, dst</code>
<code>udivs</code>	<code>src1, src2, dst</code>

(continues on next page)

(продолжение с предыдущей страницы)

udivd	src1, src2, dst
sdivs	src1, src2, dst
sdivd	src1, src2, dst

10.3.4 Операции сравнения целых чисел

CMP(s/d)b	группа из 8-ми операций сравнения:
CMP0(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

(continues on next page)

(продолжение с предыдущей страницы)

cmpandpsb	src1, src2, predicate
cmpandlesb	src1, src2, predicate
cmpandedb	src1, src2, predicate
cmpandsdb	src1, src2, predicate
cmpandpdb	src1, src2, predicate
cmpandledb	src1, src2, predicate

10.3.5 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

10.3.6 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

10.3.7 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

10.3.8 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
-----	-----------------

10.3.9 Выбор из двух операндов

MERGEs/d	sss/ddd	выбрать один из операндов 32/64 как результат (требуется 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

10.4 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

(continues on next page)

(продолжение с предыдущей страницы)

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 с формированием результата в виде предиката
FCMPOD(s/d)b	ssb/ddb fp32/fp64	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.

10.4.1 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:

fscals	src1, src2, dst
fscald	src1, src2, dst

10.4.2 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

10.4.3 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)src1[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

(continues on next page)

(продолжение с предыдущей страницы)

fdtois	src2, dst
fdtoid	src2, dst
fstoistr	src2, dst
fdtoistr	src2, dst
fstoidtr	src2, dst
fdtoidtr	src2, dst

10.5 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.

10.5.1 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 Logical Predicate - 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.

10.5.2 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.
```

10.5.3 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>.
```

10.5.4 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>.
```

10.5.5 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>.
```

10.5.6 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
```

10.6 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.

10.6.1 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

10.6.2 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 }

10.6.2.1 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 квадр слова

Язык ассемблера:

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

10.6.2.2 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 }

10.6.3 Операции обращения к массиву

10.6.3.1 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

10.6.3.2 Операции записи в массив

STAAB	pps	запись байта
STAAN	pps	запись полуслова
STAAN	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

10.7 Операции преобразования адресных объектов

10.7.1 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
-------	-----------

10.7.2 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

10.8 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 predetermined-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.

10.8.1 Операции управления регистровым окном

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 `setbn`, `setbp` модифицируют поля регистра BR, `setwd` - поля регистра WD:

- `wsz` - размер окна регистров, в терминах quadro-слов или спаренных регистров (0x8 означает 16 числовых регистров; NB! даже для версий v5+, где размер регистров вырос до 128 бит)
- `nfx` - устанавливается в 1, если в процедуре нет операций FX; исключительно оптимизационное поле, помогает сократить время работы ядра ОС при сохранении/восстановлении контекста
- `dbl` - включает режим расширения записи в регистр назначения на старшую часть для 32-битных операций.
- `psz` - размер вращаемой области в PF; количество PR минус 1; вращаемая область включает PR[0], ... PR[BR.psz] ;
- `rbs` - база вращаемой области NR-ов (в терминах quadro NR) в текущем окне - ;
- `rsz` - размер вращаемой области в RF (в терминах quadro NR) - количество quadro NR минус 1;
- `rcur` - текущее значение числовой базы (в терминах quadro NR);

Операции `setwd/setbn`, как правило, используются в нулевом такте работы каждой процедуры - если требуется расширить размер используемого окна регистров. Операции `setwd/setbn/setbp` также предваряют и завершают конвейеризированные циклы, т.к. для таких циклов требуется задать область вращаемых регистров и предикатов. При ручном программировании на ассемблере: каких-либо ограничений по модификации регистрового окна в течение времени работы процедуры нет. Более подробно о программных соглашениях по работе с регистровым окном - см раздел *Механизм регистровых окон*

10.9 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 - СТП);

- фактическая передача управления (control transfer - CT).

Операции **СТР** 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 **СТР** содержат всю или часть информации о передаче управления (тип, адрес, etc); эта информация сохраняется до операции **СТ** на одном из регистров **СТPRj** 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**;
 - тэгируемая 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.

10.9.1 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.

```

10.9.2 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.

```

10.9.3 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 (MOVTD)*; 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.

```

10.9.4 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.
```

10.9.5 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;
```

10.9.6 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.
```

10.10 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, выполняется 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 }
```

10.10.1 Подготовленный переход (BRANCH)

BRANCH выполняет непроцедурную подготовленную передачу управления.

BRANCH выполняется последовательностью двух операций:

- СТ подготовка в `ctp_reg`, **СТР**;
- фактическая **СТ** операция из этого `ctp_reg`.

Язык ассемблера для СТ 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 **СТ**:

```
ct           ctp_reg { control_condition }
```

где:

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

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

10.10.2 Непосредственный переход (IBRANCH)

IBRANCH выполняет непосредственную непроцедурную передачу управления.

Язык ассемблера:

```
ibbranch     label { control_condition }
```

Вариант операции:

```
ibbranch     label ? %MLOCK
```

является синонимом:

```
rbranch     label
```

10.10.3 Операция CALL

CALL выполняет процедурную подготовленную передачу управления.

