

УДК 681.3.06 (07)

Р 851



МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ТАГАНРОГСКИЙ ГОСУДАРСТВЕННЫЙ
РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ



**Руководство
к циклу лабораторных работ
по курсу**

ПРОГРАММИРОВАНИЕ СИГНАЛЬНЫХ ПРОЦЕССОРОВ

Для студентов специальности 2204

Таганрог 2000

УДК 681.3.06 (076.5) + 681.325.5. (076.5)

Составители: Н.Ш. Хусаинов, Д.П. Калачев

Руководство к циклу лабораторных работ по курсу “Программирование сигнальных процессоров”. Таганрог: Изд-во ТРТУ, 2000. ____ с.

Предназначено для студентов специальности 2204, изучающих курс "Программирование сигнальных процессоров". Содержит описание лабораторных работ, целью которых является знакомство с особенностями архитектуры и программирования на языке ассемблера сигнальных процессоров семейства ADSP-21000.

Ил. 9. Библиогр.: 5 назв.

Рецензент П.П.Кравченко, д-р. техн. наук, профессор кафедры МОП ЭВМ ТРТУ.

Хусаинов Наиль Шавкятович
Калачев Дмитрий Петрович

Руководство
к циклу лабораторных работ
по курсу

ПРОГРАММИРОВАНИЕ СИГНАЛЬНЫХ ПРОЦЕССОРОВ

Для студентов специальности 2204

Ответственный за выпуск Хусаинов Н.Ш.
Редактор Васютина О.К.
Корректор Проценко И.А.

ЛР № 020565 от 23.06.1997г. Подписано к печати . .2000 г.
Формат 60x84 ^{1/16} Бумага офсетная. Гарнитура литературная.
Офсетная печать. Усл.п.л. — 3,3. Уч.-изд. л. — 3,0.
Заказ № Тираж 200 экз.

"С"

Издательство Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Некрасовский, 44
Типография Таганрогского государственного
радиотехнического университета
ГСП 17А, Таганрог, 28, Энгельса, 1

1. БАЗОВАЯ АЛГОРИТМИКА И СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ADSP-21000

1.1. Микропроцессоры семейства ADSP-21000

Семейство микропроцессоров компании Analog Devices – ADSP-21000 ориентировано на обработку данных в формате с плавающей точкой. К данному семейству относятся микропроцессоры ADSP 21020, 21010, 21060, 21062.

Базовый представитель данного семейства - микропроцессор ADSP-21020 -имеет тактовую частоту 33,3 МГц и выполняет команды за 30 нс. Его производительность составляет 66 MFLOPS. Структурная схема ADSP-21020 приведена на рис.1.1.

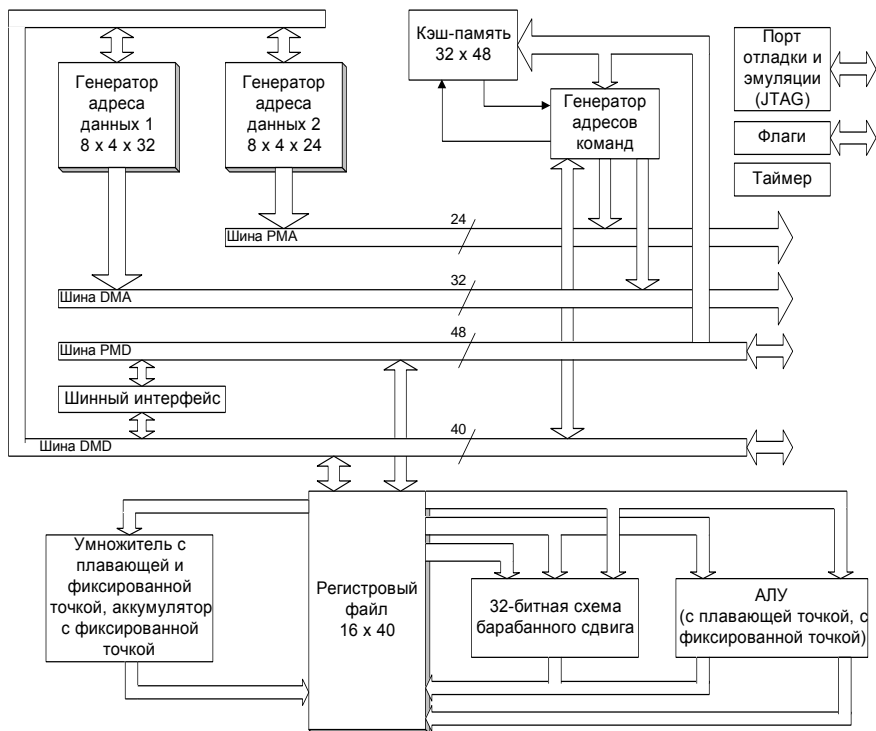


Рис.1.1. Структура микропроцессора

ADSP-21020 Вычислительные устройства

Каждый ADSP-21000 содержит три независимых полнофункциональных устройства:

- арифметико-логическое устройство (ALU), которое выполняет стандартный набор арифметических и логических операций с числами в формате с фиксированной запятой (ФЗ) и плавающей запятой (ПЗ);
- умножитель (MAC), выполняющий умножение с плавающей и с фиксированной точкой, а также умножение с накоплением для ФЗ-операндов;
- сдвигатель (Shifter), реализующий команды логических и арифметических сдвигов, манипулирования битами, выделения и депонирования битовых полей.

Вычислительные устройства функционируют независимо, что позволяет результату работы любого устройства быть операндом любого другого устройства уже в следующем такте. Реализация многофункциональных инструкций позволяет ALU и MAC выполнять операции в одном такте параллельно и независимо друг от друга.

Регистровый файл

Для обмена данными между вычислительными модулями служит регистровый файл общего назначения, содержащий 32 40-битных регистра (16 основных и 16 теневых).

Генераторы адресов данных

Генераторы адресов данных (DAG1 и DAG2) обеспечивают адресацию для доступа к памяти и поддерживают аппаратную реализацию циклических буферов и реверсирование битов адреса. Два DAG содержат 16 основных и 16 альтернативных регистров.

Шины памяти

Процессор не содержит памяти на чипе (кроме кэша инструкций) и может адресовать до 2^{24} слов внешней памяти команд (которая может содержать как команды, так и данные) и до 2^{32} слов внешней памяти данных (которая содержит только данные). Благодаря независимым шинам адресов и данных для каждого вида памяти, процессор может за один такт считывать/записывать два слова памяти разрядностью 32, 40 или 48 бит.

Кэш инструкций

Процессоры ADSP-21000 содержат высокопроизводительный кэш инструкций, который кэширует только те команды, выборка которых конфликтует с выборкой данных из памяти программ.

Программный секвенсор

Программный секвенсор выполняет конвейерную обработку команд в три стадии: выборка, декодирование и выполнение. Генератор адресов программы поддерживает выполнение условных переходов, вызовов подпрограмм и обработчиков прерываний, а также выполнение циклов без дополнительных тактов задержки.

Прерывания

ADSP-21000 имеет 5 внешних аппаратных прерываний: четыре – прерывания общего назначения (IRQ_{3-0}) и специальное прерывание сброса. Кроме того, поддерживаются внутренние прерывания таймера, переполнения кругового буфера, переполнения стека, арифметических исключений, а также программные прерывания, заданные программистом.

Таймер

Программируемый интервальный таймер обеспечивает периодическую генерацию прерываний. Он представляет собой 32-разрядный регистр, декрементируемый каждый такт. При достижении нуля происходит прерывание от таймера, а регистр счетчика обновляется и счет продолжается.

1.2. Состав средств разработчика

Система средств разработки программного обеспечения состоит из ассемблера, компоновщика, сплиттера, библиотеки утилит и симулятора и поддерживает все процессоры цифровой обработки сигналов семейства ADSP-21000 с плавающей точкой.

Блок-схема процесса разработки системы семейства ADSP-21000 приведена на рис.1.2.

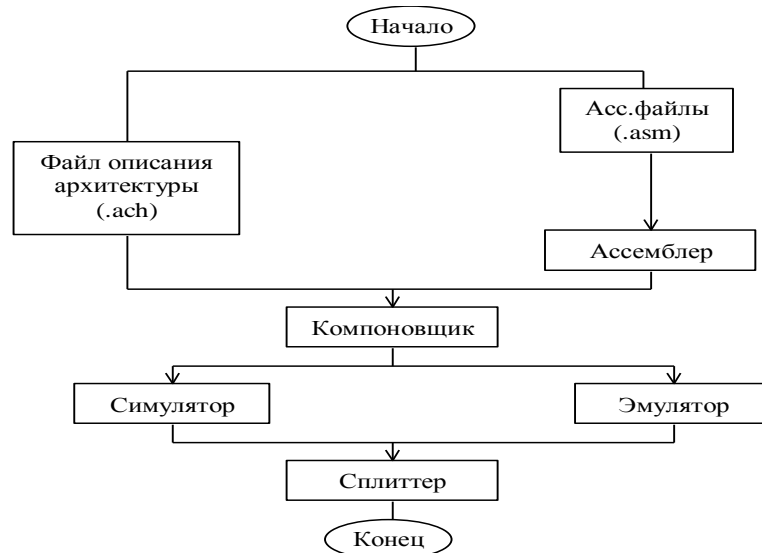


Рис.1.2. Процесс разработки системы семейства ADSP-21000

Процесс разработки программного обеспечения начинается с определения объектной аппаратуры. При этом задаются конфигурация системы, параметры доступа к памяти и размещение в ней сегментов, порты ввода/вывода для компоновщика, симулятора и эмулятора. Файл описания архитектуры (ФОА) имеет расширение ".ach".

Генерация кода начинается с написания исходных файлов на языке ассемблера. Каждый файл ассемблируется отдельно. Отдельные файлы затем компонуются вместе для формирования операционной программы.

Компоновщик читает информацию об объектной аппаратуре из ФОА чтобы определить расположение сегментов кода и данных. Неперемещаемые сегменты кода и данных размещаются по определенным адресам памяти с необходимыми атрибутами. Местоположение в памяти перемещаемых объектов определяется компоновщиком.

Компоновщик на основе объектных фалов и файлов-библиотек генерирует файл с расширением "exe", содержащий "карту памяти", который может быть загружен в симулятор или эмулятор для тестирования.

Симулятор – средство отладки программ, моделирующее устройство ADSP-21000 и память, определенную в файле архитектуры, а также обмен данными с устройствами ввода/вывода посредством использования файлов данных. Симулятор программно выполняет программу так же, как и процессор ADSP-21000 в действительности. Это моделирование позволяет отлаживать систему и анализировать ее производительность перед переходом к аппаратному прототипу.

После полного моделирования системы и программных средств можно использовать эмулятор в аппаратном прототипе для тестирования аппаратных схем, синхронизации и выполнения программы в реальном времени.

Сплиттер переводит исполнимые программы с выхода компоновщика в файл одного из промышленных стандартов для программатора ППЗУ. После установки ППЗУ в объектную аппаратуру процессор ADSP может выполнять прикладные программы.

Библиотекарь – это утилита, которая группирует объектные файлы вместе для получения библиотечных файлов. При компоновке программы можно специфицировать библиотечный файл, и компоновщик автоматически скомпилирует любой файл с библиотекой, которая содержит метку, используемую в основной программе.

1.3. Ассемблер

Для вызова ассемблера используется командная строка вида

ASM21K [-ключ] имя_файла

Возможные значения ключа:

-h	список возможных ключей
-pp	прогон препроцессора без ассемблера
-sp	пропуск препроцессора, прогон только ассемблера
-d идентификатор [=буквы]	определение идентификатора для препроцессора
-l	генерация листинг-файла (.lst)
-o имя	имя выходного объектного файла (.obj)
-v	вывод на экран программной информации

1.4. Компоновщик

Компоновщик генерирует файл образа памяти посредством компоновки вместе отдельных ассемблерных сегментов. Выход компоновщика – это файл изображения памяти (загрузочный модуль), который загружается в симулятор и эмулятор для отладки.

Компоновщик сканирует каждый ассемблированный сегмент и восстанавливает глобальные обращения между сегментами. Он назначает адреса для перемещения кода и данных. Компоновщик читает ФОА и размещает код и данные в соответствии с системным распределением памяти.

Компоновщик вызывается следующим образом:

LD21K [-ключ] *файл_1* [*файл_2*]

Возможные значения ключа:

-a archfile	имя ФОА (archfile)
-o executable	имя выходного ехе-файла (executable)
-i listfile	компоновка всех файлов, перечисленных в файле listfile
-m	генерация map-файла
-s	удаление таблицы символов из ехе-файла
-x	удаление таблицы локальных символов из ехе-файла
-v	отображение на дисплее программной информации
-h	возможные варианты ключа

1.5. Симулятор

Вызов симулятора осуществляется следующим образом:

SIM21K -a файл.ехе -е файл.асм

В отличие от ассемблера и компоновщика симулятор обеспечивает взаимодействие с пользователем посредством оконного интерфейса.

Для DOS-симуляторов процессоров семейства ADSP-21000 активизация меню выполняется по нажатию клавиши [F3]. Основные элементы главного меню:

File	загрузка файлов .ach, .exe
Core	информация о процессорном ядре, регистрах;
Memory	распределение памяти, сегментация, операции с памятью;
Execution	выполнение программ
Window	настройка окон
Help	Помощь

1.5.1. Загрузка программы

В начале моделирования необходимо загрузить файл архитектуры и файл карты памяти. Для этого в меню File следует выбрать пункт Load File и загрузить необходимый архитектурный файл (.ACH), а затем аналогично загрузить файл памяти (.EXE). Результат распределения памяти после загрузки ФОА можно посмотреть, выбрав пункт меню Memory->Segment Map.

1.5.2. Просмотр содержимого памяти

Для просмотра содержимого памяти выбрать меню Memory и в нем соответствующий пункт (Program, Data, ...). Если необходимо посмотреть содержимое памяти по определенному адресу – нажать клавишу [Ctrl-G] и ввести требуемый адрес памяти или метку памяти.

Режим просмотра информации в окне изменяется при нажатии [Ctrl-T] и может принимать следующие значения: Float, Fixed, Hex, Disassembled. Естественно, что при просмотре памяти программ следует использовать режим Disassembled, а при просмотре данных – в зависимости от характера данных.

Для каждой ячейки памяти высвечивается ее тип:

ROM – только чтение;

RAM – чтение и запись;

PORT – порт;

----- - несуществующая память, не определенная в ФОА.

1.5.3. Загрузка содержимого памяти из файла данных и выгрузка содержимого памяти в файл данных

Для загрузки данных выберите пункт меню Memory->Operations->Load, для выгрузки – Memory->Operations->Dump. При загрузке и выгрузке данные представляются в ASCII-кодах их шестнадцатеричного представления.

1.5.4. Выполнение программы

Для выполнения шага программы выбрать пункт меню Execution->Step или нажать клавишу [F10].

