Учебное издание

А.А. Савин, Е.П. Ворошилин

Программирование цифровых сигнальных процессоров семейства SHARC с использованием среды разработки «Analog Devices Visual DSP 3.5»

Руководство к курсу лабораторных работ по дисциплине «Проектирование цифровых сигнальных процессоров» для студентов радиотехнического факультета

Кафедра радиотехнических систем (РТС)

А.А. Савин, Е.П. Ворошилин

Программирование цифровых сигнальных процессоров семейства SHARC с Использованием среды разработки «Analog Devices Visual DSP 3.5»

Руководство к курсу лабораторных работ по дисциплине «Проектирование цифровых сигнальных процессоров» для студентов радиотехнического факультета

.

2012

Министерство образования и науки Российской Федерации
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

Кафедра Радиотехнических систем (РТС)

А.А. Савин, Е.П. Ворошилин

Программирование цифровых сигнальных процессоров семейства SHARC с Использованием среды разработки «Analog Devices Visual DSP 3.5»

Руководство к курсу лабораторных работ по дисциплине «Проектирование цифровых сигнальных процессоров» для студентов радиотехнического факультета

Томск 2012

Содержание

Введение.

Учебно-методическое пособие предназначено для проведения курса лабораторных работ по дисциплине «Микропроцессоры». Выполнение лабораторных работ предусматривает разработку программных модулей для процессоров ADSP-21062 и ADSP-21065L с использованием пакета VisualDSP 3.5 компании Analog Devices (AD).

В рамках учебно-методического пособия, помимо курса из семи лабораторных работ, приведено:

- краткое описание пакета VisualDSP 3.5, которое включает в себя рекомендации по установке и правильной настройке пакета, описание необходимых для качественного выполнения курса лабораторных работ возможностей пакета VisualDSP 3.5.

- краткое описание языка ассемблер, используемого наряду с C и C++ для программирования процессоров семейства SHARC.

- краткое описание архитектуры процессоров семейства SHARC, сравнение их быстродействия с другими процессорами фирмы AD.

- примеры программ на ассемблере, разработанные специально для данного методического пособия и снабженные подробными комментариями.

Следует отметить, что русскоязычной литературы, посвященной вопросам программирования процессоров фирмы Analog Devices, в настоящее время немного, а описание пакета VisualDSP 3.5 на русском языке отсутствует вовсе. Только на сайте компании Analog Devices (www.analog.com) приведены описания пакета, процессоров SHARC и поддерживаемых языков программирования на английском языке. Следует отметить, что руководства имеют объем от 400 до 1500 листов формата A4. Поэтому самостоятельное изучение данного вопроса потребует больших временных затрат.

Основная задача пособия − помочь студенту разобраться в азах программирования микропроцессоров и сразу же применить полученные

знания на практике. В пособии содержится вся необходимая информация для успешного выполнения курса лабораторных работ и последующей их защиты.

Принцип изложения материала в пособии − от простого к сложному. Каждая глава дополняет предыдущую. Итогом прочтения методического пособия и выполнения курса лабораторных работ должно быть создание представления о принципах работы цифровых сигнальных процессоров (ЦСП), выработка навыков программирования на языке ассемблер.

Пособие может быть использовано при дистанционной технологии обучения.

1. Описание пакета VisualDSP++ 3.5 Analog Devices

1.1. Задачи раздела

Основной задачей данного раздела является описание интерфейса и возможностей пакета VisualDSP++ 3.5 на уровне, достаточном для успешного выполнения курса лабораторных работ по дисциплине «Микропроцессоры». Большая часть возможностей, которым необходимо обучиться, снабжены иллюстрациями и краткими пояснениями, что позволит существенно ускорить процесс освоения. В разделе приведены рекомендации по установке, настройке VisualDSP++ 3.5, описаны наиболее важные встроенные возможности пакета, с помощью которых можно:

- получить представление о среде разработки программного обеспечения для процессоров семейства SHARC;
- облегчить отладку и написание исходного кода программ для цифровых сигнальных процессоров (ЦСП);
- выгодно представить результаты работы и проверить правильность ее выполнения;
- быстро и качественно составить отчет.