CALL выполняется последовательностью двух операций:

- СТ подготовка в `ctp_reg`, **СТР**;
- фактическая **СТ** операция из этого `ctp_reg`.

Язык ассемблера для СТ 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 **СТ**:

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 **СТР** и **СТ** в программе непосредственно одну за другой, они могут быть размещены на любом расстоянии друг от друга, при условии, что ctp_reg не используется повторно.

10.10.4 Возврат из аппаратного обработчика прерываний (DONE)

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

Язык ассемблера:

done	{ control_condition }
------	-----------------------

10.11 Операции поддержки наложений цикла

10.11.1 Операции Set BR

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

10.11.2 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).

10.11.3 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, закодированного 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).

10.11.4 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.

10.11.5 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.

10.11.6 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

$$\text{area_size} = (64 \text{ байта}) * (2^{**} \text{asz}).$$

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 AAD_j;

ind ссылка на значение начального индекса (`init_index`), то есть номер `j` регистра AAIND_j;

```
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 AAINCR_j; 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.

10.11.7 Начать предподкачку массива (BAP)

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

Язык ассемблера:

```
bap
```

10.11.8 Остановить предподкачку массива (EAP)

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

Язык ассемблера:

```
eap
```

10.11.9 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

10.12 Разные операции

10.12.1 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
```

10.12.2 Операция вставить пустые такты (BUBBLE)

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

Язык ассемблера:

```
por          { NUM }
```

10.12.3 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.

10.12.4 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.

10.12.5 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
```

10.12.6 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

Режим Безопасных Вычислений

В этом дополнении будет кратко описан особый режим компиляции и исполнения программ на C/C++, называемый Режимом Безопасных Вычислений (РБВ).

11.1 Назначение и общий принцип работы РБВ

Идея использования РБВ заключается во введении дополнительного контроля поведения программ на C/C++, отсутствующего в обычных реализациях этих языков. Контроль опирается на специализированные аппаратные функции платформы Эльбрус. Его ключевые особенности:

- контролируются обращения за границами объектов,
- контролируется использование неинициализированных переменных,
- контролируется создание новых указателей и конверсия указателей.

Дополнительный контроль позволяет обнаруживать сложные ошибки исполнения программ, связанные с переполнением буфера, использованием случайных данных. Более того, в РБВ становится невозможной эксплуатация многих программных уязвимостей, которые не были обнаружены до начала эксплуатации.

Для работы программы в РБВ вводятся дополнительные требования к её исходным текстам. Они более жёсткие, чем общие стандарты языков C/C++.

РБВ основан на аппаратной поддержке контроля ссылок/указателей, типов данных и контекста исполнения, с точностью до модуля. Аппаратный контроль (технология безопасных вычислений, ТБВ) является более глубоким и общим свойством архитектуры Эльбрус, и может быть задействован в реализации разных языков программирования. На данный момент времени он используется только для языков C/C++, и в руководстве мы ограничимся его описанием в данной части.

11.2 Устройство указателя в РБВ

В режиме РБВ указатели расширяются до дескрипторов. В отличие от обычного указателя C/C++, который совместим с целочисленным типом, дескриптор РБВ содержит в себе дополнительную информацию:

- указатель на начало выделенной области памяти/объекта (64 бита);
- размер выделенной памяти/объекта (32 бита);
- смещение относительно начала (32 бита).

Дескриптор в ТБВ имеет размер 128 бит, выровнен по размеру и подтверждён аппаратными тэгами. В рамках ТБВ его можно получить ограниченным числом способов:

- получить от ядра операционной системы с помощью системного вызова (malloc);
- получить указатель на процедурный фрейм пользовательского стека;
- сконвертировать из другого указателя, с уменьшением области памяти (переход от объекта к подобъекту);
- скопировать из указателя на область глобалов модуля.

11.3 Тэги

Аппаратные тэги - добавочные биты информации, которые хранятся отдельно от основных данных и используются для разметки дополнительных свойств ТБВ. Для размещения тэгов «рядом» с данными в оперативной памяти используется часть битов ЕСС.

Тэги позволяют сохранять в процессе исполнения информацию о типах данных и в дальнейшем контролировать их аппаратно. Для РБВ используются типы:

- числовые данные;
- указатель (дескриптор);
- неинициализированные данные.

Наличие тэгов учитывается семантикой всех операций из системы команд, но в руководстве мы ограничимся лишь самыми общими правилами:

- целостность дескриптора сохраняется лишь в том случае, когда дескриптор записывается целиком (атомарно), в пару соседних регистров (квадрорегистр) либо в выровненные 16 байт памяти;
- операции чтения и записи проверяют целостность тэгов аргумента-дескриптора, и вырабатывают исключение/диагностику, если целостность тэга нарушена.

Таким образом, с помощью тэгов гарантируется одно из главных свойств РБВ:

- работоспособный дескриптор нельзя получить из частей разных дескрипторов и/или числовых данных, ни случайно, ни специально.

11.4 Дополнительные ограничения на исходные коды

Для компиляции в РБВ программа на C/C++:

- не должна содержать преобразование из целого в указатель;
- не должна закладываться на `sizeof(void*) == sizeof(long)`;
- не должна закладываться на `sizeof(void*) == sizeof(double)`;
- не может содержать отдельных объектов размером более 2^{32} - это ограничение является временным для текущей реализации.

В процессе исполнения в РБВ программа получит исключительную ситуацию:

- при обращении по указателю за пределами, указанными в дескрипторе `[base, base+size]`;
- при использовании данных, прочтенных из неинициализированной памяти;
- при попытке разыменования некорректного дескриптора (попытка неявного преобразования целого в указатель через память).

Полный список ограничений достаточно велик и выходит за рамки документа. Заметим, что довольно часто нарушения дополнительных ограничений происходит в собственных менеджерах памяти программ.

11.5 Частичные аналоги РБВ

Идеи контроля границ объектов были реализованы в отечественной и мировой практике как программными средствами (`valgrind`, `address sanitizer = asan`), так и на уровне аппаратуры (`cheri`). В некоторых ситуациях РБВ работает точнее своих аналогов, обнаруживая нарушения границ на любое смещение, в отличие от, например, `asan`.

11.6 Пример сборки и исполнения теста