Пошаговое выполнение – относительно медленный способ выполнения программы. Для «прогона» программы (непрерывного выполнения команд) выберите пункт меню Execution->Run или нажмите [F4]. Чтобы остановить симулятор, когда он выполняет прогон, надо войти в любое окно и нажать любую клавишу на клавиатуре. Значения всех окон будет скорректировано и последнее активное окно станет активным.

Для остановки выполнения и сброса выберите пункт меню Execution->Chip Reset (аналогично подаче сигнала RESET на вход процессора) или Execution->Simulator Reset (еще и сброс содержимого памяти и карты памяти).

1.5.5. Точки останова

Точки останова (или контрольные точки) останавливают прогон симулятора точно перед тем, как процессор выполняет команду по указанному адресу. Для установки контрольных точек останова выберите строку программы (в памяти программ) и выберите пункт меню Execution->Assemble Breaks->Toggle или нажмите [F9].

При установке так называемых "мультиконтрольных" точек есть возможность остановить симулятор на требуемой ячейке памяти только перед n-проходом. Для этого установите курсор на требуемую команду, нажмите [Shift-F10] и введите число n.

1.5.6. Наблюдение выполнения программы

Для контроля хода выполнения программы симулятор позволяет отображать на экране содержимое всех регистров, доступных для программиста:

Окно	Со- держимое	Описание
Program Counter	PC DADDR FADDR PCSTK PCSTKR	Окно программного секвенсора: адрес выполняемой инструкции; адрес декодируемой инструкции; адрес выбираемой инструкции; адрес на верхушке стека команд; количество адресов в стеке команд
PC Stack		Окно стека команд
Loop Counter	CURLCNTR LCNTR	Окно счетчика циклов: счетчик текущего цикла = $\text{top}(\text{Loop Counter Stack})$; счетчик внешнего цикла = $\text{top}(\text{Loop Counter Stack}) + 1$;

	LADDR	указатель на последнюю команду текущего цикла
Loop Counter Stack		Окно стека счетчиков циклов
Loop Address Stack	term addr term code loop type	Окно стека адресов последних команд и условий окончания вложенных циклов; адрес последней команды цикла; код выхода из цикла; тип цикла: 00 - по арифм. условию; 01 – по счетчику = 1; 10 – по счетчику = 2; 11 – по счетчику > 2
PX		Регистры обмена между шинами PMD и DMD
MR		Регистры аккумулятора MAC'a
ASTAT		Флаги
USTAT		Регистр статуса пользователя

STKY		Регистр «липких» флагов (сброс только вручную)
MODE1		Регистр управления
MODE2		Регистр управления
Cycles	LS Cycles	Счетчик выполненных инструкций
DAG1 и DAG2	I M B L	Окна генераторов адресов памяти данных и памяти команд: индексный регистр; регистр-модификатор; регистр базы; регистр длины
Register File	R0 ÷ R15	Окно регистрового файла

1.5.7. Имитация работы с портами ввода/вывода

Симулятор позволяет имитировать работу с портами ввода/вывода, отображаемыми в память. Для имитации потока данных используются текстовые файлы со значениями, записанными по одному в строке. Для установления связи между ячейкой памяти по заданному адресу и файлом данных используется пункт меню Data->Ports->Open. Опция AUTOWRAP для порта ввода данных позволяет при достижении конца файла продолжить чтение в ячейку данных с его начала.

Другой способ задания отображаемых в память портов ввода/вывода –загрузка в симулятор файла с описанием используемых портов (тестовый файл с расширением prt). Синтаксис строки описания каждого порта имеет вид:

port pm|dm address hex|fix|float y|n infile outfile

где переключатель hex|fix|float позволяет задать формат данных для обмена с портом, а переключатель y|n разрешает или запрещает режим AUTOWRAP. Например:

```
port dm 0x00100 hex y input.dat null
```

Физический обмен данными между симулятором и файлом осуществляется в момент чтения/записи ячейки при прогоне программы.

1.5.8. Имитации внешних прерываний и флагов

Симулятор позволяет имитировать генерируемые внешними устройствами запросы на прерывания по каналам IRQ_{3-0} . Для конфигурирования внешних автоматических прерываний используется пункт меню Set-up->Auto Interrupt Control. При выборе режима Periodic в поле Period вводится количество тактов, через которые будут периодически генерироваться прерывания. Для режима Random следует заполнить поля Min и Max, обозначающие соответственно нижнюю и верхнюю границы (в тактах), в пределах которых генерируется случайное число тактов до следующего прерывания. Значение в поле Cycles содержит количество тактов до следующего прерывания на текущий момент выполнения программы.

Аналогично могут быть заданы установки для имитации внешних входных флагов $FLAG_{3-0}$ (пункт меню Set-up->Auto-Flag Control).

1.6. Контрольные вопросы

1. Какие элементы входят в состав средств разработчика программного обеспечения для процессоров семейства ADSP-21000?
2. Какова последовательность действий от написания программы на языке ассемблера до получения выполняемого кода программы для ADSP-21000?
3. Что такое ассемблер? Для чего используется компоновщик?
4. Получите исполняемый файл с программой и загрузите его в симулятор.

5. Просмотрите содержимое памяти по заданному адресу.
6. Какие регистры используются в вычислительных операциях ADSP-21000?
7. Какие регистры используются при адресации данных ADSP-21000?
8. Свяжите отображаемый в память порт ввода или вывода с файлом на диске.

2. ОРГАНИЗАЦИЯ ДОСТУПА К ВНЕШНЕЙ ПАМЯТИ ADSP-21000. ФАЙЛ АРХИТЕКТУРЫ

2.1. Конфигурация памяти ADSP-21000

Процессоры семейства ADSP-21000 построены по Гарвардской архитектуре и имеют два различных, но принципиально однотипных интерфейса с внешней памятью (память на кристалле имеется только в процессорах семейства ADSP-2106x): интерфейс с памятью данных (Data Memory (DM), содержащей только данные) и интерфейс с памятью программ (Program Memory (PM), которая может содержать и команды, и данные). Общее адресуемое пространство памяти программ составляет 16М слов, а памяти данных – 4Г слов.

Каждая система на основе процессоров ADSP-21000 может иметь уникальную конфигурацию аппаратного обеспечения и может использовать различные объемы физической внешней памяти. Для спецификации конкретной аппаратной конфигурации памяти, расположения в ней сегментов кода и данных, а также портов ввода/вывода необходимо написать файл описания архитектуры, который используется компоновщиком для размещения сегментов кода и данных в возможном пространстве памяти, а симулятором – для точного моделирования системной архитектуры.

2.2. Файл описания архитектуры

Файл описания архитектуры (ФОА) имеет расширение АСН. В нем указываются символические имена для задания конфигурации системы, сегментов памяти и отображаемых в память портов ввода/вывода. Использование аналогичных имен в программе позволяет компоновщику определить точное расположение всех сегментов программы в

памяти.

Все символические имена должны быть уникальными. Символическое имя – это цепочка букв, цифр и символов подчеркивания длиной не более 32 элементов. Некоторые ключевые слова являются зарезервированными и не могут быть использованы в ФОА, например в качестве имени сегмента.

2.2.1. Директивы файла архитектуры

Ниже приводится описание основных директив, используемых в файле описания архитектуры, и их синтаксис. Формат некоторых директив требует указания дополнительных параметров, разделяемых наклонной чертой. Общая форма директивы может быть записана в виде

.DIRECTIVE /спецификатор/спецификатор/... параметр

Директива присваивания имени системе SYSTEM должна быть указана первой в ФОА. Директива SYSTEM имеет форму

.SYSTEM system_name;

Директива ENDSYS завершает описание конфигурации системы и должна быть последней в файле архитектуры:

.ENDSYS;

Директива PROCESSOR идентифицирует, какой процессор семейства ADSP-21000 применяется в системе. Она используется средствами разработки программного обеспечения ADSP для определения, какие особенности архитектуры он имеет. Директива имеет вид:

.PROCESSOR=наименование процессора;

где наименование процессора – это ADSP21020, ADSP21010, ADSP21060, ADSP21061 или ADSP21062. Например, ФОА для процессора ADSP-21020 включает в себя директиву:

.PROCESSOR=ADSP21020;

Директива объявления сегмента памяти SEGMENT определяет часть системной памяти (сегмент) и описывает ее атрибуты. Средствами разработчика не поддерживается распределение памяти по умолчанию – необходимо определить всю системную память директивами SEGMENT. Эта информация поступает в компоновщик, симулятор и эмулятор. Компоновщик размещает программный код и данные в соответствии с параметрами указанного распределения памяти.

Директива SEGMENT имеет вид:

.SEGMENT спецификатор... имя сегмента

Сегменту назначается символическое имя. Это имя должно состоять из восьми или менее элементов. Основные спецификаторы директивы SEGMENT:

/RAM	память для чтения/записи
/ROM	память только для чтения
/PORT	сегмент отображается в порт
/PM	сегмент отображается в пространство программной памяти
/DM	сегмент отображается в пространство памяти данных
/BEGIN =адрес	начало сегмента
/END=а дрес	конец сегмента
/WIDTH =n	разрядность слов в сегменте (n может принимать значения 16, 32, 40 или 48)

Например, директива

```
.SEGMENT /RAM/BEGIN=0x0/END=0x7FF/PM IRQTable;
```

объявляет сегмент RAM программной памяти, названный IRQTable, который расположен в начале пространства памяти команд и занимает 2048 слов.

Количество сегментов не ограничено, однако для минимальной работы программы в памяти должен быть хотя бы один командный сегмент, расположенный по адресу 0x00000008 (по этому адресу передается управление при прерывании по сбросу процессора ADSP-2102x).

Спецификатор /PORT объявляет отображаемые в память порты ввода/вывода, обычно представляющие собой сегмент размером в одно слово. Отображение в память портов является одним из механизмов организации взаимодействия ADSP-21000 с другими устройствами в системе. Порты могут располагаться как в памяти данных, так и в памяти программ. Например, директива

```
.SEGMENT /PORT/BEGIN=0x400/END=0x400/DM port_1;
```

объявляет порт с именем port_1, который расположен по адресу 1024 в памяти данных.

Директива конфигурации памяти системы BANK позволяет определить размеры банков физической памяти, состояния ожидания, размеры страниц и режимы доступа к ним, что особенно важно при работе с памятью со страничной организацией (DRAM). Директива BANK используется для конфигурации физической памяти, тогда как директива SEGMENT – для логической конфигурации памяти. Директива BANK имеет следующий формат:

.BANK [спецификатор(ы)]

В качестве спецификаторов могут выступать следующие ключевые слова:

/WTSTATES =n	количество состояний ожидания для банка (n=0÷7)
/WTMODE= режим	<p>режим ожидания при обращении к банку памяти. Параметр режим может принимать одно из значений:</p> <p>=EXTERNAL (процессор ожидает сигнала подтверждения от памяти);</p> <p>=INTERNAL (сигнал подтверждения игнорируется, количество циклов ожидания определяется числом, записанным в регистре управления памятью для данного банка или в поле WTSTATES);</p> <p>=BOTH (процессор ждёт сигнал подтверждения от памяти, но число циклов ожидания не может быть меньше указанного в регистре управления памятью);</p> <p>=EITHER (цикл обращения к памяти завершается либо по истечении числа тактов ожидания, либо по приходу сигнала подтверждения).</p>
/PGSIZE=n	размер страницы DRAM-памяти (значение n должно быть в пределах от 256 до 32768 и представлять собой степень числа 2)
/PM0 или /PM1	выбор банка памяти команд

/DM0 или /DM1 или /DM2 или /DM3	выбор банка памяти данных
/BEGIN=адрес	начальный адрес выбранного банка памяти

Например, директивы

.BANK /PM1/WTSTATES=0/WTMODE=INTERNAL/BEGIN=0x008000;

определяет банк №1 памяти команд с количеством тактов ожидания, равным нулю (т.е. использование статической (SRAM) памяти), режимом ожидания INTERNAL и начинающийся с адреса 0x008000.

.BANK /DM0/WTSTATES=2/WTMODE=EITHER/BEGIN=0x0;

определяет банк №0 памяти данных (нулевой банк всегда начинается с нулевого адреса) с режимом ожидания EITHER и количеством тактов ожидания, равным 2.

При написании программ для отладки без использования аппаратных средств отладки (только симулятор) директива .BANK в ФОА может быть опущена (в этом случае используется конфигурация банков памяти по умолчанию).

2.2.2. Комментарии в файле архитектуры

Строки в файле архитектуры, начинающиеся с символов # или !, считаются комментарием и игнорируются синтаксическим анализатором файла архитектуры.

2.3. Адресация данных

2.3.1. Режимы адресации

Процессоры семейства ADSP-21000 поддерживают следующие режимы адресации:

- непосредственную адресацию, обеспечивающую запись константы в регистр:

R6 = 0x2000; ! запись в регистр R6 значения 0x2000

B2 = in_buffer; ! запись в регистр B2 константы - адреса ! буфера in_buffer
L2 = @in_buffer; ! запись в регистр L2 константы - длины ! массива in_buffer

- прямую адресацию, обеспечивающую запись в память или чтение из памяти значения универсального регистра.

Например:

DM(temp1) = R6; ! запись значения регистра R6 в ячейку ! памяти temp1

R2 = RM(temp2); ! чтение из памяти значения в регистр R6

- косвенную адресацию, которая осуществляется с использованием регистров адресных генераторов DAG1 и

DAG2. Например:

DM(I1, 0) = R6; ! запись значения регистра R6 в ячейку
! памяти по адресу I1

R2 = RM(I8, M9); ! запись в регистр R2 значения по адресу
! I8 с последующей модификацией регистра
! I8 значением регистра M9

2.3.2. Генераторы адреса данных

В процессорах семейства ADSP-21000 существует 2 независимых генератора адресов данных для того, чтобы одновременно иметь доступ к памяти данных и памяти программы:

- DAG1 – генерирует 32-битные адреса, которые используются для доступа к памяти данных;

- DAG2 – генерирует 24-битные адреса, которые используются для доступа к памяти программ.

Каждый из адресных генераторов DAG содержит регистровый файл, включающий в себя восемь наборов регистров. Генератор DAG1 содержит с 0-го по 7-й наборы, а DAG2 - с 8-го по 15-й. Каждый набор состоит из следующих регистров:

- базовые регистры (B – регистры), определяющие базовый адрес буфера данных;

- регистры длины буфера данных в словах (L – регистры);

- регистры текущего адреса (I – регистры);

- регистры модификации текущего адреса (M – регистры).