1.2. Общие сведения о пакете

Пакет VisualDSP++ 3.5 предназначен для разработки и настройки программного обеспечения для цифровых сигнальных процессоров фирмы Analog Devices. В пакете VisualDSP++ 3.5 успешно сочетаются эргономичный пользовательский интерфейс, удобный редактор кода, множество инструментов для отладки программ. Возможности пакета позволяют реализовать полный цикл разработки программного обеспечения, от написания исходного кода и компиляции до создания финальных (загрузочных) модулей. Пакет снабжен полным описанием на английском языке, большим количеством специализированных библиотек и примеров

готовых программ. Редактор кода позволяет разрабатывать программное обеспечение для процессоров на языках ассемблер, C/C ++.

Пакет VisualDSP++ 3.5 обеспечивает возможность разработки программного обеспечения для процессоров семейства SHARC и более современного Tiger SHARC.

VisualDSP++ 3.5 поддерживает следующие процессоры семейства SHARC:

ADSP-21020, ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, ADSP-21160, ADSP-21161, ADSP-21261, ADSP-21262, ADSP-21266, ADSP-21267, ADSP-21363, ADSP-21364, ADSP-21365.

VisualDSP++3.5 поддерживает следующие процессоры семейства Tiger SHARC:

ADSP-TS101 DSP, ADSP-TS201 DSP, ADSP-TS202 DSP,

ADSP-TS203 DSP.

Для установки и запуска VisualDSP++ 3.5 ваш компьютер должен удовлетворять следующим аппаратным и программным требованиям:

• Windows 98 SR2/NT 4.0 SP3/2000/ME/XP;

• не менее 100 Mb свободного дискового пространства;

• не менее 32 Mb оперативной памяти;

• привод CD-ROM;

• Internet Explorer 4.01 или более новый.

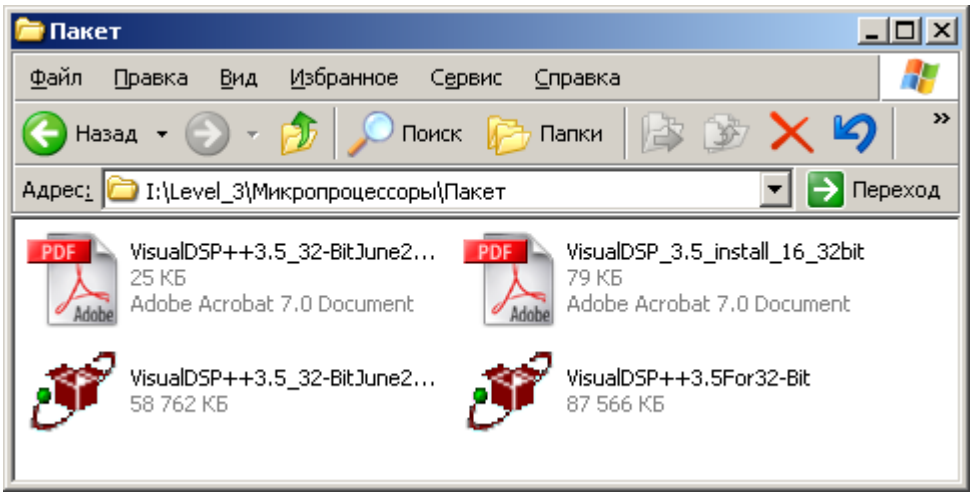
1.3. Установка VisualDSP++3.5 на ЭВМ

Для успешной установки пакета на ЭВМ необходимо:

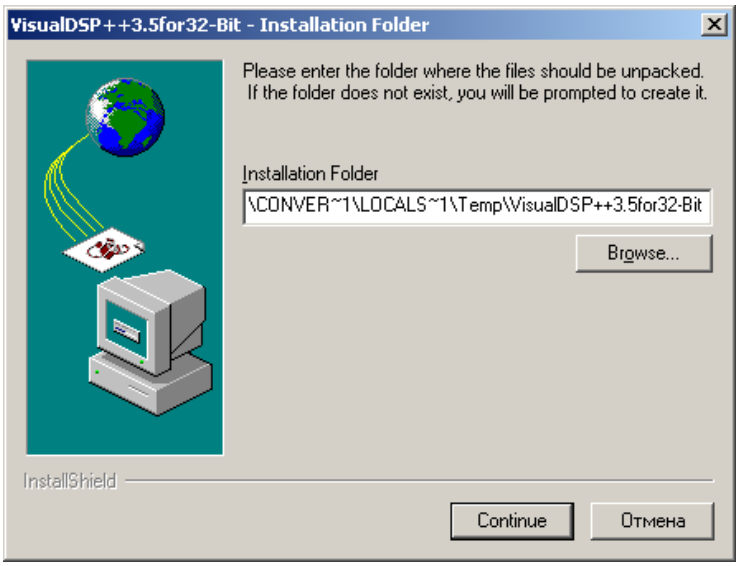
- приобрести пакет VisualDSP++ 3.5 либо скачать с сайта производителя демонстрационную версию (активна в течение 30 дней после установки);
- найти на сайте Analog Devices последние обновления для VisualDSP++ 3.5 и скопировать их на свой компьютер;

- желательно найти и скопировать описания дополнений к программе.

Если все сделано правильно, то в каталоге с установочной версией программы должны оказаться файлы:

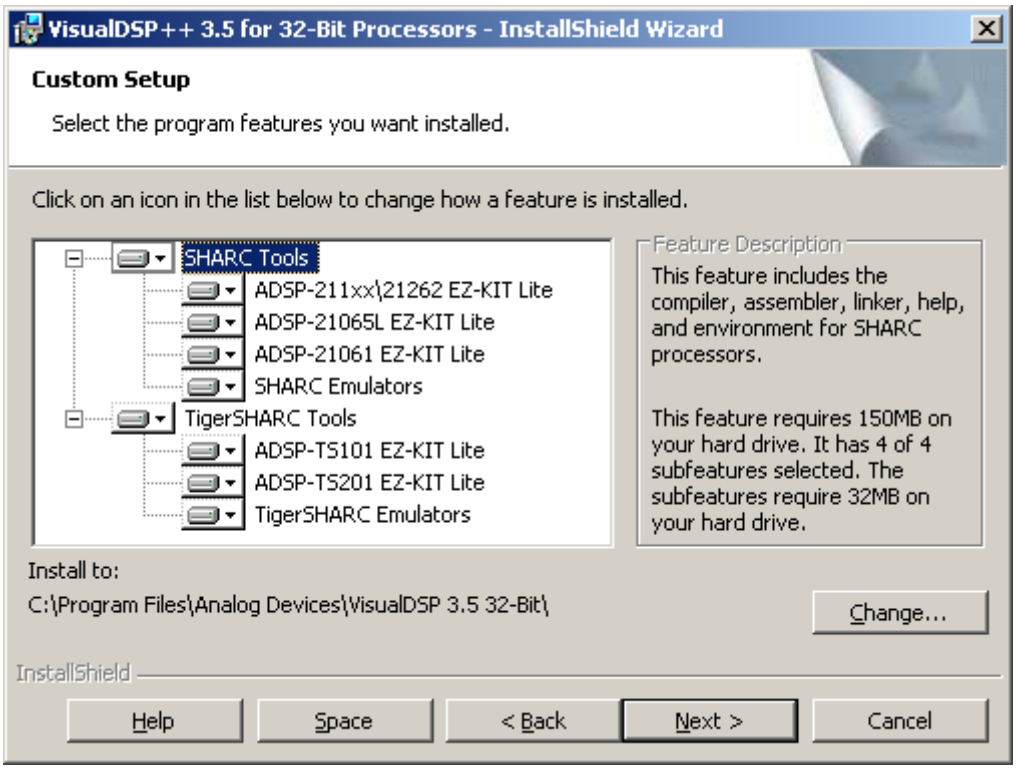


После этого можно приступать к установке пакета. Для этого необходимо запустить файл VisualDSP++ 3.5 For32-bit. В результате на экране ЭВМ появится окно:



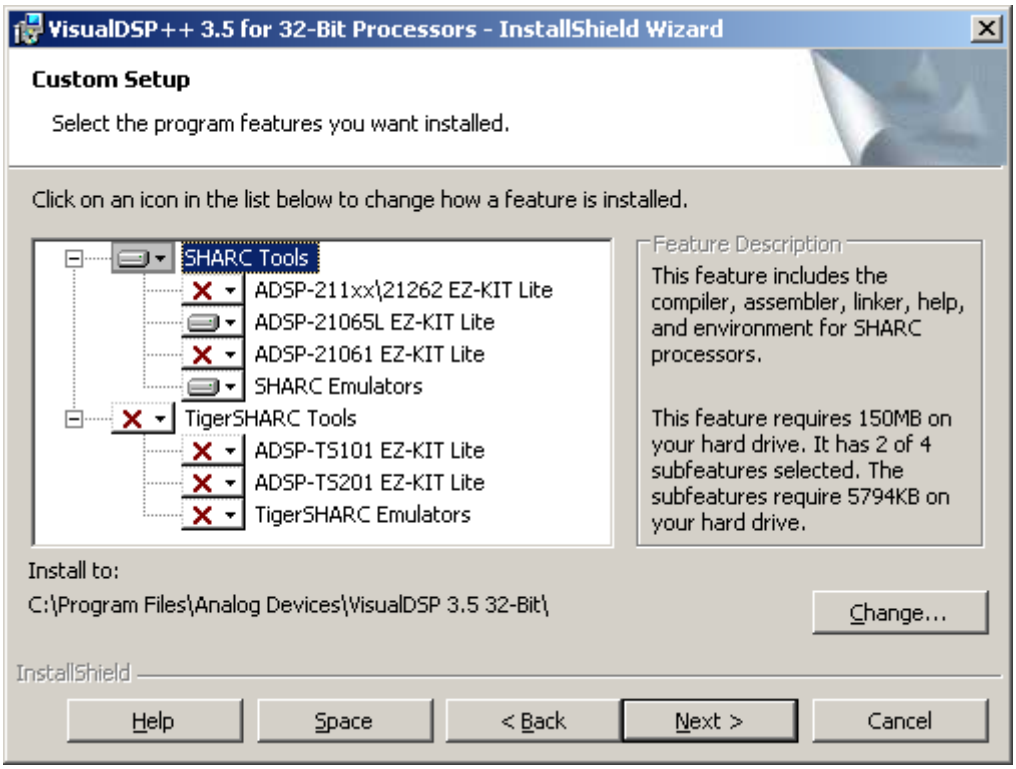
В окне редактирования Installation Folder необходимо указать путь, куда будут распакованы временные файлы, и нажать кнопку Continue. После завершения распаковки временных файлов автоматически запустится мастер установки приложения:

Для продолжения установки необходимо нажать кнопку Next, в результате чего появится окно настройки конфигурации, с которой VisualDSP++ 3.5 будет установлен.