```
#include <stdlib.h>
#define N 1000

int *gp;
int main()
{
    gp = (int*)malloc(N*sizeof(int));
    gp[N/2] = N/2;
    gp[-1] = 14; // ошибка
    return 0;
}
```

```
$ /opt/mcst/bin/lcc ./t.c -o t
$ ./t
$ /opt/mcst/bin/lcc -mptr128 ./t.c -o t_128
$ ./t_128
Segmentation fault
```

Как видно из примера, версия теста в РБВ упала на исполнении с ошибкой сегментации, в то время как в обычном режиме тест отработал без ошибки, несмотря на запись за пределами выделенного объекта.

11.7 Ассемблер теста в РБВ

Посмотрим на ассемблер функции main теста из предыдущего раздела.

```
main:
{
  nop 1
  setwd wsz = 0x13, nfx = 0x1, dbl = 0x0
  setbn rsz = 0x7, rbs = 0xb, rcur = 0x0
  setbp psz = 0x0
  getsap,0      _f32s,_lts1 0xffffffffe0, %r18
}
{
  nop 1
  stapb,2      %r18, 0x0, %r18
}
{
  disp %ctpr1, malloc; ipd 2
  movtq,0,sm   %r18, %b[0]
  addd,2      0x0, _f16s,_lts0lo 0xfa0, %b[2]
  adds,3      0x0, _f16s,_lts0hi 0x1f4, %r1
  adds,4      0xe, 0x0, %r2
}
{
  nop 3
  aptoapb,0   %r18, _f16s,_lts0lo 0x20, %b[0]
}
{
  call %ctpr1, wbs = 0xb
}
{
  return      %ctpr3; ipd 2
  stgdq,2,sm  0x0, [ _f32s,_lts0 gp ], %b[0]
  adds,3      0x0, 0x0, %r0
}
{
  stapw,5     %b[0], _f16s,_lts0lo 0x7d0, %r1
}
{
  nop 2
  ldgdq,0     0x0, [ _f32s,_lts0 gp ], %r20
}
{
  stapw,2     %r20, _f16s,_lts0lo 0xffffc, %r2
}
{
  ct %ctpr3
}
}
```

Отличия от обычного режима исполнения:

- получение указателя на фрейм стека: `getsap` вместо `getsp`; результатом является 128-битный дескриптор.
- команда `stapw` вместо `stw`: обращение в память по дескриптору. В качестве аргумента должен быть 128-битный регистр, реализованный в качестве пары соседних 64-битных регистров, в котором находится дескриптор.
- `movtq` копирует 128-битный дескриптор с сохранением тэгов.
- команда преобразования указателя `aptoarb`: возвращает дескриптор, полученный из дескриптора `%r18-%r19` в первом аргументе, до размера `0x20`, указанного во втором аргументе.
- команды `stgd` и `ldgd` нужны для обращения к глобальным объектам. В качестве дескриптора области глобальных объектов выступает регистр `%gd`.

Планирование последовательности операций

12.1 Зависимости по передаче данных и управления

Содержащиеся в данном параграфе таблицы построены следующим образом - по вертикали перечисляются операции, являющиеся источником зависимости. Им соответствуют строки таблиц. По горизонтали перечисляются зависимые операции. Им соответствуют столбцы таблиц.

В таблицах указано значение *relative_latency* - относительное время исполнения операции, поставляющей данные; с точки зрения планирования это - минимально необходимое расстояние в тактах/командах между одноименными фазами зависимых операций;

Корректность некоторых информационных передач поддержана аппаратно (о чем сказано рядом с таблицей), и тогда нарушение минимального расстояния между операциями вызывает ту или иную блокировку конвейера. Это чревато потерей производительности, но не может привести к нарушению корректности вычислений.

Задержка между источником и потребителем может быть реализована размещением между ними реальных или фиктивных команд (операция *nop* N , где $N > 0$), либо их смеси. Если требуемая задержка (указанная в таблице) равна D , между источником и потребителем должно присутствовать минимум $(D-1)$ реальных или фиктивных команд. В некоторых, специально оговоренных случаях фиктивные команды не годятся, и нужно пользоваться только реальными (разрешается операция *por* с нулевым аргументом).

12.1.1 Передача данных через регистровый файл

12.1.1.1 Общие положения

Все операции, получающие аргументы через регистровый файл, считывают их в своей фазе R(базирование адресов операндов и передача их на регистры регистрового файла) или E0(начало выполнения операции в исполнительном устройстве). Результаты операций могут передаваться потребителю следующими путями:

- через регистровый файл (*rf*)

- через входные регистры rf (rfbps)
- через байпас группы целочисленных операций (ibps)
- через байпас группы вещественных и мультимедийных операций (fbps)
- через байпас записываемого значения операций записи в память (sbps).