Модификация регистров текущего адреса возможна в двух вариантах: постмодификация и предмодификация. В

первом случае значение в I-регистре изменяется после обращения к памяти. Во втором – на шину адреса выставляется модифицированное значение, но само значение в I-регистре не изменяется! Любой I-регистр может быть модифицирован константой или любым M-регистром, но только из того же DAG;

Например:

Пример команды	Значение, выставленное на шину адреса	Значение I-регистра после выполнения команды
$R6 = DM(I4, M2)$	I4	$I4 + M2$
$R2 = DM(I4, 0x0200)$	I4	$I4 + 0x0200$
$R6 = DM(M2, I4)$	$I4 + M2$	I4
$R6 = DM(0x0200, I4)$	$I4 + 0x0200$	I4

В некоторых задачах (например, фильтрации) требуется повторное продвижение указателя в область памяти. Для этого в ADSP-21000, используется круговой буфер, по которому индексный указатель перемещается с применением постмодификации. Если модифицированный указатель выходит за пределы буфера, то длина буфера вычитается (при значении M-регистра меньше 0 – прибавляется) из полученного значения чтобы вернуть указатель к началу буфера.

Примечание. При инициализации значений I- и V-регистров одинаковыми значениями, сначала нужно инициализировать значение V-регистра, т.к. при записи значения в регистр Vx в регистр Ix с тем же номером заносится то же

значение.

Например:

```
B1 = in_buffer;      ! в I-регистр автоматически
                     ! заносится адрес in_buffer
L1 = @in_buffer;      ! задать размер кругового буфера
M1 = 1;               ! задать смещение текущего адреса
                     ! после каждого шага
```

2.3.3. Особенности использования DAG регистров

При работе с адресными генераторами следует учитывать следующие их особенности:

- если следом за инструкцией загрузки идёт адресация с использованием того же генератора адреса, то ADSP автоматически вставляет дополнительный цикл (операцию por). Например:

```
L2 = 8;
DM(I0, M1) = R1;
```

- дополнительный цикл также добавляется процессором ADSP после команды записи в регистры управления памяти, если после неё идёт адресация с использованием соответствующего этой памяти генератора DAG. Например:

```
DMBANK1 = значение;
R10 = DM(I0, M1);
```

- если за инструкцией записей в регистры Mx или Lx в DAG2 следует команда чтения регистра Ix с тем же номером, то между ними необходимо вставить операцию por, иначе из регистра будет прочитано старое значение. Например:

```
L8 = 24;
por;
R0 = I8;
```

- недопустимы команды загрузки из памяти и сохранения в память DAG регистров с использованием косвенной адресации с тем же DAG набором. Например:

```
DM(M2, I1) = I0;      ! недопустимо
```

2.4. Пример

Написать файл описания архитектуры и программу на языке ассемблера, которая выполняет заданные арифметические действия над целыми числами, расположенными в памяти команд (P) и в памяти данных (D) с указанными смещениями от некоторого произвольного базового адреса:

$$8*(P_0 + D_0) - ((P_1 - D_1) * (D_2 - P_2) - (D_4 + D_6 + D_8 + D_{10}) * (P_{12} + P_{10} + P_8 + P_6))$$

Файл описания архитектуры:

```
.system example;
.processor ADSP21020
.segment /ram /begin=0x00008 /end=0x0000f /pm pm_rsti;
.segment /ram /begin=0x00100 /end=0x00200 /pm pm_code;
.segment /ram /begin=0x00400 /end=0x00410 /dm dm_data;
.segment /ram /begin=0x00400 /end=0x00410 /pm pm_data;
.segment /ram /begin=0x00410 /end=0x00411 /dm dm_res;
.endsys;
```

Текст программы на языке ассемблера:

```
#include <def21020.h>
#define DM_Size 11
#define PM_Size 13
! сегмент в памяти данных для хранения значений массива D
.SEGMENT/DM dm_data;
.VAR D[DM_Size]="t_1.dat";    ! загружается из t_1.dat
.ENDSEG;
! сегмент в памяти программ для хранения массива P
.SEGMENT/PM pm_data;
.VAR P[PM_Size]="t_2.dat";    ! загружается из t_2.dat
```

```

.ENDSEG;
! сегмент в памяти данных для хранения результата
.SEGMENT/DM dm_res;
.VAR res;
.ENDSEG;
! сегмент обработки прерывания сброса
.SEGMENT/PM pm_rsti;
  JUMP start (DB);          ! переход по сбросу с задержкой
  DMWAIT=0x21;              ! установить параметры памяти
  PMWAIT=0x21;
.ENDSEG;
      ! сегмент выполнения вычислений
.SEGMENT/PM pm_code;
start:
I1 = D; M1 = 1;              ! I1= $\wedge$ D0
I8 = P; M8 = 1;              ! I8= $\wedge$ P0
F6 = 8.0;
F0 = DM(I1,M1), F3=PM(I8,M8); ! F0=D0, F3=P0
      ! F0=(D0+P0), F1=D1, F2=P1, I1= $\wedge$ D2, I8= $\wedge$ P2
F0 = F0+F3, F1 = DM(I1,M1), F2=PM(I8,M8);
M1 = 2; M8 = 2;
      ! F3=P1-D1, F1=D2, F2=P2, I1= $\wedge$ D4, I8= $\wedge$ P4
F3 = F2-F1, F1=DM(I1,M1), F2=PM(I8,M8);
F4 = F1-F2;                  ! F4=D2-P2
MODIFY(I8,2);                ! I8= $\wedge$ P6
      ! F10=(P1-D1)*(D2-P2), F3=D4, F7=P6, I1= $\wedge$ D6, I8= $\wedge$ P8
F10 = F3*F4, F3=DM(I1,M1), F7=PM(I8,M8);
      ! F8=D6, F12=P8, I1= $\wedge$ D8, I8= $\wedge$ P10

```

```

F8 = DM(I1,M1), F12=PM(I8,M8);
      ! F3=(D4+D6), F8=D8, I1=^D10
F3 = F3+F8, F8=DM(I1,M1);
      ! F7=(P6+P8), F12=P10, I8=^P12
F7 = F7+F12, F12=PM(I8,M8);
      ! F3=(D4+D6+D8), F8=D10, I1=^D12
F3 = F3+F8, F8=DM(I1,M1);
      ! F7=(P6+P8+P10), F12=P12, I8=^P0
F7 = F7+F12, F12=PM(I8,M8);
F3 = F3+F8;                                ! F3=(D4+D6+D8+D10)
F7 = F7+F12;                                ! F7=(P6+P8+P10+P12)
      ! F14=(D4+D6+D8+D10) * (P6+P8+P10+P12), F0=P0+D0
F14 = F3*F7;
      ! F0=(P0+D0)*8, F1=( ) * ( ) - ( ) * ( )
F0 = F0*F6, F1=F10-F14;
F0 = F0-F1;                                ! результат
DM(res) = F0;
idle;
.ENDSEG;

```

Время выполнения программы с метки start составляет 44 такта.

2.5. Варианты заданий

Написать файл описания архитектуры и программу на языке ассемблера, которая выполняет указанные арифметические действия над целыми числами, расположенными в памяти команд (A) и в памяти данных (B) с указанными смещениями от некоторого произвольного базового адреса, не упрощая выражения.

1. $((A_1 + B_2) - (A_3 + B_4) - (A_5 + B_6)) + ((A_2 + B_1) - (A_4 + B_3) - (A_6 + B_5)) - (A_0 + B_7 - A_8 + B_7)$
2. $(A_1 + B_1 - A_3 + B_2) - ((A_5 - B_3 + A_7 - B_4) - (A_9 + B_5) - (A_0 + B_0)) - (A_0 + B_2 - A_3 + B_7)$
3. $(A_0 + B_0 - A_1 + B_1 - A_3 - B_2 + A_5 - B_3) - ((A_5 + B_7) - (A_6 + B_5)) - (A_7 + B_3) - (A_8 + B_1)$

4. $(A_9 + B_8) - (A_7 + B_7) - (A_5 - B_6 + A_4 - B_5) - ((A_0 + B_1) - (A_3 + B_2) - (A_6 + B_3)) - (A_9 + B_4)$
5. $((A_2 + B_5) - (A_3 + B_3) - (A_4 + B_1 + A_5 - B_7)) - ((A_0 + B_5) + (A_1 + B_3) - (A_4 + B_2) - (A_6 + B_3))$
6. $((A_3 + B_2) - (A_1 + B_0)) - (A_3 + B_5 + A_7 - B_9)) - (A_0 + B_3 - (A_2 + B_6) - (A_4 + B_8)) - (A_2 - B_1))$

2.6. Контрольные вопросы

1. Каковы особенности организации интерфейса процессора с внешней памятью и регистры процессора, отвечающие за организацию интерфейса процессора с внешней памятью?
2. Что такое файл описания архитектуры? Каково его назначение?
3. Какие директивы являются обязательными в ФОА, а какие можно опустить и в каких случаях?
4. Какая связь между одноименными директивами SEGMENT в файле архитектуры и в тексте программы на языке ассемблера ADSP-21000?
5. Назовите режимы адресации, используемые в программах на языке ассемблера.
6. Что такое генератор адресов данных и для чего он используется в ADSP?
7. Какие регистры входят в состав DAG?
8. Что такое кольцевые буферы? Как задать кольцевой буфер в программе?

3. ВЫЧИСЛИТЕЛЬНЫЕ БЛОКИ ПРОЦЕССОРОВ СЕМЕЙСТВА ADSP-21000

3.1. Обзор

Для числовой обработки данных в процессорах семейства ADSP-21000 имеется три вычислительных блока: арифметико-логическое устройство (АЛУ), умножитель и устройство сдвига (сдвигатель). Структура вычислительных блоков приведена на рис. 3.1.

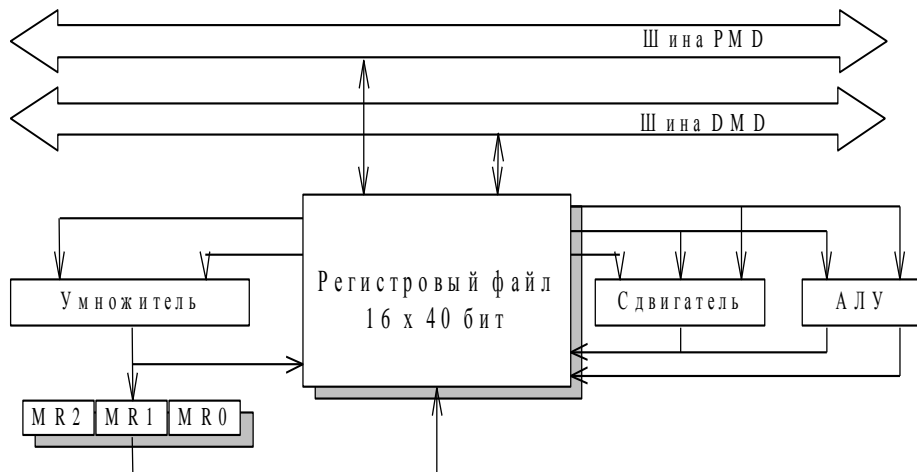


Рис.3.1. Структура вычислительных блоков процессоров семейства ADSP-21000

АЛУ выполняет стандартный набор арифметико-логических инструкций над числами в формате с фиксированной запятой (ФЗ) и плавающей запятой (ПЗ). Умножитель выполняет ФЗ- и ПЗ-умножение, а также ФЗ-умножение с накоплением (сложением или вычитанием). Сдвигатель выполняет логические и арифметические сдвиги, битовые манипуляции, операции с битовыми полями. Выполнение инструкции в любом вычислительном блоке занимает один цикл.

Вычислительные блоки получают данные и выдают результаты в регистровый файл (РФ), состоящий из 16 основных и 16 альтернативных (теневых) регистров. Регистровый файл доступен программе и шинам памяти данных для пересылки данных между вычислительными блоками и внешней памятью или другими частями процессора.

Регистры РФ, имеющие в программе на языке ассемблера префикс "F", используются в ПЗ-операциях, а имеющие

префикс "R" – в ФЗ-операциях. Физически ПЗ- и ФЗ-данные разделяют одни и те же регистры РФ, а соответствующие префиксы определяют только как АЛУ, сдвигатель или умножитель должны интерпретировать и обрабатывать данные. Вычислительными блоками могут обрабатываться 32-разрядные числа с фиксированной запятой и 32- или 40-разрядные числа с плавающей запятой.

Числа с ФЗ всегда представляются 32 битами и являются выровненными влево (занимают 32 старших разряда) в 40-битных полях данных. Они могут обрабатываться как целые или дробные числа, а также как беззнаковые числа или числа в дополнительном коде.

Формат представления чисел в ADSP-21000 приведен в приложении 1.

3.2. Арифметико-логическое устройство

3.2.1. Режимы работы АЛУ

АЛУ считывает один или два входных операнда (X, Y), которые могут быть любыми регистрами данных в регистровом файле. Обычно АЛУ возвращает один результат; в операциях дуального сложения/вычитания – два результата; в командах сравнения - не возвращает никакого результата (только модифицируются флаги). Формат результата зависит от формата операндов.

На работу АЛУ влияют три бита состояния в регистре MODE1 (приложение 2):

Б ит	Им я	Функция
1 3	AL USAT	1 – разрешает режим насыщения АЛУ / 0 – запрещает
1 5	TR UNC	1 – округление к нулю / 0 – округление к ближайшему
1 6	RN D32	1 – округление к 32-битной границе / 0 – округление к 40-битной границе

3.2.2. Флаги АЛУ

После каждой операции АЛУ модифицирует 7 флагов состояния и биты аккумулирующего сравнения в регистре ASTAT.

Б ит	Им я	Определение
0	AZ	нулевой результат или потеря значимости при ПЗ-операции
1	AV	переполнение
2	AN	отрицательный результат
3	AC	ФЗ-перенос
4	AS	знак X-операнда (для инструкций ABS и MANT)
5	AI	некорректная ПЗ-операция
6	AF	флаг ПЗ-операции
3 1-24	CA CC	результаты последних восьми операций сравнения (COMP)

АЛУ модифицирует также 4 "липких" флага в регистре STKY (однажды установленный "липкий" флаг остается таким до принудительного сброса).

Б ит	Им я	Определение
0	AU S	потеря значимости при ПЗ-операции
1	AV S	переполнение при ПЗ-операции
2	AO S	переполнение при ФЗ-операции
5	AIS	некорректная ПЗ-операция

3.2.3. Инструкции АЛУ

В приложении 3 приведены инструкции АЛУ для работы с ФЗ- и ПЗ-числами и показано их действие на флаги.

3.3. Умножитель

3.3.1. Операции умножителя

Умножитель выполняет ФЗ- и ПЗ-умножение и ФЗ-умножение с накоплением. ПЗ-инструкции умножителя оперируют с 32- или 40-битными ПЗ-операндами и вырабатывают 32- или 40-битный ПЗ-результат. ФЗ-инструкции умножителя оперируют с 32-битными ФЗ-данными и производят 80-битный результат. Входные операнды обрабатываются как дробные или целые числа, беззнаковые числа или числа в дополнительном коде.

Умножитель берет два входных операнда (X и Y), которые могут быть любыми регистрами РФ. Результат ПЗ-операции всегда записывается в регистр РФ. ФЗ-операции могут накапливать ФЗ-результаты в любом из локальных регистров результата умножителя (MR) или записывать результаты обратно в РФ. Результаты, хранящиеся в MR-регистрах, могут также быть округлены или насыщены отдельными операциями.