Для выполнения всего курса лабораторных работ достаточно установить VisualDSP++3.5 с поддержкой SHARC процессоров, все остальное можно исключить. Это существенно уменьшит размеры пакета и позволит избежать путаницы при работе с примерами программ. После

последовательного отключения невостребованных возможностей, окно редактирования принимает вид:

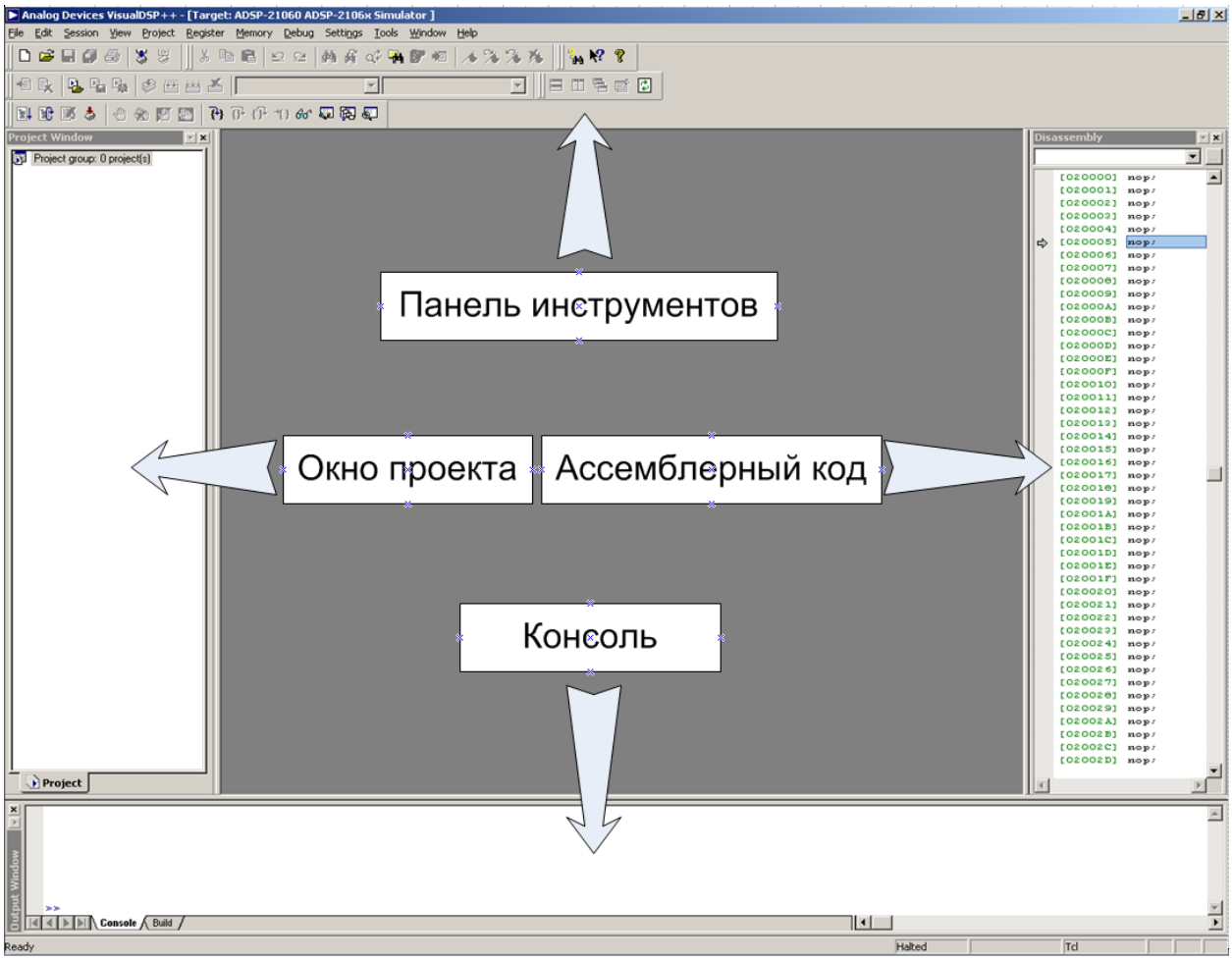


Продолжите установку нажатием кнопки Next, после чего начнется копирование файлов программы на ваш компьютер. По окончании установки нажмите кнопку **Finish**.