Байпас -это аппаратный механизм, позволяющий операции потребителю получить результат до его записи в регистровый файл (напрямую от операции источника)

Микропроцессор «Эльбрус» имеет 6 каналов для выполнения арифметико-логических операций, при этом каналы 0,1,2 относятся к одному кластеру, а каналы 3,4,5 относятся к другому кластеру. Байпас не может работать между кластерами, работая в каждом отдельно.

Группа, использующая байпас ibps, включает:

- скалярные целочисленные операции, исключая операции умножения, деления, получения остатка и извлечения корня
- операции считывания их памяти
- адресные операнды операций записи в память.

Группа, использующая байпас fbps, включает:

- все вещественные операции - скалярные и пакетные
- все операции преобразования вещественных и целых чисел - скалярные и пакетные
- целочисленные операции умножения
- целочисленные операции деления, получения остатка и извлечения корня
- все пакетные целочисленные операции.

Группа, использующая байпас sbps, включает:

- записываемое значение операций записи в память.

Байпасирование не применяется или ограничивается в следующих случаях:

- результаты операций считывания из памяти в случае промаха в Data Cache первого уровня (DC) передаются только через rf;
- результаты операций пересылки данных из APB передаются только через rf;
- если формат результата-источника (с учетом режима расширения формата флагом WD.dbf) меньше формата операнда-потребителя, передача выполняется через rfbps или rf; операнд-значение, записываемое в память, может передаваться только через rf;
- результаты операций группы fbps передаются в операции группы ibps только через rfbps или rf;
- результаты операций группы ibps передаются в операции группы fbps с задержкой в 1 такт (через rf, rfbps или ibps);
- второй операнд тех расширенных вещественных операций (FX...), которые требуют его преобразования в расширенный формат (s/d -> x), не передается через байпас fbps; к таким операциям относятся $FXADD(s/d)(s/d/x)$, $FXSUB(s/d)(s/d/x)$, $FXRSUB(s/d)(s/d/x)$, $FXCMP(s/d)(f/b)$; передача осуществляется через rfbps или rf.

В случае суперпозиции условий выбирается «наихудший» вид передачи.

Кроме того, байпас fbps не применяется при определенных сочетаниях типов/форматов результатов и операндов зависимых операций этой группы (эти сочетания считаются статистически незначимыми); передача осуществляется через rfbps или rf. В приведенной ниже таблице по вертикали перечислены типы/форматы результата операции-источника, а по горизонтали - типы/форматы операндов

операции-потребителя (сочетания, не упомянутые в таблице, подчиняются общим правилам). В таблице использованы следующие обозначения:

+ - байпас fbps применяется;

f80 - вещественные числа формата extended, а также результаты и операнды других форматов, если они вырабатываются/потребляются операциями над вещественными числами формата extended;

f64 - вещественные числа формата double;

f32 - вещественные числа формата single;

pf32 - вещественные числа формата single, упакованные в данных формата double;

mmx64 - целые числа, упакованные в данных формата double;

mmx32 - целые числа, упакованные в данных формата single;

fcm64 - результаты операций вещественного сравнения формата double;

fcm32 - результаты операций вещественного сравнения формата single;

int64 - целые числа формата double, являющиеся результатами/операндами операций умножения, деления и преобразования целых чисел;

int32 - целые числа формата single, являющиеся результатами/операндами операций умножения, деления и преобразования целых чисел/

Таблица байпассирования результатов для операций группы fbps представлено в таблице 12.1

Таблица 12.1

ист	п	о	т	п	п	п	п	п	п
ист	f80	f64	f32	pf32	mmx64	mmx32	int64	int32	
f80	+				+	+	+	+	
f64		+			+				
f32			+			+			
pf32				+	+				
mmx64		+		+	+	+	+	+	
mmx32			+			+		+	
fcm64		+		+	+	+	+	+	
fcm32			+			+		+	
int64					+	+	+	+	
int32						+		+	

12.1.1.2 Времена исполнения операций

С точки зрения времени исполнения операции, поставляющие результат в *gf*, подразделяются на несколько групп (такое разбиение используется в таблице 12.2):