Местоположение ФЗ-результата в 80-битном поле регистра MR зависит от того дробный или целый формат имеет

результат (рис.3.2).

M R 2		M R 1		M R 0	
79	64	63	32	31	0
переполнение		дробный результат		потеря значимости	
переполнение		переполнение		целый результат	

Рис.3.2. Размещение ФЗ-результата умножителя

Данные могут быть записаны в MR0, MR1, MR2 из старших 32-х разрядов ячейки РФ. При записи значения в MR1 происходит его знаковое расширение в MR2. При записи значения в MR0 знакового расширения не происходит. При записи в РФ значения из MR0, MR1, MR2 младшие 8 бит 40-битного слова заполняются нулями.

Два регистра умножителя MR называются регистрами переднего плана (MRF) и заднего плана (MRB) и имеют идентичный формат. В отличие от других блоков, для переключения к "теневым" наборам которых требуется переключение бита в регистре управления MODE1, регистры переднего и заднего плана умножителя доступны одновременно и выбираемый регистр указывается в команде. Например:

```
MR1F = R5;    ! записать в поле MR1 регистра MRF значение
               ! из R5
R2 = MR0B;    ! записать в R2 значение поля MR0 регистра
               ! MRB
MRF = MRF + R5*R0;    ! прибавить к содержимому регистра
                     ! MRF результат произведения R5*R0
```

Умножитель управляется двумя битами состояния режима в регистре MODE1, (TRUNC и RND32), которые влияют на операции умножителя так же, как и на операции АЛУ.

3.3.2. Флаги умножителя

Умножитель модифицирует 4 флага состояния в регистре ASTAT после каждой операции.

Б ит	Им я	Определение
6	MN	отрицательный результат
7	MV	переполнение
8	MU	потеря значимости
9	MI	некорректная ПЗ-операция

Умножитель модифицирует также 4 "липких" флага в регистре STKY.

Б ит	Им я	Определение
6	MO S	переполнение при ФЗ-операции
7	MV S	переполнение при ПЗ-операции
8	MU S	потеря значимости
9	MI S	некорректная ПЗ-операция

3.3.3. Инструкции умножителя

Инструкции умножителя и их действие на флаги описаны в приложении 3. В скобках после инструкции приводят-

ся модификаторы, определяющие тип операндов и режим округления результата:

S – знаковый операнд;

U – беззнаковый операнд;

I – целый операнд;

F – дробный операнд(ы);

FR – дробные операнды, результат округлять в соответствии с установкой битов RND32 и TRUNC в регистре MODE1.

3.4. Сдвигатель

3.4.1. Операции сдвигателя

Сдвигатель берет от одного до трех входных операндов: X – операнд, который обрабатывается, Y – операнд, специфицирующий абсолютные значения величины сдвига, длины битовых полей или позиции битов, Z – операнд, который обрабатывается или модифицируется. X- и Z – операнды – всегда 32-битные ФЗ-величины. Y-операнд – 32-битная ФЗ-величина или 8-битное поле (shf8), позиционированное в регистре РФ в битах 15÷8. Входные операнды читаются из старших 32 битов ячейки РФ или непосредственно из инструкции.

Сдвигатель возвращает один результат в РФ. Некоторые инструкции сдвигателя порождают 8- или 6-битные результаты. Эти результаты помещаются в поля shf8 (биты 15÷8) или bit6 (биты 13÷8), при этом знак распространяется на старшие разряды регистра РФ. Таким образом, сдвигатель всегда возвращает 32-битный результат.

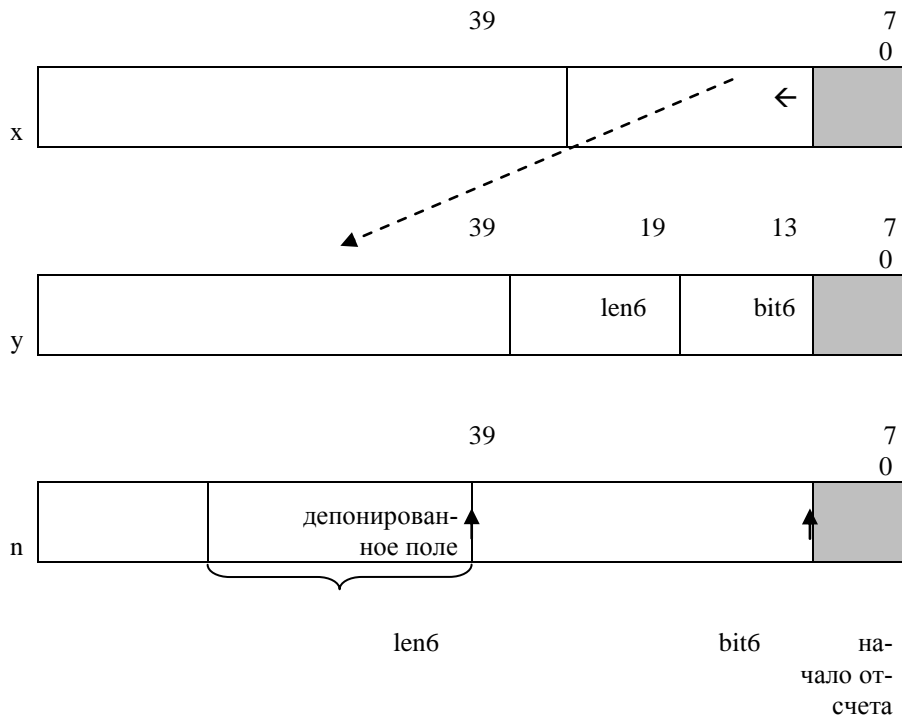
3.4.2. Операции выделения и депонирования битовых полей

Инструкции депонирования и выделения битовых полей сдвигателя позволяют манипулировать с группой битов внутри 32-битного поля ФЗ-величины. Инструкция депонирования помещает битовое поле нужной длины в заданную позицию регистра результата. Инструкция выделения выделяет в операнде битовое поле указанной длины и помещает их в регистр результата.

Y-операнд для этих инструкций определяет две 6-битные величины: поле bit6 (биты 13÷8) и поле len6 (биты 19÷14), интерпретируемые как положительные целые значения. Поле bit6 является позицией начального бита для де-

понирования или выделения, а поле $lenb$ — длиной битового поля, которое определяет количество выделяемых или депонируемых битов.

На рис. 3.3 показаны схемы работы инструкций депонирования и выделения полей.



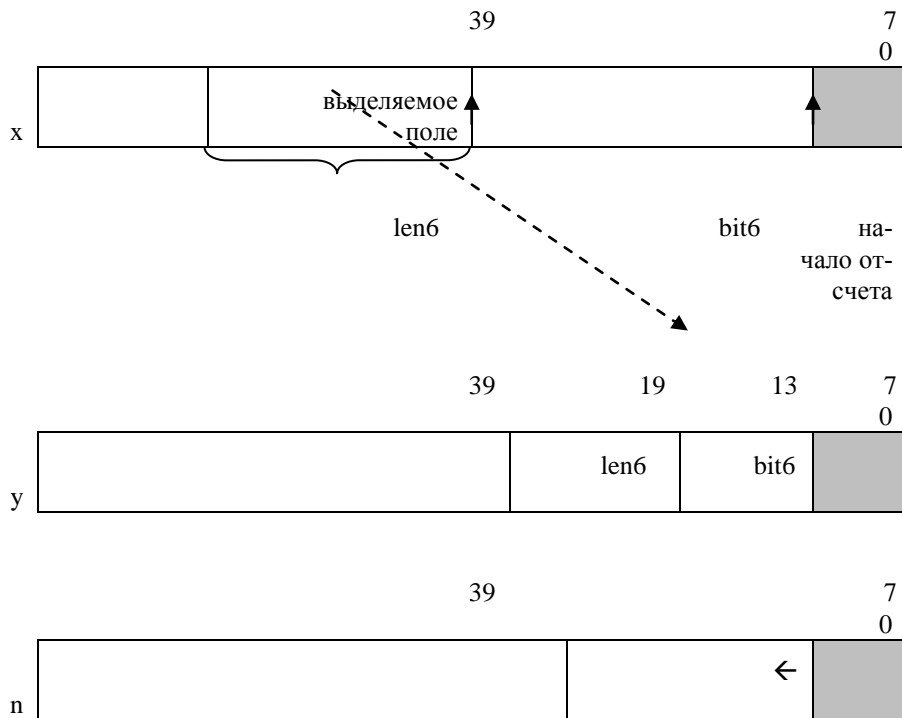


Рис.3.3. Схема выполнения операций депонирования и выделения полей

Значения bit6 и len6 позволяют выполнять выделение и депонирование полей длиной от 0 до 32 битов начиная с позиции от 0 до 32 (т.е. за пределами ФЗ-поля).

3.4.3. Флаги сдвигателя

Сдвигатель в конце операции устанавливает три флага состояния в регистре ASTAT.

Б ит	Им я	Определение
1 1	SV	переполнение
1 2	SZ	нулевой результат
1 3	SS	знак операнда (только для операции определения экспоненты)

3.4.4. Инструкции сдвигателя

Инструкции сдвигателя и их действие на флаги описаны в приложении 3.

3.5. Многофункциональные вычисления

ADSP обеспечивает многофункциональные вычисления, которые сочетают параллельные операции MAC-а и АЛУ или дуальные функции АЛУ, исполняемые за один такт. Совмещённые операции выполняются аналогично обычному исполнению, флаги определяются так же. Исключение составляют флаги АЛУ для дуального сложения (вычитания), которые устанавливаются вместе по логическому “или”. Любые из четырех входных операндов в многофункциональных вычислениях, использующих MAC и АЛУ, должны указываться как регистры в четырех различных группах регистрового файла: $0 \div 3/4 \div 7/8 \div 11/12 \div 15$.

Дуальное сложение/вычитание:

$$R_a = R_x + R_y, \quad R_s = R_x - R_y;$$

$$F_a = F_x + F_y, \quad F_s = F_x - F_y;$$

ФЗ-умножение с накоплением и сложение, вычитание или вычисление среднего:

$$\left| \begin{array}{l} R_m = R_{3-0} * R_{7-4} \text{ (SSFR)} \\ MRF = MRF + R_{3-0} * R_{7-4} \text{ (SSF)} \\ R_m = MRF + R_{3-0} * R_{7-4} \text{ (SSFR)} \\ MRF = MRF - R_{3-0} * R_{7-4} \text{ (SSF)} \\ R_m = MRF - R_{3-0} * R_{7-4} \text{ (SSFR)} \end{array} \right|, \quad \left| \begin{array}{l} R_a = R_{11-8} + R_{15-12} \\ R_a = R_{11-8} + R_{15-12} \\ R_a = (R_{11-8} + R_{15-12}) / 2 \end{array} \right|$$

ПЗ-умножение и операции АЛУ:

$$F_m = F_{3-0} * F_{7-4}, \quad \left| \begin{array}{l} F_a = F_{11-8} + F_{15-12} \\ F_a = F_{11-8} - F_{15-12} \\ F_a = \text{FLOAT } R_{11-8} \text{ by } R_{15-12} \\ F_a = \text{FIX } F_{11-8} \text{ by } R_{15-12} \\ F_a = (F_{11-8} + F_{15-12}) / 2 \\ F_a = \text{ABS } F_{11-8} \\ F_a = \text{MAX } (F_{11-8}, F_{15-12}) \\ F_a = \text{MIN } (F_{11-8}, F_{15-12}) \end{array} \right|$$

Умножение и дуальное сложение/вычитание:

$$R_m = R_{3-0} * R_{7-4} \text{ (SSRF)}, \quad R_a = R_{11-8} + R_{15-12}, \quad R_s = R_{11-8} - R_{15-12}$$

$$F_m = F_{3-0} * F_{7-4}, \quad F_a = F_{11-8} + F_{15-12}, \quad F_s = F_{11-8} - F_{15-12}$$

R_m, R_a, R_x, R_y – любая ФЗ-ячейка РФ, **F_m, F_a, F_x, F_y** – любая ПЗ-ячейка РФ. **SSRF** – X- и Y – знаковые дробные входные операнды, результат округлен к ближайшему; **SSF** – X- и Y – знаковые дробные входные операнды, результат не округляется.

3.6. Пример

Написать подпрограмму на языке ассемблера для аппроксимации функции синус по формуле

$$\sin(x) = 3.140625*x + 0.02026367*x^2 - 5.325196*x^3 + 0.5446778*x^4 + 1.800293*x^5$$

Написать программный код для вызова подпрограммы.

Файл описания архитектуры:

```
.system example;
.processor ADSP21020
.segment /ram /begin=0x00008 /end=0x0000f /pm pm_rsti;
.segment /ram /begin=0x00100 /end=0x001ff /pm pm_code;
.segment /ram /begin=0x00200 /end=0x00300 /pm pm_sinus;
.segment /ram /begin=0x00000 /end=0x0000f /dm sin_coef;
.segment /ram /begin=0x00010 /end=0x00020 /dm dm_val;
.endsys;
```

Текст файла sin.asm:

```
.GLOBAL    sinus;          ! делаем метку sinus глобальной
! сегмент данных - констант аппроксимации
.SEGMENT/DM    sin_coef;
.VAR D[5]=3.140625, 0.02026367, -5.325196, 0.5446778, 1.800293;
.ENDSEG;
! сегмент кода - содержит подпрограмму sinus
.SEGMENT/PM    pm_sinus;
! Входные параметры: F0 - x;
! Выходные параметры: F1 - sin(x)
! Используемые регистры: I0, M0, F0-F6, F8-F12, F14, F15
! Время выполнения подпрограммы sinus - 15 тактов
sinus:
    I0 = D;
    M0 = 1;
```

```

! получение степеней x и загрузка коэффициентов
F14 = PASS F0, F8 = DM(I0,M0);      ! F14 = x      F8 = C1
F15 = F0*F0, F12 = DM(I0,M0);      ! F15 = x2     F12 = C2
F3 = F15*F0, F4 = DM(I0,M0);      ! F3 = x3      F4 = C3
F2 = F3*F0, F5 = DM(I0,M0);      ! F2 = x4      F5 = C4
F1 = F2*F0, F6 = DM(I0,M0);      ! F1 = x5      F6 = C5
! перемножение x на коэффициенты со сложением
F8 = F14*F8;                        ! F8=C1*x
F12 = F15*F12;                      ! F12=C2*x2
F9 = F3*F4, F12 = F8+F12;      ! F9=C3*x3   F12=C1*x + C2*x2
! F11=C4*x4   F12=C1*x + C2*x2 + C3*x3
F11 = F2*F5, F12 = F9+F12;
    rts (db);
! F1=C5*x5   F12=C1*x + C2*x2 + C3*x3 + C4*x4
F1 = F1*F6, F12 = F11+F12;
! F1=C1*x + C2*x2 + C3*x3 + C4*x4 + C5*x5
F1 = F1+F12;
.ENDSEG;

```

Текст файла lab3.asm:

```

#include <def21020.h>
.EXTERN sinus;                ! объявляем внешнюю метку
! сегмент, содержащий исходное и результирующее значения
.SEGMENT/DM dm_val;
.VAR src_val=2.0;             ! проверяемое значение x
.VAR dst_val;
.ENDSEG;
! сегмент кода для начала работы

```



```

.SEGMENT/PM pm_rsti;
JUMP start (DB);          ! переход по сбросу с задержкой
DMWAIT=0x21;              ! установить параметры памяти
PMWAIT=0x21;
.ENDSEG;
! сегмент кода для демонстрации вызова подпрограммы sinus
.SEGMENT/PM pm_code;
start:
    call sinus (db);
    F0 = DM(src_val);
    nop;
    DM(dst_val) = F1;
    idle;
.ENDSEG;

```

3.7. Варианты заданий

Написать программы для вычисления арифметического или логического выражения:

1. Аппроксимация функции синус по формуле, приведенной в примере, но с минимизацией количества используемых регистров за счет увеличения числа тактов.
2. Построить произвольную интерполяционную функцию (многочлен) по четным точкам ряда и рассчитать среднюю ошибку интерполяции.
3. Переставить биты в обратном порядке в массиве слов.
4. Переставить тетрады в обратном порядке в массиве слов.
5. Найти сумму попарных произведений соседних элементов массива.
6. Найти сумму членов ряда, заданного выражением:

$$\sum_{i=1}^{10} \frac{(-1)^{i+1} (2\pi x_i)}{(-2)^{i-1} ((2\pi+1)x_i)}$$

3.8. Контрольные вопросы

1. Какие вычислительные блоки входят в состав процессоров семейства ADSP-21000?
2. Какие основные форматы данных поддерживают процессоры семейства ADSP-21000?
3. Что такое регистровый файл? Из каких регистров он состоит?
4. Что такое теневые (альтернативные) регистры?
5. Какие режимы работы есть в АЛУ, умножителе и сдвигателе?
6. Что такое умножение с накоплением? Опишите структуру аккумулятора умножителя.
7. Как работают команды выделения и депонирования битовых полей?
8. Что представляют собой биты накапливающегося сравнения в регистре ASTAT?
9. Почему в приведенном выше примере использована инструкция PASS вместо обычной инструкции присваивания регистров?

4. УПРАВЛЕНИЕ ПРОГРАММОЙ В ПРОЦЕССОРАХ СЕМЕЙСТВА ADSP-21000

4.1. Программный секвенсор ADSP-21000

4.1.1. Структура программного секвенсора

Изменение линейной структуры программы на языке ассемблера достигается путем использования циклов, под-программ, переходов, прерываний, инструкции IDLE (специальная инструкция, которая "замораживает" процессор, сохраняя его текущее состояние до какого-либо прерывания).

Процессоры семейства ADSP-21000 выполняют инструкцию за три цикла:

- в цикле получения данных процессор из памяти программ (или из внутреннего кэша команд) читает инструкцию, адрес которой содержится в регистре программного секвенсора (ПС) FADDR;
- в течение цикла декодирования инструкция, адрес которой содержится в регистре ПС DADDR, расшифровывается, генерируя условия, которые управляют выполнением инструкции;
- в цикле выполнения процессор исполняет инструкцию, адрес которой находится в регистре ПС программного секвенсора.

Эти циклы частично перекрываются, образуя конвейерное выполнение команд, как показано на рис. 4.1.

Время (циклы)	1	2	3	4	→5
Выпол- нение (PC)	0 x0108	0x 0109	0x 010A	0x 010B	0x 010C
Рас- шифровка (DADDR)		0x 0108	0x 0109	0x 010A	0x 010B
Выбор- ка (FADDR)			0x 0108	0x 0109	0x 010A

Рис.4.1. Конвейеризованные циклы выполнения

Программный секвенсор определяет адрес следующей инструкции путем исследования текущей исполняемой инструкции и текущего состояния процессора. Если никакие условия не требуют иного, то ADSP-21000 выполняет команды из памяти программ в последовательном порядке просто инкрементируя адрес выборки.

Ветвление происходит тогда, когда адрес выборки не является следующим по счету за адресом предыдущей выборки. ADSP-21000 поддерживает условные и безусловные переходы, вызовы подпрограмм и возвраты. Единственным различием между переходом и вызовом подпрограммы (с точки зрения ПС) является то, что при вызове подпрограммы адрес возврата проталкивается в стек ПС, так что он становится доступен, когда позже выполняется команда возврата. Переходы же направляют выполнение программы по другой ветви, не допуская автоматического возврата.

Для выполнения перехода используется команда JUMP, для вызова подпрограммы – команда CALL. Для возврата из подпрограммы используется команда RTS. Например:

```
...
R1=DM(I1,0); R2=PM(I8,M12); ! чтение из памяти в 1 такте
CALL Plus;                  ! вызов подпрограммы
DM(I1,0) = R3;               ! запись результата в память
...
! тело подпрограммы Plus
Plus:      R3 = R1 + R2;      ! суммирование регистров
RTS;       ! возврат из подпрограммы
! окончание подпрограммы Plus
...
```

Переходы, вызовы подпрограмм и возвраты могут быть условными и безусловными. Если никакое из условий не определено, то переход всегда выполняется.

Переходы и вызовы подпрограмм могут быть косвенными, прямыми и относительными. Косвенный переход отправляет к адресу, содержащемуся в генераторе адреса данных DAG2. Прямые переходы отправляют к 24-битному адресу, определенному непосредственно в поле инструкции перехода. Относительные переходы также используют значение, определенное в инструкции, но секвенсор, чтобы вычислить адрес, прибавляет это значение к текущему содержанию ПС.

Например:

```
JUMP    next_label;           ! прямой переход
CALL (PC, 0x0002);           ! относительный переход
IF EQ   JUMP (M8, I12);       ! косвенный переход
next_label: ...
```

Переходы и возвраты могут быть с задержкой и без задержки.

4.1.2. Переходы с задержкой и без задержки

Наличие управляющего параметра после команды перехода (DB) показывает, что выполняется переход с задержкой. В этом случае процессор продолжает выполнять еще две инструкции, пока инструкция по адресу перехода выбирается и декодируется; в случае вызова подпрограммы адрес возврата является третьим после инструкции перехода (рис. 4.2).

Время				
↑ PC	n	n+1	N+2	j
↑ DA DDR	n+1	n+2	j	j+1
FA DDR	n+2	j	J+1	j+2
	команда CALL на адрес i	в стек ПС проталкивается (n+3)		

Рис.4.2. Выполнение инструкции задержанного перехода

Если же выполняется переход без задержки (отсутствует параметр (DB) после команды перехода), то две инструкции после инструкции перехода, которые находятся на стадии декодирования и выборки, не выполняются; для вызова подпрограммы адрес возврата является адресом следующей после команды перехода инструкции (рис. 4.3).

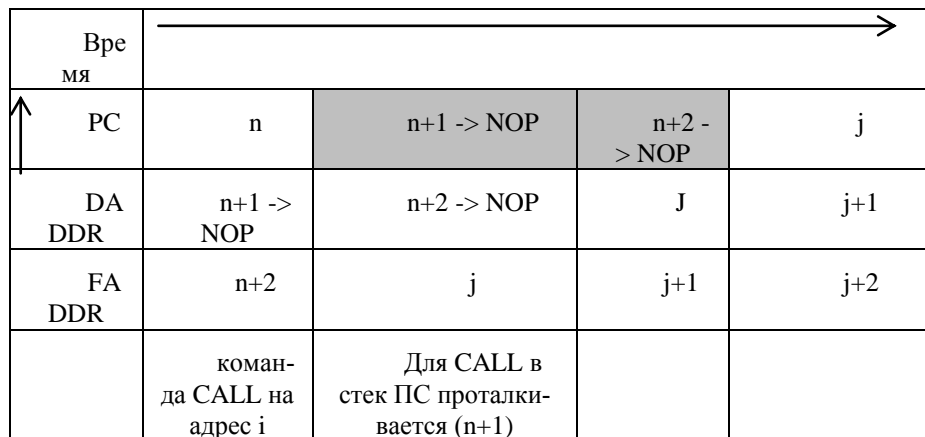


Рис. 4.3. Выполнение инструкции незадержанного перехода (серым цветом показаны циклы "потерь времени")

Из-за конвейеризации инструкций инструкция задержанного перехода и следующие за ней две инструкции должны выполняться последовательно. Поэтому инструкциями, непосредственно следующими за инструкцией задерживаемого перехода, не могут быть:

- другие команды переходов или возвратов (JUMP, CALL, RTS, RTI);

- команды записи и чтения из стека ПС (PUSH и POP);
- команды записи в стек ПС или изменения указателя стека ПС;
- инструкция цикла DO...UNTIL;
- инструкция IDLE.

Любое прерывание, происшедшее во время выполнения задерживаемого перехода, защелкивается, но не обрабатывается, пока не завершится переход.

4.1.3. Стек программного секвенсора

Стек ПС содержит адреса возврата для подпрограмм и обработчиков прерываний, а также адреса вершин циклов (см. ниже). Глубина стека ПС равна 20 элементам.

При вызове обработчика прерывания, подпрограммы, входе в тело цикла адрес возврата (или вершины цикла) заносится в вершину стека ПС. При возврате из прерываний (инструкция RTI), из подпрограмм (инструкция RTS) и при прерываниях циклов из стека ПС выталкивается верхнее значение.

4.2. Инструкции условных переходов

Программный секвенсор вычисляет условия, чтобы определить, выполнять ли условную инструкцию. Условия основываются на информации из регистра арифметического состояния ASTAT, регистра управления режимом MODE1, на входных флагах и счетчике цикла. Каждое условие имеет ассемблерную мнемонику (табл. 4.1).

Таблица 4.1

Ус- ловие	Опреде- ление	Истинно, если	Обрат- ное условие
EQ	АЛУ, рав- но нулю	$AZ = 1$	NE
LT	АЛУ, меньше нуля	$\left[\overline{AF} \& \left(AN \oplus \left(AV \& \overline{ALUSAT} \right) \right) \vee \left(AF \& AN \& \overline{AZ} \right) \right] = 1$	GE
LE	АЛУ, меньше либо равно нулю	$\left[\overline{AF} \& \left(AN \oplus \left(AV \& \overline{ALUSAT} \right) \right) \vee \left(AF \& AN \right) \right] \vee AZ$	GT
AC	АЛУ, пе- ренос	$AC = 1$	NOT FC
AV	АЛУ, пе- реполнение	$AV = 1$	NOT AV
MV	МАС, пе- реполнение	$MN = 1$	NOT MV
MS	МАС, меньше нуля	$MN = 1$	NOT MS
SV	SHIFTER, переполнение	$SV = 1$	NOT SV

SZ	SHIFTER, равно нулю	SZ = 1	NOT SZ
FLA G0_IN	Флаг 0, ввод	FI0 = 1	NOT FLAG0_IN
FLA G1_IN	Флаг 1, ввод	FI1 = 1	NOT FLAG1_IN
FLA G2_IN	Флаг 2, ввод	FI2 = 1	NOT FLAG2_IN
FLA G3_IN	Флаг 3, ввод	FI3 = 1	NOT FLAG3_IN
TF	Флаг про- верки битов	BTF = 1	NOT TF
LCE	Цикл за- кончен	CURLNTR = 1	
NOT LCE	Цикл не закончен	CURLNTR <> 1	
FOR EVER	Всегда ложь	Всегда	
TRU E	Всегда ис- тина	Всегда	

4.3. Циклы

Для организации циклов в ADSP-21000 используется команда DO ... UNTIL. При выполнении этой команды программный секвенсор помещает адрес последней инструкции цикла и условие прерывания для выхода из цикла (обе компоненты определяются в инструкции DO...UNTIL) в стек адреса цикла, а адрес вершины цикла – в стек ПС.

Из-за конвейеризации процессор проверяет условие прерывания цикла (и если цикл основан на счетчике, декрементирует счетчик) перед концом цикла, чтобы следующая выборка либо возвращала к началу цикла, читая адрес с вершины стека ПС, либо переходила к инструкции, следующей за циклом, выталкивая при этом значения из стека цикла и стека ПС.

Инструкция безусловного перехода JUMP с параметром (LA) вызывает автоматическое прерывание цикла, если она находится в теле цикла (подобно команде break в языке C).

4.3.1. Стеки циклов

Стек адреса цикла разрядностью 32 бита имеет в глубину шесть уровней. Каждое слово в стеке адреса цикла содержит следующие поля:

Биты	Содержание
0 – 23	адрес окончания цикла (адрес последней инструкции цикла)
24 – 28	код завершения
30 – 31	тип цикла: 00 – по условию 01 – по счётчику (длина цикла – 1 инструкция) 10 – по счётчику (длина цикла – 2 инструкции) 11 – по счётчику (длина цикла больше 2-х инструкций)

Регистр LADDR содержит вершину стека адреса цикла. Чтение и запись LADDR не изменяет содержимое стека. Когда стек адреса цикла пуст, LADDR содержит значение 0xFFFF FFFF.

Стек счётчика цикла имеет глубину 6 уровней 32-битных слов и работает синхронно со стеком адреса цикла. Это значит, что оба стека имеют одинаковое число занятых мест.

Программный секвенсор оперирует с двумя отдельными счётчиками цикла: текущим счётчиком CURLCNTR, который отсчитывает итерации для цикла, выполняемого в текущий момент, и содержит значение вершины стека счётчика цикла; и просто счётчиком LCNTR, который содержит значение перед тем, как цикл начнёт выполняться. Инструкция DO ... UNTIL LCE (цикл по счётчику) проталкивает значение LCNTR в стек счётчика цикла, чтобы установить новые значения CURLCNTR. Предыдущее значение CURLCNTR сохраняется в стеке на одну позицию ниже. Если CURLCNTR = 0xFFFF FFFF, то стек цикла пуст.

4.3.2. Ограничения при использовании циклов

Ограничения определяются 3-ступенчатой конвейеризацией в процессорах семейства ADSP-21000:

- последние три инструкции цикла не могут быть любыми переходами, за исключением перехода с прекращением цикла с модификатором (LA);
- вложенные циклы не могут прерываться на 1 и той же инструкцией;
- для предотвращения пустых избыточных тактов циклы единичной длины данных выполняются минимум три раза, а двоичной длины - два раза;
- если цикл, основанный не на счетчике, является вложенным, то адрес окончания внешнего цикла должен располагаться по крайней мере на два адреса после адреса окончания внутреннего цикла.

4.3.3. Циклы, основанные не на счетчике

Для правильного функционирования цикла, основанного не на счетчике, необходимо, чтобы его длина была не менее трех инструкций, так как условие выхода проверяется за две инструкции до конца цикла. В противном случае количество итераций цикла может оказаться неверным. Поэтому обычно используют бесконечный цикл с командой условного перехода JUMP с модификатором (LA) для своевременного выхода из цикла.