out_0i:

в случае числового результата:

ADDs/d, SUBs/d, AND(N)s/d, OR(N)s/d, XOR(N)s/d, SCLs/d, SHLs/d, SHRs/d, SARs/d, MERGEs/d, SXT, GETF, INSF

GETTAGs/d/q, PUTTAGs/d, BITREVs/d, LZCNTs/d, POPCNTs/d

out_1i:

Целочисленные комбинированные(трех-операндные) операции.

AND, ANDN, OR, ORN, XOR, XORN, MERGE, ADD, SUB, SCL, SCR, SHL, SHR, SAR, GETF, INSF

out_1a:

в случае выработки флагов:

ADD, SUB*, AND(N)*, OR(N)*, XOR(N)*, SCL*, SHL*, SHR*, SAR*, INC*, DEC**

RRs/d (кроме *FPCR, FPSR, PFPFR*)

GDTOAP, CUDTOAP, GETVA, GETPL, CAST, APTOAP(B), ODTOAP, MOVTs/d/q, TDTOMP, GETTD, GETSP, GETSAP, GETSOD

GETTST

out_2i:

в случае попадания в DC1:

*LD**, за исключением *LDAA**

out_4i:

MOVAB, MOVAH, MOVAW, MOVAD, MOVAQ

out_1m:

PADD, PSUB*, PCMP*, PAVG*, PMIN*, PMAX*, PUNPCK*, PACK*, PMOV**

PSR, PSL*, PSHUF*, PEXTRH, PINSH, PMERGE*

PAND, POR*, PXOR*, PSIGN**

PHADD, PHSUB**

out_2m:

PMULHH, PMULLH, PMADDH, PMULUBHH, PMULHRSH, PMADDUBSH

PSADBW, MPSADBH, PHMINPOSUH

FCMP(s/d)f, FCMP(s/d), PFCMP(s/d)

FMIN(s/d), FMAX(s/d), PFMIN(s/d), PFMAX(s/d)

out_4f:

в случае денормализованного операнда или отрицательного переполнения произведения - *out_5f*

FMULs/d, PFMULs/d, FXMULsx, FXMULdx, FXMULxx, FSCALEs/d, FXSCALEsx

MULs/d, UMULX, SMULX, UMULHd, SMULHd

FADDs/d, FSUBs/d, PFADDs/d, PFSUBs/d, PFHADDs, PFHSUBs, PFADDSUBs

операции преобразования *STO*S, STO*D, DTO*S, DTO*D*

MOVFI, MOVIF, MOVx

FXCMP(s/d/x)f

out_5f:

в случае денормализованного операнда или отрицательного переполнения произведения - out_6f

FXMULss, FXMULdd, FXMULxs, FXMULxd

FXADD..(7 операций), FXSUB..(7 операций), FXRSUB..(4 операции)

операции преобразования *STO*X, XTO*S, DTO*X, XTO*D, STO*XTO*S, DTO*XTO*D*

out_6f:

FRCPs, FRSQRTs

в случае NaN - out_13f

out_8r:

в случае попадания в DC2 (см. также out_19-r):

*LD**,

для операций типа «TLB probe»

out_8f:

FMUL_(ADD/SUB/RSUB)(s/d)

PFMUL_(ADD/SUB/RSUB)(s/d)

PFMUL_(HADD/HSUB/ADDSUB)s

в случае, когда постепенное отрицательное переполнение или денормализованный операнд вызывают необходимость денормализации или нормализации произведения (в операциях умножения), - тогда это относится к out_9f

F(ADD/SUB)_(ADD/SUB/RSUB)(s/d)

PF(ADD/SUB)_(ADD/SUB/RSUB)(s/d)

PF(HADD/HSUB/ADDSUB)_(HADD/HSUB/ADDSUB)s

out_9f:

RRs/d (FPCR, FPSR, PFPFR)

out_10f:

FSQRTs, PFSQRTs

Эти операции могут относиться к другим группам в некоторых редких (менее 0.5%) случаях - out_12f; в случае NaN - out_13f.

out_11f:

FDIVs, PFDIVs (в случае NaN - out_13f)

FXSQRTIsx, FXSQRTIdx, FXSQRTIxx (в случае NaN - out_18f)

FSQRTId

out_10-14f:

SMODx, UMODx

out_11-14f:

SDIVs/x, UDIVs/x

out_11-16f:

FXDIVss, FXDIVsx, FXDIVdx, FXDIVxs, FXDIVxx

FXDIVTss, FXDIVTsx, FXDIVTdx

в случае NaN - out_18f

out_11-18f:

SDIVd, UDIVd

out_14f:

FDIVd, PFDIVd

в случае NaN - out_16f

FSQRTTd, PFSQRTTd

в некоторых редких (менее 0.5%) случаях - out_31f; в случае NaN - out_16f;

out_14-16f:

FXDIVdd, FXDIVxd, FXDIVTdd

в случае NaN - out_18f

out_16f:

FXSQRTU..(3 операции), FXSQRTT..(3 операции)

в случае NaN - out_18f

out_19-r:

*LD** (в случае промаха в DC2)

С точки зрения получения операндов через байпасы различных типов операции также подразделяются на несколько групп:

in_i:

ADDs/d, SUBs/d, AND(N)s/d, OR(N)s/d, XOR(N)s/d

SHLs/d, SHRd, SARs/d

*LD**

адресные операнды (src1,src2) операций *ST**

in_f:

все FP-операции

все операции умножения

все операции деления

все мультимедийные операции

in_r:

все операции из другого кластера (за исключением операнда - значения для записи в память)

in_s:

операнд (src3) - значение для записи в память (зависит от кластера):

*ST**

12.1.1.3 Зависимости между операциями

В Таблице 12.2 приведены рекомендации по планированию операций для случая зависимости типа запись - считывание. Таблица содержит минимально возможные времена передачи операндов, то есть без учета динамически возникающих конфликтов.

В колонках 2-4 указаны времена передач с использованием всех доступных байпасов. Для многих операций такие пути возможны только при условии, что операции источник и потребитель расположены в одном кластере.