4.4. Пример

Написать программу перемножения двух квадратных целочисленных матриц, расположенных в памяти построчно.

Файл описания архитектуры:

```
.system example;
.processor ADSP21020
.segment /ram /begin=0x00008 /end=0x0000f /pm pm_rsti;
.segment /ram /begin=0x00100 /end=0x00200 /pm pm_code;
.segment /ram /begin=0x00400 /end=0x00500 /dm dm_data1;
.segment /ram /begin=0x00400 /end=0x00500 /pm pm_data2;
.segment /ram /begin=0x00600 /end=0x00700 /dm dm_data3;
```

```
.endsys;
```

Текст программы:

```
#include <def21020.h>
#define F 4                      ! размер строки (столбца матрицы)
#define FF 16                    ! размер матрицы
! сегмент в памяти данных для хранения матрицы 1 по строкам
.SEGMENT/DM dm_data1;
.VAR matrix1[FF]="t_1.dat";
.ENDSEG;
! сегмент в памяти программ для матрицы 2 по строкам
.SEGMENT/PM pm_data2;
.VAR matrix2[FF]="t_2.dat";
.ENDSEG;
! сегмент в памяти данных для хранения матрицы результата ! по строкам
.SEGMENT/DM dm_data3;
.VAR matrix3[FF];
.ENDSEG;
! сегмент обработки прерывания сброса
.SEGMENT/PM pm_rsti;
JUMP start (DB);                ! переход по сбросу с задержкой
DMWAIT=0x21;                    ! установить параметры памяти
PMWAIT=0x21;
.ENDSEG;
! Сегмент кода
.SEGMENT/PM pm_code;
start:
! матрица 1 - идем по строкам
```

```

R1 = matrix1;      ! начало очередной строки матрицы 1
L1 = F;
M1 = 1;
R4 = F;            ! для перехода на следующую строку в матр.1
! матрица 2 - идем по столбцу
L8 = @matrix2;
M8 = F;            ! прыгаем через F элементов на следующую строку
! матрица результата - заполняется по строкам
B3 = matrix3;
M3 = 1;
! внешний цикл по строкам
LCNTR=F, DO rows UNTIL LCE;
    B1 = R1;        ! на след. строку в матр.1
    B8 = matrix2;   ! на начало матрицы 1
    ! вложенный цикл по всем столбцам данной строки
    LCNTR=F, DO columns UNTIL LCE;
        MRF = 0, R0=DM(I1,M1), R2=PM(I8,M8);
        ! рассчитать очередной элемент
        LCNTR=F, DO inner UNTIL LCE;
inner: MRF = MRF+R0*R2 (SSI), R0=DM(I1,M1), R2=PM(I8,M8);
        R3 = MR0F;
        DM(I3,M3)=R3;
        MODIFY(I1,-1);      ! к началу той же строки в матр.1
        MODIFY(I8,-F);      !к началу того же столбца в матр.2
columns: MODIFY(I8,1);      ! на след. столбец матрицы 2
rows:
    R1 = R1 + R4;           ! на следующую строку матрицы 1
idle;

```

.ENDSEG;

Время выполнения фрагмента программы с метки start (при $F > 2$) составляет:

$$12 + F(4 + F(7 + F)) = (F^3 + 7F^2 + 4F + 12) - \text{тактов.}$$

4.5. Варианты заданий

Написать программу для выполнения одного из следующих заданий и оценить время ее работы (в тактах).

1. Сортировка элементов вектора.
2. Сортировка элементов матрицы.
3. Перемножение неквадратных матриц.
4. Транспонирование квадратной матрицы.
5. Формирование вектора из всех положительных (отрицательных) элементов матрицы.
6. Формирование обратной матрицы.

4.6. Контрольные вопросы

1. Какова разрядность ячеек каждого из стеков программного секвенсора?
2. Сколько вложенных циклов может быть в программе?
3. Как изменяется содержимое стеков ADSP-21000 при выполнении следующего фрагмента кода:

```
LCNTR = 10;  
DO label1 UNTIL LCE;  
    MRF = 0;  
    LCNTR = 20;  
    DO label2 UNTIL LCE;  
label2: MRF=MRF+R1*R2(SS1), R1=DM(I1,M1), R2=PM(I8,M8);  
    R3 = MR0F;  
    DM(result1) = R3;  
    PM(I10, 0) = R3;  
label1: MODIFY (I10,1);
```

RTS;

4. Каковы потери производительности при выполнении команд условного и безусловного перехода с задержкой и без задержки?
5. В чем отличие выполнения команды перехода и команды вызова подпрограммы?
6. Какие ограничения существуют при организации циклов?
7. Как изменится время выполнения программы, приведенной в примере, при $F < 2$? Будет ли она работоспособной?
8. Обоснуйте приведенную для примера оценку длительности его выполнения в тактах.

5. ОБРАБОТКА ПРЕРЫВАНИЙ В ПРОЦЕССОРАХ СЕМЕЙСТВА ADSP-21000

5.1. Работа таймера

Процессоры семейства ADSP-21000 имеют программируемый интервальный таймер, который может генерировать периодические прерывания. Программируется таймер записью в два универсальных 32-разрядных регистра: TPERIOD (регистр периода) и TCOUNT (регистр счетчика). Таймер декрементирует TCOUNT в каждом тактовом цикле. Когда значение TCOUNT становится нулевым, таймер генерирует два прерывания таймера (высокого и низкого приоритета) и автоматически перезагружает TCOUNT значением из TPERIOD.

Содержимое регистра TPERIOD задает частоту прерываний от таймера: количество циклов между прерываниями равно $TPERIOD + 1$.

Запуск и останов таймера осуществляется управлением битом TIMEN в регистре MODE2: если $TIMEN = 1$, то

таймер включен, иначе – выключен. Перед включением таймера необходимо инициализировать оба регистра TPERIOD и TCOUNT.

5.2. Обработка прерываний

Вызов прерывания происходит либо при запросе внешним устройством по одному из входов прерывания (IRQ_{3-0}), либо при возникновении какого-либо внутреннего события, например переполнения стека, прерывания таймера, пользовательского прерывания. Прерывание приводит к передаче управления на заранее определенный адрес – подпрограмму обработки прерывания. Процессор семейства ADSP-21000 назначает уникальный адрес для обработчика каждого распознаваемого прерывания. На один обработчик отводится восемь инструкций; более длинные обслуживающие программы могут быть приспособлены для перехода в другую область памяти. При вызове прерывания в стек ПС помещается адрес возврата. Если произошло внешнее прерывание или прерывание от таймера, то дополнительно к этому в стек состояния помещаются текущие значения регистров ASTAT и MODE1. Возвращение из прерывания осуществляется командой RTI, при выполнении которой из стека выталкивается адрес возврата (если выполнялась обработка внешнего прерывания или прерывания от таймера – то выталкиваются и значения регистров ASTAT и MODE1 из стека состояния).

5.2.1. Защелкивание прерываний

Процессор ADSP-21000 реагирует на прерывание в три этапа, каждый из которых занимает один цикл: синхронизация и защелкивание, распознавание, переход по вектору прерывания (рис. 5.1).


Вре мя					
PC	n-1	N	NOP	N OP	V
DA DDR	n	n+1 -> NOP	n+2 -> NOP	V	v +1
FA DDR	n+1	n+2	v	v +1	v +2
	про- изошло прерыва- ние	распо- знавание	n+1 протал- кивается в стек ПС, чтение первой команды обработ- чика		

Рис. 5.1. Обработка прерывания

32-разрядный регистр IRPTL защелкивает (фиксирует) прерывания, генерируемые внешним или внутренним событием. Этот регистр содержит все текущие обрабатываемые и ожидающие обработки прерывания. Поля регистра IRPTL (табл.5.1) соответствуют адресам расположения обработчиков прерывания в памяти.

Таблица 5.1

Б ит	Адрес обработчи- ка	Им я	Функция
0	0x00	—	зарезервировано для эмуляции
1	0x08	RS TI	сброс (только для чтения)
2	0x10	—	зарезервировано
3	0x18	SO VFI	переполнение стека состояний или цикла или наполненность стека программ- ного секвенсора
4	0x20	TM ZHI	прерывание от таймера высокого при- оритета
5	0x28	IRQ 3I	есть сигнал по линии IRQ 3
6	0x30	IRQ 2I	есть сигнал по линии IRQ 2
7	0x38	IRQ 1I	есть сигнал по линии IRQ 1
8	0x40	IRQ 0I	есть сигнал по линии IRQ 0

9	0x48	—	зарезервировано
10	0x50	—	зарезервировано
11	0x58	CB 7I	переполнение кругового буфера 7
12	0x60	CB 15I	переполнение кругового буфера 15
13	0x68	—	зарезервировано
14	0x70	TM ZLI	прерывание от таймера низкого приоритета
15	0x78	FIX I	переполнение при операциях с ФЗ
16	0x80	FL TOI	переполнение при операциях с ПЗ
17	0x88	FLI UI	потеря значимости при операциях с ПЗ
18	0x90	FL TII	некорректная операция с ПЗ

1 9-23	0x98 – 0xB8	— - —	зарезервировано
2 4-31	0xC0 – 0xF8	SFT OI – SFT7I	пользовательские прерывания 0 – 7

Биты прерывания в IRPTL расположены в порядке убывания приоритета. Этот приоритет определяет, какое прерывание обслуживается первым, если два прерывания происходят в одном цикле. Если прерывание инициируется установкой бита в регистре STKY (при выполнении арифметических операций, переполнении стеков, переполнении кругового буфера), программа обработки прерывания должна очищать соответствующий бит STKY, чтобы не произошел повторный вызов прерывания.

5.2.2. Маскирование прерываний

Все прерывания могут быть разрешены и запрещены глобальным битом разрешения прерываний IRPREN (бит 12 в MODE1). Для глобального разрешения прерываний он должен быть установлен. Все прерывания, за исключением прерывания сброса, могут быть замаскированы. Биты регистра управления маскирования прерываний IMASK соответствуют битам регистра защелки прерываний IRPTL. Если бит = 1, соответствующее прерывание не замаскировано, то есть разрешено. Регистр IMASK препятствует только обработке прерываний, но не их защелкиванию в регистре IRPTL.

5.2.3. Вложенность прерываний

Обслуживающая программа может быть прервана приходом прерывания более высокого приоритета. Когда бит режима вложенности NESTM (в регистре MODE1) равен нулю, обслуживающая программа не может прерваться и любое пришедшее прерывание будет обработано только после ее завершения. В противном случае незамаскированные прерывания с более высоким приоритетом могут прервать выполнение обслуживающей программы.

В режиме вложенности ADSP-21000 использует указатель маски прерывания (регистр IMASKP) чтобы создавать

временную маску прерываний для каждого уровня вложенности. При поступлении нового прерывания, если вложенность разрешена, генерируется новая временная маска IMASKP, запрещающая все прерывания с равным или более низким приоритетом, путем переноса из регистра IMASK только битов, соответствующих прерываниям более высокого приоритета, чем поступившее.

При возврате из обслуживающей программы бит обслуживающего прерывания в IMASKP сбрасывается и снова генерируется новая маска прерываний путем маскирования всех прерываний равного и низшего приоритетов относительно бита самого высокого прерывания из установленных в IMASKP.

ADSP-21000 игнорирует и не защелкивает прерывания, происходящие, когда обслуживающая его программа уже выполняется.

5.2.4. Обработка внешних и программных прерываний

Каждое из четырех внешних прерываний IRQ_{3-0} может защелкиваться или по уровню, или по фронту. Прерывание по уровню считается действительным, если при проверке соответствующего входа определяется состояние "активно" (низкий уровень). Прерывание по уровню должно быть деактивировано прежде, чем произойдет возврат из обслуживающей подпрограммы, в противном случае процессор будет трактовать активный уровень как новый запрос.

Прерывание по фронту считается действительным, если в одном цикле фиксируется высокий уровень, а в другом – низкий. Для прерываний по фронту нет необходимости снимать запрос.

Режим чувствительности для каждого входа IRQ_{3-0} определяется соответствующим битом в регистре MODE2.

Инициирование программного (пользовательского) прерывания выполняется установкой одного из битов 24-31 в регистре IRPTL. ADSP-21000 обслуживает это прерывание аналогично другим видам прерываний.

5.3. Пример

Написать программу для выполнения фильтрации целочисленных входных отсчетов с использованием фильтра порядка F (для простоты F является степенью 2) с постоянными целочисленными коэффициентами. Определить максимальную частоту поступления входных отсчетов.

Файл описания архитектуры:

```
.system example;
.processor ADSP21020
.segment /ram /begin=0x00008 /end=0x0000f /pm pm_rsti;
.segment /ram /begin=0x00020 /end=0x00027 /pm timer_hi;
.segment /ram /begin=0x00100 /end=0x00200 /pm pm_code;
.segment /ram /begin=0x00800 /end=0x00900 /pm pm_data;
.segment /ram /begin=0x00000 /end=0x000ff /dm dm_data;
.segment /port /begin=0x00100 /end=0x00100 /dm dm_port1;
.segment /port /begin=0x00101 /end=0x00101 /dm dm_port2;
.endsys;
```

Текст программы:

```
#include <def21020.h>
#define F 4 ! порядок фильтра
! сегмент в памяти данных для хранения F входных отсчетов
.SEGMENT/DM dm data;
.VAR in_buffer[F]; ! буфер для очередных F элементов
.ENDSEG;
! сегмент для хранения F коэффициентов фильтра (const)
.SEGMENT/PM pm_data;
.VAR coefs[F]="t_2.dat"; ! коэффициенты фильтра
.ENDSEG;
! сегмент для обработки прерывания сброса
.SEGMENT/PM pm rsti;
    JUMP start (DB); ! переход по сбросу с задержкой
    DMWAIT=0x21; ! установить параметры памяти
    PMWAIT=0x21;
.ENDSEG;
```

```

! сегмент обработки высокоуровневого прерывания от
! таймера (6 циклов)
.SEGMENT/PM timer_hi;
! считать из порта и выполнить свертку для F элементов
  call ComputeFilter (DB);      ! переход на обработчик
  R3 = DM(res_in);              ! загрузить значение
  DM(I7, 1) = R3;               ! записать в массив
  ! возврат из прерывания с очередного значения в порт
  rti (DB);
  DM(res_out) = R0;             ! записать в порт
  nop;
.ENDSEG;
! сегменты описания входного и выходного портов
.SEGMENT/DM dm_port1;
.VAR res_in;                    ! входной порт
.ENDSEG;
.SEGMENT/DM dm_port2;
.VAR res_out;                   ! выходной порт
.ENDSEG;
! сегмент инициализации переменных (выполняется 1 раз)
.SEGMENT/PM pm_code;
start:                          ! указатели
  B7 = in_buffer;               ! базовый адрес входного буфера
  L7 = @in_buffer;              ! круговой буфер
  M7 = 1;                       ! смещение указателя
! указатели для чтения значений из входного буфера
  B1 = in_buffer;
  I1 = in_buffer+1;             ! чтобы новое значение было

```



```

                                ! последним в свертке
L1 = @in_buffer;                ! круговой буфер
M1 = 1;

! указатели для доступа к коэффициентам фильтра
B15 = coefs;
L15 = @coefs;                    ! круговой буфер
M15 = 1;

! делитель в операции свертки
R4 = F;
R4 = LEFTZ R4;
R5 = 31;
R4 = R4-R5;                      ! фактор сдвига

! установки для таймера
TPeriod = 25;
TCount = 25;

! установка прерываний
bit set IMASK TMZHI; ! разрешить прерывания от таймера
bit set MODE2 TIMEN; ! включить таймер
bit set MODE1 IRPTEN; ! разрешить прерывания
! постоянное ожидание, прерываемое таймером
wait: idle;
    jump wait;                ! после обработки прерывания -
                                ! вернуться на idle

! процедура получения очередного выходного значения по F ! входным значениям
! время выполнения - (6+F) циклов
ComputeFilter:
! загрузить первые значения

```

```

MRF=0, R2=DM(I1,M1), R1=PM(I15,M15);
LCNTR=F, DO outsample UNTIL LCE;          ! цикл - F раз
outsample:
    MRF=MRF+R2*R1 (UUI), R2=DM(I1,M1), R1=PM(I15,M15);
    R2 = MR0F;                             ! запомнить результат
    rts (DB);                               ! выйти с задержкой
    R0 = ASHIFT R2 BY R4;                   ! разделить на F
    modify(I15,-1);                         ! вернуться к началу фильтра
.ENDSEG;

```

Минимально допустимое число тактов между приходами двух входных отсчетов составляет $(18+F)$ тактов. Таким образом, при длительности такта 30нс (ADSP-21020) и порядке фильтра F, равном 4, получаем предельно допустимую частоту входных отсчетов: $1/((18+4)*30^{-9})=1,52$ МГц. Так как один входной отсчет соответствует одному выходному отсчету, то частота выходных отсчетов будет такой же.