В последних двух колонках указаны времена передач только через rfbps или rf. Как правило, это передачи между кластерами, либо передачи, для которых не реализованы более короткие пути. Дополнительно нужно учитывать следующие особенности:

- Для передач в другой кластер из операций-источников out_0i предусмотрен сокращенный путь, отображенный в строке «через байпас»,
- Цифры предпоследнего столбца (in_s) нужно корректировать, если результат операции-источника имеет меньший формат, чем записываемое значение, - задержка увеличивается на 1 (передача только через rf); исключения составляют цифры, помеченные знаком „*“, так как они уже соответствуют передаче только через rf; также не требует коррекции цифра в строке «out_0i через байпас».

Таблица 12.2

first op (modifying)	second op (reading)				
	в своем кластере (через байпасы)			в другой кластер или через rfbps, rf (см. 12.1.1.1)	
	in_i	in_f	in_s	in_s	in_r
: через байпас	1	1	1	1	2
out_0i					
через rfbps или rf				3	4
out_1i	2	2	1	3	4

(continues on next page)

(продолжение с предыдущей страницы)

out_1a	2	2	1	3	4	
out_2i	3	3	4	4	4	
out_4i	5	5	4	4 *)	5	
out_1m	4	1	1	3	4	
out_2m	4	2	2	3	4	
out_4f	6	4	4	6	6	
out_5f	7	5	5	7	7	
out_6f	8	6	6	8	8	
out_8r	9	9	8	8 *)	9	
out_8f	10	8	8	10	10	
out_9f	11	9	9	11	11	
out_10f	12	10	10	12	12	
out_10-14f	12-16	10-14	10-14	12-16	12-16	
out_11f	13	11	11	13	13	
out_11-14f	13-16	11-14	11-14	13-16	13-16	
out_11-16f	13-18	11-16	11-16	13-18	13-18	
out_11-18f	13-20	11-18	11-18	13-20	13-20	
out_12f	14	12	12	14	14	
out_13f	15	13	13	15	15	
out_14f	16	14	14	16	16	
out_16f	18	16	16	18	18	
out_14-16f	16-18	14-16	14-16	16-18	16-18	
out_18f	20	18	18	20	20	
out_31f	33	31	31	33	33	
out_19-r	20-...	20-...	19-...	19-...*)	20-...	

Корректность передачи данных через регистровый файл контролируется аппаратно, посредством scoreboard'а. При попытке считывания неготовых данных вырабатывается блокировка конвейера.

Если источником являются операции считывания из памяти (или специальные операции, кодирующиеся как LDxxxx), блокировка вырабатывается на четное количество тактов. В остальных случаях блокировка вырабатывается на количество тактов, кратное четырем. Таким образом, возможна потеря либо одного, либо от одного до трех тактов в случае, когда истинное условие блокировки (неготовность данных) длится некратное количество тактов.

Зависимость типа считывание - запись, когда первая операция должна считать старое значение прежде, чем вторая операция запишет в ту же ячейку новое, отсутствует. Операция, модифицирующая ячейку rf, может стоять в той же команде, что и операция, считывающая старое значение из этой ячейки.

Проблема зависимости типа запись - запись, когда первая операция должна завершиться раньше второй, решается аппаратно. Операции могут стоять в соседних командах, тем не менее в rf будет записан результат второй операции.

12.1.2 Передача предикатов через предикатный файл

В Таблице 12.3а приведены ограничения на планирование операций для случая зависимости типа запись - считывание.

Таблица 12.3а

first op (modifying)	second op (reading)			
	CT (cond)	RLP MRGC	ELP	CT(CALL) (pf save)
CMPANDs/d	3 *1)	2 *2)	1	0
CMPs/d				
CCTOLP				
CLP				
MLP				
CMPANDs/d *4)	4			
CMPs/d				
CCTOLP				
FCMPs/d	5 *1)	4 *2)	3 *2)	0 *3)
FCMPes/d				
CT(RETURN) (pf restore)	1	1	1	1

ct(cond) - при такой операции ct происходит потребление предиката из под условия

ct(call) - при такой операции ct происходит аппаратный спил, сохранение регистров

ct(return) -при такой операции ct происходит восстановление регистров из связующего стека

* 1) - передачи контролируются аппаратурой; нарушение указанных соотношений приводит к блокировке конвейера на недостающее количество тактов;

* 2) - передачи контролируются аппаратурой; нарушение указанных соотношений приводит к блокировке конвейера на четное количество тактов;

* 3) - если операции расположены в одной и той же или в соседних командах, возможна блокировка конвейера на 2 такта для цели перехода;

* 4) - особым случаем является следующая исполняемая последовательность команд (с информационной зависимостью по результатам/операндам):

- 1 считывание из памяти с проверкой DAM (MAS - «check» или «check & unlock»);
- 2 следующая команда содержит операцию сравнения (или CCTOLP), потребляющую результат считывания; при этом не имеет значения, содержит ли команда (1) операцию {пор N} или нет;
- 3 произвольным образом реализованная пауза в 2 команды для передачи предиката в операцию (4);
- 4 условная операция передачи управления (за исключением gbranch), потребляющая результат сравнения (2);

Эта последовательность может работать некорректно в случае, когда проверка DAM операцией (1) закончилась «звонком». Проблема заключается в том, что в текущих реализациях процессора операция (4) выполняется раньше, чем «check» (1) проверит DAM, и даже в случае «звонка» она перевыполнена быть не может. Этот случай не распознается и не контролируется аппаратурой.

Следует иметь в виду, что приведенная последовательность является последовательностью «по исполнению», то есть не обязательно принадлежит одному базовому блоку кода, а может быть составлена из команд, динамически связанных передачами управления.

Общим решением проблемы является удлинение паузы (3) до 3-х команд.

Существуют также частные обстоятельства, когда особый случай не образуется, и специальных мер для соблюдения корректности принимать не требуется:

- Если команда (1) содержит операцию gbranch, особый случай не образуется.

В Таблице 12.3b приведены ограничения на планирование операций для случая зависимости типа считывание - запись. В этом случае первая операция должна считать старое значение прежде, чем вторая операция запишет в ту же ячейку новое.

Таблица 12.3b

first op	second op (modifying)		
(reading)	CMPANDs/d	FCMPEs/d	CT(RETURN)
	CMPs/d	FCMPs/d	(pf restore)
	CCTOLP		
	CLP		
	MLP		
CT(cond)	0	0	0 *1)
PUTTSD, INVTS			
RLP	0	0	0 *1)
MRGC			
ELP	0	0	0 *1)
CT(CALL)	1 *1)	1 *1)	1 *1)
(pf save)			

* 1)- корректность передач для данных величин задержек гарантируется аппаратурой;

В Таблице 12.3с приведены ограничения на планирование операций для случая зависимости типа запись - запись. Такая зависимость возникает, когда двум последовательным операциям назначается под результат одна и та же ячейка файла. В этом случае первая операция должна завершиться раньше второй.

Таблица 12.3с

first op (modifying)	second op (modifying)		
	CMPANDs/d CMPs/d CCTOLP CLP MLP	FCMPEs/d FCMPs/d	CT(RETURN) (pf restore)
CMPANDs/d	1	1	0 *2)
CMPs/d			
CCTOLP			
CLP			
MLP			
FCMPEs/d	3 *3)	1	0 *2)
FCMPs/d			
CT(RETURN) (pf restore)	1 *1)	1 *1)	1 *1)

* 1)- корректность передач для данных величин задержек гарантируется аппаратурой;

* 2)- проблема решается аппаратно, не накладывая ограничений на планирование: операции могут стоять как в соседних командах, так и в той же команде, тем не менее в pf будет записан результат второй операции. Следует, однако, иметь в виду, что данный алгоритм предназначен только для тех случаев, когда результат первой операции не используется. Если последовательность операций имеет вид запись - чтение - запись, следует пользоваться таблицами 12.3а),b).

* 3)- передачи контролируются аппаратурой; нарушение указанных соотношений может привести к блокировке конвейера на четное количество тактов;

Корректность передач предикатов через предикатный файл не контролируется аппаратурой, поэтому нарушение временных соотношений таблиц 12.3а,b,c может привести к непредсказуемым результатам.

12.1.3 Передача атрибутов СТ-операций через регистры СТРР

В этом разделе используются следующие обозначения:

СТ_проц - процедурные переходы (вход или возврат);

СТ(код) - отражает зависимость по готовности подготавливаемого кода;

СТ(ctpr) - отражает зависимость по заполнению регистров СТРР; имеет смысл в следующих случаях:

- команда передачи управления (СТ) ожидает заполнения СТРР даже при ложном условии перехода («not taken»);

- выработка прерываний по некорректной подготовке ожидает заполнения СТР;

В Таблице 12.4а приведены ограничения на планирование операций для случая зависимости типа запись - считывание.

Таблица 12.4а

first op (modifying)	second op (reading)			
	CT (код)	CT (ctpr)	DONE (ctpr)	RRs/d
DISP	5	2 *1)		1