5.4. Варианты заданий

Составить программы и оценить частоту входного и выходного потоков для выполнения одного из следующих действий:

1. Понижение частоты дискретизации входного потока в два раза;
2. Повышение частоты дискретизации входного потока в два раза;
3. Фильтрация входного сигнала фильтром произвольного порядка (не являющимся степенью 2);
4. Квантование отсчетов входного потока с понижением разрядности (результатирующее значение записывается в младших N ($N < 32$) разрядах выходного отсчета);
5. Представление каждого отсчета в виде разности между текущим и предыдущим отсчетами;
6. Представление каждого отсчета в виде знака разности между текущим и предыдущим отсчетами (результатирующее значение записывается в N-м разряде выходного отсчета).

5.5. Контрольные вопросы

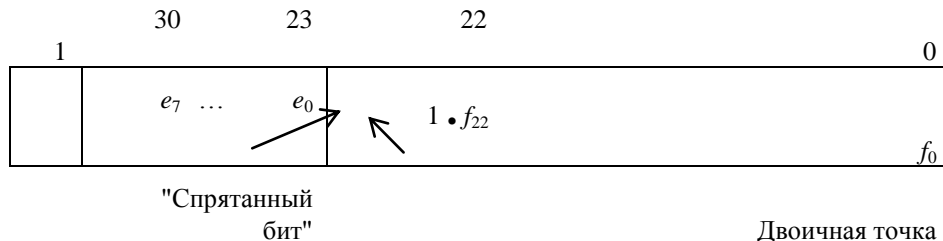
1. Какой минимальный и максимальный интервал времени может быть задан между прерываниями таймера?
2. Какова последовательность действий при обработке прерывания?
3. Для чего используется команда IDLE?
4. Что такое маскирование прерывания? Какие прерывания не могут быть замаскированы?
5. Приведите пример использования вложенных прерываний.
6. Как организовать обработку внешнего или программного прерывания?
7. Обоснуйте приведенное в примере количество тактов, необходимых для обработки одного отсчета.

ЛИТЕРАТУРА

1. ADSP-21000 Family User's Manual. Third Edition 9/95. - Norwood: Analog Devices Inc., 1995. - 496 p.
2. ADSP-21020/21010 User's Manual. Second Edition. - Norwood: Analog Devices Inc., 1994. - 396 p.
3. ADSP-21060 SHARC Preliminary User's Manual. Second Edition 3/94. - Norwood: Analog Devices Inc., 1994. - 186 p.
4. ADSP-21000 Family Assembler Tools & Simulator Manual. Second Edition. - Norwood: Analog Devices Inc., 1994. - 236 p.
5. ADSP-21000 Family Application Handbook Volume 1. First Edition. - Norwood: Analog Devices Inc., 1994. - 352 p.

ФОРМАТЫ ДАННЫХ ADSP-21020

32-битный формат одинарной точности с плавающей запятой в стандарте IEEE 754/854



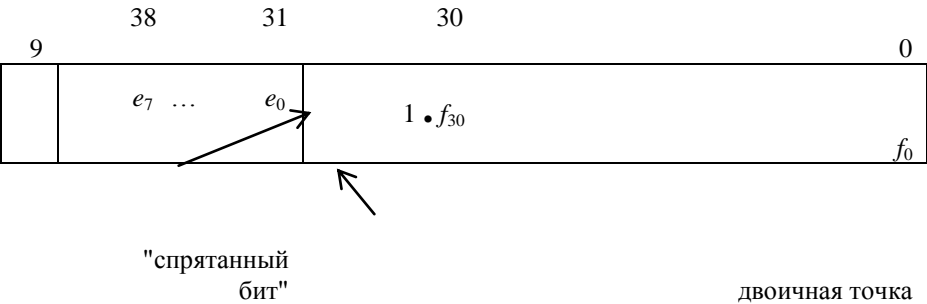
Число одинарной точности в формате с плавающей запятой состоит из знакового бита s , 24-битового поля мантиссы ($f_{22}-f_0$) и 8-битовой беззнаковой смещенной двоичной экспоненты (e_7-e_0). Предполагается, что мантисса состоит из 23-битовой части и "спрятанного" бита, предшествующего биту f_{22} и равного единице. Значение беззнаковой двоичной экспоненты e находится в диапазоне от 1 до 254 и получается путем прибавления к несмещенной двоичной экспоненте числа +127 (при вычислении несмещенной двоичной экспоненты, соответственно, необходимо вычесть это же число из смещенной экспоненты).

Стандарт IEEE в формате чисел с плавающей запятой одинарной точности определяет также некоторые "особые" типы данных, используемые для сигнализации о переполнении, потере значимости и т.п. Типы данных в формате ПЗ-чисел с одинарной точностью приведены в табл. П.1.

Таблица П.1

Тип	Экспонента	Мантисса	Значение
NaN	255	$\neq 0$	не определено
Infinity	255	$= 0$	$(-1)^s * \infty$
Normal	$1 \leq e \leq 254$	любая	$(-1)^s * (1, f_{22-0}) 2^{e-127}$
Zero	0	0	$(-1)^s * 0$

40-битный формат расширенной точности с плавающей запятой

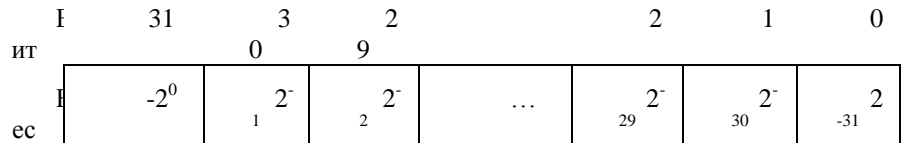


ПЗ-формат повышенной точности отличается от ПЗ-формата одинарной точности только расширенным полем мантиссы (31 бит). Во всем остальном формат повышенной точности соответствует стандартному IEEE формату.

32-битные форматы чисел с фиксированной запятой

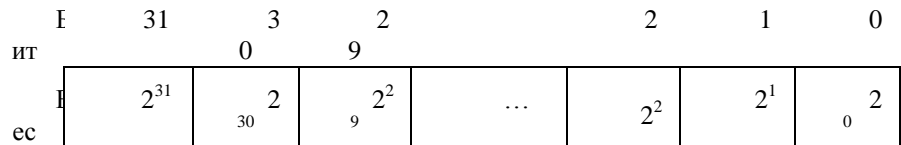
ADSP-21000 поддерживает два ФЗ-формата: формат дробных чисел и формат целых чисел. В обоих форматах числа могут быть знаковыми (в этом случае они представляются в дополнительном коде) или беззнаковыми. В дробном формате предполагается, что двоичная запятая расположена слева от старшего значащего бита. В целом формате считается, что двоичная точка расположена справа от наименьшего значащего бита.



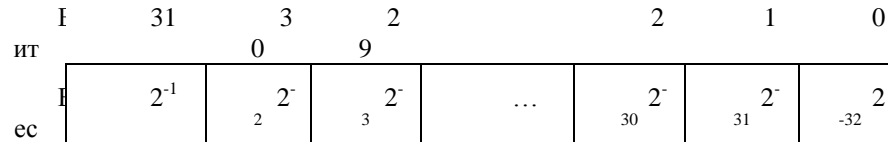


Зна-
ковый
бит

Знаковое дробное



Беззнаковое целое



Зна-
ковый
бит

Беззнаковое дробное

Приложение 2

СИСТЕМНЫЕ РЕГИСТРЫ ADSP-21020

Системные регистры могут быть загружены непосредственным значением или записаны/считаны из памяти данных или любого универсального регистра.

Ре- гистр	Функция	Значение после сброса
MO DE1	Управляющий ре- гистр 1	0x0
MO DE2	Управляющий ре- гистр 2	0xN000 0000 (биты 28-31 со- держат после идентификатора – процессора (версия))
IRP TL	Защелка прерывания	0x0
IM ASK	Маска прерываний	0x3
IM ASKP	Указатель маски пре- рываний	0x0
AS TAT	Арифметический ста- тус	0x00NN 0000 (биты 19-22 соот- ветствуют значениям ножек FLAG ₃₋ 0).
ST KY	Липкий статус	0x0540 0000
US TAT1	Статусы для пользо- вателя	0x0 0x0

US TAT2		
------------	--	--

Запись в любой из системных регистров, за исключением USTAT1 и USTAT2, требует 1 цикла, перед тем как изменения начнут действовать. Циклы ожидания не добавляются. Если вслед за записью в системный регистр следует команда чтения, то будет считано новое записанное значение, кроме регистра IMASKP, для которого требуется дополнительный цикл для обновления значения.

Регистры USTAT1 и USTAT2 – могут использоваться пользователем для временного хранения флагов и данных.

Регистр MODE 1

Б ит	Им я	Описание
0		зарезервирован
1	BR 0	режим бит–реверса для I0 (только при адресации в 0-й банк памяти). Реверсирование (=1) происходит при выводе из DAG1 и не влияет на значение в I0. Последний реальный адрес, выставленный на шину адреса, можно прочитать из регистров DMADR и PMADR
2	SR CU	бит переключения регистров MR переднего и заднего плана (обычно не используется, т.к. можно указывать явно в команде)
3	SR D1H	=1 выбирает альтернативные регистры (7-4) в DAG1
4	SR D1L	=1 выбирает альтернативные регистры (3-0) в DAG1
5	SR D2H	=1 выбирает альтернативные регистры (15-12) в DAG2
6	SR D2L	=1 выбирает альтернативные регистры (11-8) в DAG2

7	SR RFH	=1 выбирает альтернативную группу РФ (15-8)
1 0	SR RFL	=1 выбирает альтернативную группу РФ (7-0)
1 1	NE STM	=1 разрешает вложенные прерывания
1 2	IRP TEN	=1 разрешение прерываний глобальное
1 3	AL USAT	=1 разрешает насыщение в ALU при ФЗ – операциях
1 5	TR UNC	округление при ПЗ – операциях: =1 – усечение; =0 – к ближайшему
1 6	RN D32	=1 – округление ПЗ – данных к 32-битной границе; =0 – 40 бит ПЗ

Регистр MODE 2

Б ит	Им я	Описание
---------	---------	----------

0 -3	IRQ 0E- IRQ3E	=1 – чувствительность по фронту к внешним прерываниям; =0 – чувствительность по уровню к внешним прерываниям
4	CA DIS	=1 – запрещение кэша
5	TI MEN	=1 – включение таймера
1 5-18	FL G00- FLG30	конфигурирование ножек FLAG3-0 как выхода (=1) или как входа (=0). Эти выводы используются для условной синхронизации процессоров. Если флаг сконфигурирован как вход, то его значение появляется в соответствующем бите в ASTAT. Для выходного флага нужно записать в соответствующий бит в ASTAT. <u>Проверка:</u> IF FLAG0_ IN IF NOT FLAG0_ IN
1 9	CA FRZ	=1 – заморозить обновление кэша при конфликтах. Кэш продолжает работать
2 8-31		поле идентификатора версии

Регистр ASTAT

Б ит	Им я	Описание
0	AZ	результат ALU=0 или потеря значимости при ПЗ – операции
1	AV	переполнение ALU
2	AN	результат ALU < 0
3	AC	перенос ALU при ФЗ – операциях
4	AS	знак X – операнда ALU (для операций ABS и MANT)
5	AI	недопустимая (некорректная) ПЗ – операция в ALU
6	MN	результат MAC < 0
7	MV	переполнение MAC
8	MU	потеря значимости при ПЗ – операции в MAC
9	MI	недопустимая (некорректная) ПЗ – операция в MAC
10	AF	признак ПЗ – операции в АЛУ
11	SV	переполнение SHIFTerA

1 2	SZ	результат SHIFTerа < 0
1 3	SS	знак входного операнда SHIFTerа
1 8	BT F	флаг (результат) тестирования битов системных регистров.
1 9-22	FL G0- FLG3	значения FLAG0-FLAG3
2 4-31		CACC – биты (биты аккумулирующих сравнений).

Регистр STKY

Бит	Имя	Описание
0	AUS	потеря значимости при ПЗ – операциях в ALU
1	AVS	переполнение при ПЗ – операциях в ALU
2	AOS	переполнение при ФЗ – операциях в ALU

5	AIS	недопустимая (некорректная) ПЗ – операция ALU
6	MO S	ФЗ - переполнение в MAC
7	MV S	ПЗ – переполнение в MAC
8	MU S	ПЗ – потеря значимости в MAC
9	MI S	некорректная ПЗ – операция в MAC
1 7	CB 7S	переполнение кругового буфера 7 в DAG1
1 8	SB1 5S	переполнение кругового буфера 15 в DAG2
2 1	PC FL	PC – стек полон (не липкий)
2 2	PC EM	PC – стек пуст (не липкий)
2 3	SS OV	стек статуса переполнен (MODE1 и ASTAT)

2 4	SSE M	стек статуса пуст (не липкий)
2 5	LS OV	стек цикла переполнен (адреса циклов и счетчики циклов)
2 6	LS EM	стек цикла пуст (не липкий)

Работа с битами системных регистров

Для манипуляции с битами системных регистров (sreg) ADSP-21000 используется команда BIT с различными модификациями:

IT	S ET	sreg <data32>	Установка битов, номера которых установлены в маске <data32>
	C LR		Сброс битов, номера которых установлены в маске <data32>
	T GL		Инверсия битов, номера которых установлены в маске <data32>
	T ST		Установка бита BTF (в регистре ASTAT), если все биты, указанные в маске <data32>, установлены
	X OR		Установка бита BTF (в регистре ASTAT), если значение системного регистра равно значению <data32>

При подключении файла "def21020.h" для указания битов в системных регистрах могут использоваться их мнемонические обозначения, например

```

BIT SET IMASK TMZHI;      ! разрешить прерывания от таймера
BIT SET MODE2 TIMEN;      ! включить таймер
BIT SET MODE1 IRPTEN;     ! разрешить обработку прерываний

```

Для проверки значений битов системных регистров используется бит BTF и условия IF TF и IF NOT TF.

Приложение 3

ИНСТРУКЦИИ ВЫЧИСЛИТЕЛЬНЫХ ОПЕРАЦИЙ

Инструкции АЛУ

Инструкции	Регистр ASTAT								Регистр STKY				Краткое описание
	Z	V	N	C	S	I	F	ACC	US	VS	OS	IS	
$R_n = R_x \pm R_y$								-					Сложение (вычитание) регистров R_x и R_y . Результат помещается в ФЗ-поле регистра R_n . ПЗ-расширение регистра R_n сбрасывается в 0.
$R_n = R_x + R_y +$ CI								-					Сложение регистров R_x и R_y "с переносом"
$R_n = R_x - R_y +$ CI -1								-					Вычитание регистров R_x и R_y "с заемом"
$R_n = (R_x + R_y)/2$								-					Среднее арифметическое регистров R_x и R_y
COMP(R_x, R_y)								*					Сравнение регистров R_x и R_y путем вычитания $R_x - R_y$. Содержимое регистров не изменяется

Инструкции	Регистр ASTAT								Регистр STKY				Краткое описание
	Z	V	N	C	S	I	F	ACC	US	VS	OS	IS	
$R_n = R_x + CI$								-					Сложение переноса и содержимого регистра R_x
$R_n = R_x + CI - 1$								-					Сложение заема и содержимого регистра R_x
$R_n = R_x + 1$								-					
$R_n = R_x - 1$								-					
$R_n = - R_x$								-					
$R_n = ABS R_x$								-					Абсолютное значение регистра R_x
$R_n = PASS R_x$								-					Перенос значения регистра R_x в регистр R_n (записывается только ФЗ-поле, ПЗ-расширение регистра R_n сбрасывается в 0)
$R_n = R_x AND R_y$								-					Порярядное "И"
$R_n = R_x OR R_y$								-					Порярядное "ИЛИ"
$R_n = R_x XOR R_y$								-					Порярядное "Исключающее ИЛИ"

Инструкции	Регистр ASTAT								Регистр STKY				Краткое описание
	Z	V	N	C	S	I	F	ACC	US	VS	OS	IS	
$R_n = \text{NOT } R_x$								-					Порядная инверсия
$R_n = \text{MIN}(R_x, R_y)$								-					Возвращает меньшее из значений регистров R_x и R_y
$R_n = \text{MAX}(R_x, R_y)$								-					Возвращает большее из значений регистров R_x и R_y
$R_n = \text{CLIP } R_x$ $B_y R_y$								-					Возвращает R_x если $ R_x < R_y $. Иначе возвращает R_y
$F_n = F_x \pm F_y$								-					Сложение (вычитание) регистров F_x и F_y . Используются 40-битные операнды. Результат помещается в регистр F_n
$F_n = \text{ABS}(F_x \pm F_y)$								-					Модуль суммы (разности) регистров F_x и F_y
$F_n = (F_x + F_y)/2$								-					
$\text{COMP}(F_x, F_y)$								*					Сравнение регистров F_x и F_y путем вычитания $F_x - F_y$. Содержимое регистров не изменяется

Инструкции	Регистр ASTAT								Регистр STKY				Краткое описание
	Z	V	N	C	S	I	F	ACC	US	VS	OS	IS	
$F_n = - F_x$								-					
$F_n = ABS F_x$								-					
$F_n = PASS F_x$								-					
$F_n = RND F_x$								-					Округление регистра F_x до 32-битного значения
$F_n = SCLAB F_x$ BY R_y								-					Масштабирование экспоненты регистра F_x путем прибавления к ней знакового ФЗ-значения из регистра R_x
$R_n = MANT F_x$								-					Выделение мантиссы (со "скрытым" битом) и запись ее в беззнаковом дробном формате 1.31
$R_n = LOGB F_x$								-					Нахождение несмещенной экспоненты числа в регистре F_x
$R_n = FIX F_x$								-					Преобразование ПЗ-числа в регистре F_x в знаковое 32-битное число формата 32.0

Инструкции	Регистр ASTAT								Регистр STKY				Краткое описание
	Z	V	N	C	S	I	F	ACC	US	VS	OS	IS	
$R_n = \text{FIX } F_x \text{ } B_Y$ R_Y								-					Аналогично предыдущей инструкции, но перед преобразованием целочисленное значение со знаком регистра R_y прибавляется к экспоненте числа в регистре F_x
$F_n = \text{FLOAT } R_x$								-					Операция, обратная FIX
$F_n = \text{FLOAT } R_x$ $B_Y R_Y$								-					Операция, обратная FIX...BY
$F_n = \text{RECIPS } F_x$								-					Возвращает значение, обратное F_x ($1/F_x$)
$F_n = \text{RSQRTS}$ F_x								-					Возвращает значение, обратное квадратному корню из F_x
$F_n = F_x$ COPYSIGN F_y								-					Возвращает значение регистра F_x со знаковым битом, равным знаковому биту регистра F_y
$F_n = \text{MIN}(F_x, F_y)$								-					

Инструкции	Регистр ASTAT								Регистр STKY				Краткое описание
	Z	V	N	C	S	I	F	ACC	US	VS	OS	IS	
$F_n = \text{MAX}(F_x, F_y)$								-					
$F_n = \text{CLIP } F_x$ BY F_y								-					

R_n, R_x, R_y – любая ячейка РФ (обрабатывается как ФЗ); F_n, F_x, F_y – любая ячейка РФ (обрабатывается как ПЗ); "*" – устанавливается или обнуляется в зависимости от результатов инструкции; "#" – может установиться (но не сброситься) в зависимости от результатов инструкции; "-" – не действует; "0" – сбрасывается в 0; "1" – устанавливается в 1.

Инструкции умножителя

Инструкции	Регистр ASTAT				Регистр STKY				Краткое описание
	U	N	V	I	US	OS	VS	IS	
$\begin{matrix} R_n \\ MRF \\ MRB \end{matrix} = R_x * R_y \left(\begin{matrix} S \\ U \end{matrix} \middle \begin{matrix} S \\ U \end{matrix} \begin{matrix} F \\ I \\ FR \end{matrix} \right)$									Произведение регистров R _x и R _y . Результат помещается в аккумулятор или в ФЗ-поле регистра (для целочисленных операндов в регистр заносятся биты 31-0 результата, а для дробных – биты 63-32 результата)
$\begin{matrix} R_n = MRF \\ R_n = MRB \\ MRF = MRF \\ MRB = MRB \end{matrix} \pm R_x * R_y \left(\begin{matrix} S \\ U \end{matrix} \middle \begin{matrix} S \\ U \end{matrix} \begin{matrix} F \\ I \\ FR \end{matrix} \right)$									Сложение (вычитание) результата умножения регистров R _x и R _y с содержимым аккумулятора MAC
$\begin{matrix} R_n = SATMRF \\ R_n = SATMRB \\ MRF = SATMRF \\ MRB = SATMRB \end{matrix} \left(\begin{matrix} SI \\ UI \\ SF \\ UF \end{matrix} \right)$									Насыщение значения, содержащегося в аккумуляторе MAC, в соответствии с указанным форматом данных

Инструкции	Регистр ASTAT				Регистр STKY				Краткое описание
	U	N	V	I	US	OS	VS	IS	
$\begin{cases} R_n = RNDMRF \\ R_n = RNDMRB \\ MRF = RNDMRF \\ MRB = RNDMRB \end{cases} \begin{cases} (SF) \\ (UF) \end{cases}$									Округление значения, содержащегося в аккумуляторе MAC, по 32-х битной границе (только для дробных чисел)
$\begin{cases} MRF \\ MRB \end{cases} = 0$									
$\begin{cases} MR_x F \\ MR_x B \end{cases} = R_n$									
$R_n = \begin{cases} MR_x F \\ MR_x B \end{cases}$									
$F_n = F_x * F_y$									

R_n, R_x, R_y – любая ячейка РФ (обрабатывается как ФЗ); **F_n, F_x, F_y** – любая ячейка РФ (обрабатывается как ПЗ); **$MR_x F = MR2F, MR1F, MR0F$** – аккумуляторы результата умножения переднего плана; **$MR_x B = MR2B, MR1B, MR0B$** – аккумуляторы результата умножения заднего плана; "*" – устанавливается или обнуляется в зависимости от результатов

инструкции; "#" - может установиться (но не сброситься) в зависимости от результатов инструкции, "-" - не действует; "0" – всегда сбрасывается в 0.

Инструкции сдвигателя

Инструкция	Ре- гистр ASTAT			Краткое описание
	Z	V	S	
$R_n = \text{LSHIFT } R_x \text{ BY } R_y$				Логический сдвиг ФЗ-поля регистра R_x на значение, содержащееся в регистре R_y (если $R_y > 0$, то сдвиг влево, иначе – вправо)
$R_n = \text{LSHIFT } R_x \text{ BY } \langle \text{data8} \rangle$				Аналогично предыдущей инструкции, но значение сдвига указывается непосредственно в инструкции
$R_n = R_n \text{ OR LSHIFT } R_x \text{ BY } R_y$				Аналогично первой инструкции, но результат комбинируется с содержимым регистра R_n с использованием логического "ИЛИ"
$R_n = R_n \text{ OR LSHIFT } R_x \text{ BY } \langle \text{data8} \rangle$				
$R_n = \text{ASHIFT } R_x \text{ BY } R_y$				Арифметический сдвиг
$R_n = \text{ASHIFT } R_x \text{ BY } \langle \text{data8} \rangle$				
$R_n = R_n \text{ OR ASHIFT } R_x \text{ BY } R_y$				
$R_n = R_n \text{ OR ASHIFT } R_x \text{ BY } \langle \text{data8} \rangle$				

Инструкция	Ре- гистр ASTAT			Краткое описание
	Z	V	S	
$R_n = \text{ROT } R_x \text{ BY } R_y$				Циклический сдвиг
$R_n = \text{ROT } R_x \text{ BY } \langle \text{data8} \rangle$				
$R_n = \text{BCLR } R_x \text{ BY } R_y$				Очистка бита номер R_y в регистре R_x
$R_n = \text{BCLR } R_x \text{ BY } \langle \text{data8} \rangle$				
$R_n = \text{BSET } R_x \text{ BY } R_y$				Установка бита номер R_y в регистре R_x
$R_n = \text{BSET } R_x \text{ BY } \langle \text{data8} \rangle$				
$R_n = \text{BTGL } R_x \text{ BY } R_y$				Инвертирование бита номер R_y в регистре R_x
$R_n = \text{BTGL } R_x \text{ BY } \langle \text{data8} \rangle$				
$\text{BTST } R_x \text{ BY } R_y$				Тестирование бита номер R_y в регистре R_x . Флаг SZ сбрасывается, если бит установлен и устанавливается в противном случае
$\text{BTST } R_x \text{ BY } \langle \text{data8} \rangle$				

Инструкция	Ре- гистр ASTAT			Краткое описание
	Z	V	S	
$R_n = \text{FDEP } R_x \text{ BY } R_y \text{ (SE) }$				Депонирование битового поля. Если указан модификатор (SE), то выполняется знаковое расширение результата до 32-битного значения
$R_n = \text{FDEP } R_x \text{ BY } \langle \text{bit6} \rangle : \langle \text{len6} \rangle \text{ (SE) }$				Аналогично предыдущей инструкции, но поля $\langle \text{bit6} \rangle$ и $\langle \text{len6} \rangle$ берутся непосредственно из инструкции
$R_n = R_n \text{ OR FDEP } R_x \text{ BY } R_y \text{ (SE) }$				
$R_n = R_n \text{ OR FDEP } R_x \text{ BY } \langle \text{bit6} \rangle : \langle \text{len6} \rangle \text{ (SE) }$				
$R_n = \text{FEXT } R_x \text{ BY } R_y \text{ (SE) }$				Выделение битового поля. Если указан модификатор (SE), то выполняется знаковое расширение результата до 32-битного значения
$R_n = \text{FEXT } R_x \text{ BY } \langle \text{bit6} \rangle : \langle \text{len6} \rangle \text{ (SE) }$				

Инструкция	Ре- гистр ASTAT			Краткое описание
	Z	V	S	
$R_n = \text{EXP } R_x \text{ } (EX) $				Возвращает двоичную экспоненту ФЗ-числа (количество избыточных знаковых битов числа в регистре R_x). Результат помещается в младшие 8 бит регистра R_n . Если указан модификатор (SE), то при определении экспоненты учитывается флаг переноса АЛУ AV
$R_n = \text{LEFTZ } R_x$				Возвращает количество ведущих нулей для числа в регистре R_x
$R_n = \text{LEFT0 } R_x$				Возвращает количество ведущих единиц для числа в регистре R_x

R_n , R_x , R_y – любая ячейка РФ (обрабатывается как ФЗ); "*" - устанавливается или обнуляется в зависимости от результатов инструкции; "0" - всегда сбрасывается в 0; <data8> - 8-битная константа; <bit6> - 6-битная константа, определяющая местоположение выделяемого или депонируемого поля (младший бит); <len6> - 6-битная константа, определяющая длину депонируемого поля.

СОДЕРЖАНИЕ

1. Лабораторная работа 1. Базовая архитектура и средства разработки программного обеспечения для ADSP-21000.....	
2. Лабораторная работа 2. Организация доступа к внешней памяти ADSP-21000. Файл описания архитектуры.....	
3. Лабораторная работа 3. Вычислительные блоки процессоров семейства ADSP-21000.....	
4. Лабораторная работа 4. Управление программой в процессорах семейства ADSP-21000.....	
5. Лабораторная работа 5. Обработка прерывания в процессорах семейства ADSP-21000.....	
Литература.....	
Приложение 1. Форматы данных ADSP-21020.....	
Приложение 2. Системные регистры ADSP-21020.....	
Приложение 3. Инструкции вычислительных операций.....	