LDISP				
SDISP	*1)			
GETTSD				
GETPL	9*1)	5 *1)		4
MOVTD				
GETTC				
RETURN	6/8 *1,2)	2/4 *1,2)		6
CT_proc		1		1
RWs/d		5 *3)	5	4

* 1) - корректность модификации регистров СТР контролируется аппаратурой; нарушение указанных временных соотношений вызывает блокировку конвейера, которая длится недостающее количество тактов.

* 2) - задержка увеличивается на 2 такта, если подготовка возврата стоит после операции call, то есть выполняется сразу после выхода из процедуры; если подготовку возврата отодвигать вправо, время выполнения пошагово сокращается, пока не достигнет стандартного. Если после выхода из процедуры (выполнения операции ct) код оказался не готов в течение N тактов, эта задержка эквивалентна отодвиганию операции return на N команд и, соответственно, сокращению времени ее выполнения.

* 3) - данный случай описывает зависимость по выработке прерываний.

Корректность непомяченных передач не контролируется аппаратурой. Нарушение указанных соотношений может привести к непредсказуемым результатам.

В таблице 12.4b приведены ограничения на планирование операций для случая зависимости типа считывание - запись. В этом случае первая операция должна считать старое значение прежде, чем вторая операция запишет в СТР новое.

Таблица 12.4b

first op (reading)	second op (modifying)					
	DISP	GETPL	GETTC	RETURN	CT_proc	RWs/d
	LDISP	MOVTD				

(continues on next page)

(продолжение с предыдущей страницы)

	SDISP						
	GETTSD						
CT (ctpr)	1 *1)	1 *1)	1 *1)	1 *1)	1	1 *1)	
RRs/d	0	0	0	0	0	0	

* 1) - в связи с особенностями аппаратной реализации действует соглашение, запрещающее ставить в одной команде переход и подготовку следующего перехода по одному и тому же CTPR.

Корректность передач не контролируется аппаратурой. Нарушение указанных соотношений может привести к непредсказуемым результатам.

В таблице 12.4с приведены ограничения на планирование операций для случая зависимости типа запись - запись. В этом случае первая операция должна завершиться раньше второй.

Таблица 12.4.с

first op (modifying)	second op (modifying)					
	DISP	GETPL	GETTC	RETURN	CT_proc	RWs/d
DISP	1 *1)	1 *1)	1 *1)	1 *1)	1 *1)	1 *1)
LDISP						
SDISP						
GETTSD						
GETPL	0 *1)	1 *1)	1 *1)	0 *1)	1 *1)	1
MOVTd						
GETTC	0 *1)	1 *1)	1 *1)	0 *1)	1 *1)	3
RETURN	1 *1)	1 *1)	1 *1)	1 *1)	1 *1)	3
CT_proc	1	1 *2)	1 *2)	1	1	1
RWs/d	4	4	4	4	4	1

* 1) - корректность передач для данных величин задержек гарантируется аппаратурой;

* 2) - в связи с особенностями аппаратной реализации действует соглашение, запрещающее ставить в одной команде переход и подготовку следующего перехода по одному и тому же CTPR.

Корректность передач не контролируется аппаратурой. Нарушение указанных соотношений может привести к непредсказуемым результатам.

СИМВОЛЫ

- C
command line option, 8
- D/-U
command line option, 8
- E
command line option, 8
- H
command line option, 8
- I <dir> (-I<dir>)
command line option, 8
- L<dir> (-L <dir>)
command line option, 8
- O0
command line option, 5
- O1
command line option, 5
- O2
command line option, 5
- O3
command line option, 5
- O4
command line option, 5
- S
command line option, 7
- c
command line option, 7
- fPIC (-fpic)
command line option, 7
- fPIE (-fpie)
command line option, 7
- fprofile-generate[=<path>]
command line option, 6
- fprofile-use[=<file>]
command line option, 6
- fwhole
command line option, 6
- fwhole-shared
command line option, 6
- l<name> (-l <name>)
command line option, 8
- m128
command line option, 7
- m32
command line option, 7
- m64
command line option, 7
- o<file> (-o <file>)
command line option, 7
- shared (--shared)
command line option, 8
- static
command line option, 8
- v (--verbose)
command line option, 7

C

CFLAGS, 7

command line option

- C, 8
- D/-U, 8
- E, 8
- H, 8
- I <dir> (-I<dir>), 8
- L<dir> (-L <dir>), 8
- O0, 5
- O1, 5
- O2, 5
- O3, 5
- O4, 5
- S, 7
- c, 7
- fPIC (-fpic), 7
- fPIE (-fpie), 7
- fprofile-generate[=<path>], 6
- fprofile-use[=<file>], 6
- fwhole, 6
- fwhole-shared, 6
- l<name> (-l <name>), 8

-m128, 7
-m32, 7
-m64, 7
-o<file> (-o <file>), 7
-shared (--shared), 8
-static, 8
-v (--verbose), 7