После этого установите обновления для VisualDSP++ 3.5 в тот же каталог, что и сам пакет. На этом установку можно считать законченной, а программу готовой к работе.

1.4. Описание интерфейса пакета

Запустить программу можно из меню **Пуск->Все программы->Analog Devices->VisualDSP++ 3.5 For 32-bit Processor-> VisualDSP++ 3.5 Enviroment**. Окно программы при первом запуске будет иметь вид, приведенный ниже.



В верхней части экрана расположено меню VisualDSP++3.5. Далее будут описаны необходимые для успешного выполнения курса лабораторных работ элементы меню.

1. Меню File

- **New** − создает новый файл,

- **Open** − открывает файл,

- **Save as** − сохраняет файл под новым именем.

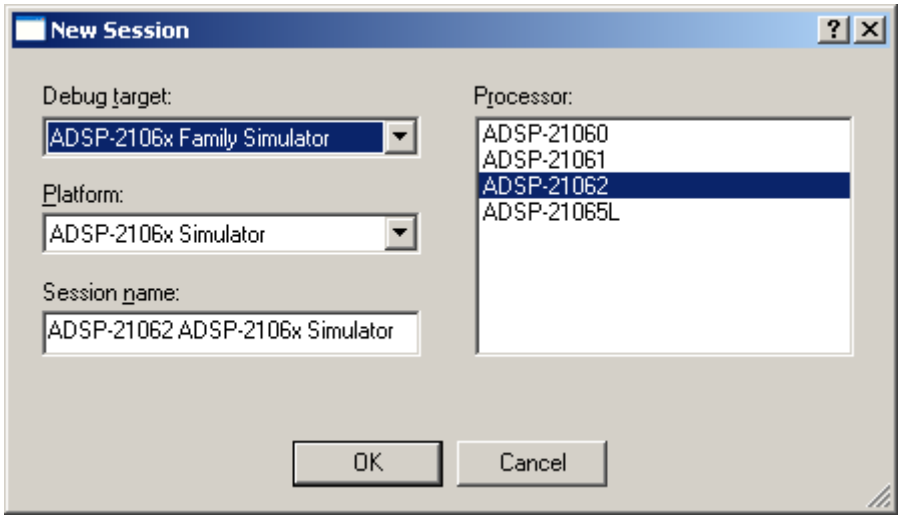
2. Меню **Edit**

Содержит стандартные элементы, не требующие комментариев.

3. Меню **Session**

- **New Session** − позволяет создать сессию для одного из процессоров. Проще говоря, выбор одного из процессоров в данном пункте меню сигнализирует пакету VisualDSP++ 3.5, что все действия пользователя будут направлены на работу с выбранным процессором, а VisualDSP++ 3.5 должен имитировать работу выбранного процессора в режиме симуляции. Вдобавок

VisualDSP++ 3.5 автоматически обновит все пункты меню, содержание которых зависит от типа процессора. Для выполнения курса лабораторных работ необходимо в поле **Debug target** окна **New Session** выбрать строчку **ADSP-2106x Family Simulator**, а в поле **Processor** выбрать процессор **ADSP-21062**. После чего окно **New Session** будет иметь следующий вид:

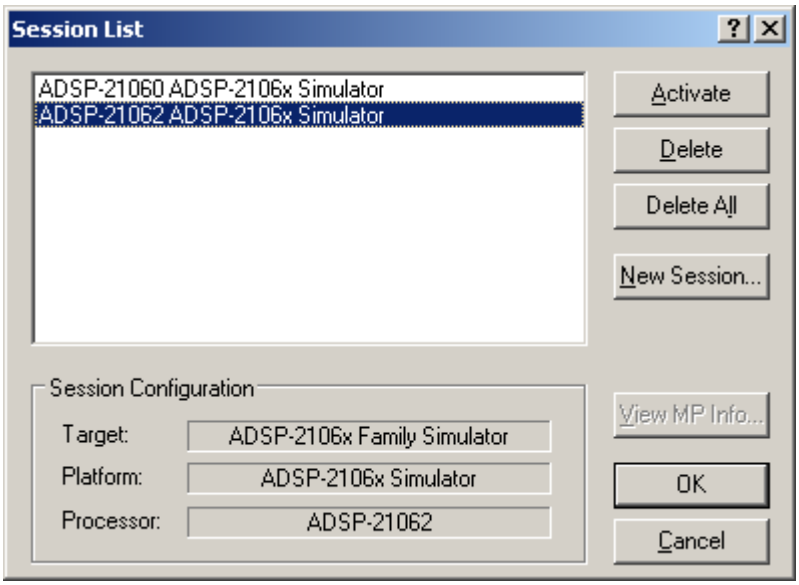


После выбора сессии необходимо нажать кнопку OK, и новая сессия для процессора **ADSP-21062** будет создана.

- **Select Session** − позволяет выбрать одну из уже существующих сессий и сделать ее активной.

Данный пункт меню полезен, когда пользователь параллельно разрабатывает программное обеспечение для двух или большего количества процессоров и ему необходимо быстро переключать настройки пакета.

- **Session List** − выводит на экран список всех ранее созданных и доступных пользователю сессий. Позволяет удалять (**Delete**, **Delete All**), добавлять (**New Session**) и активировать (**Activate**) сессии. Внешний вид окна представлен на рисунке.



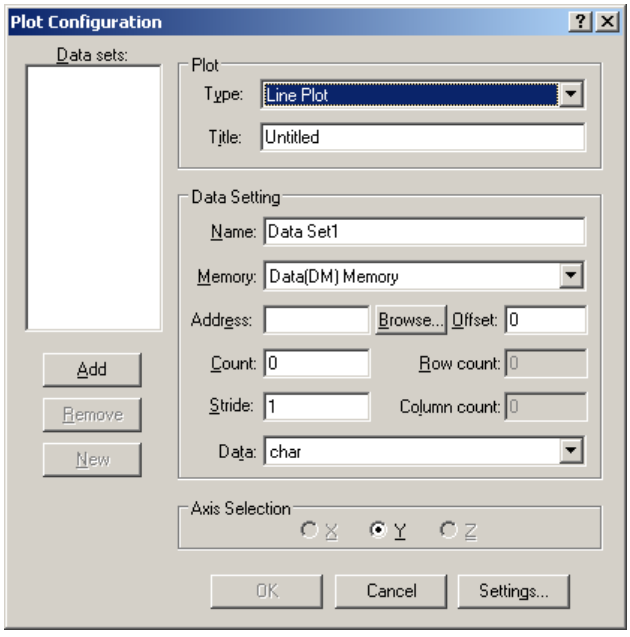
4. Меню **View**

- **Project Window** − включает и отключает режим отображения окна **Project**, которое содержит все файлы, принадлежащие данному проекту.

- **Output Window** − включает и отключает режим отображения окна **Output**. Окно **Output** содержит информацию, генерируемую **VisualDSP++ 3.5** автоматически при сборке проекта (всевозможные предупреждения об ошибках и неисправностях, обнаруженных компилятором).

- **Debug Window->Plot->New** − позволяет создать окно для построения одномерных графиков или изображений.

При выборе данного пункта меню на экране появляется окно:



С помощью активных полей окна **Plot Configuration** пользователь определяет параметры графика. Графики применяются исключительно для отображения содержимого внутренней или внешней памяти процессора.

В поле **Type** задается режим отображения данных − от обычных линейных графиков до спектрограмм и двухмерных изображений.

Поле **Title** содержит текст заголовка графика.

Поле **Name** содержит названия блока данных, который необходимо отобразить на экране (заполняется пользователем самостоятельно).

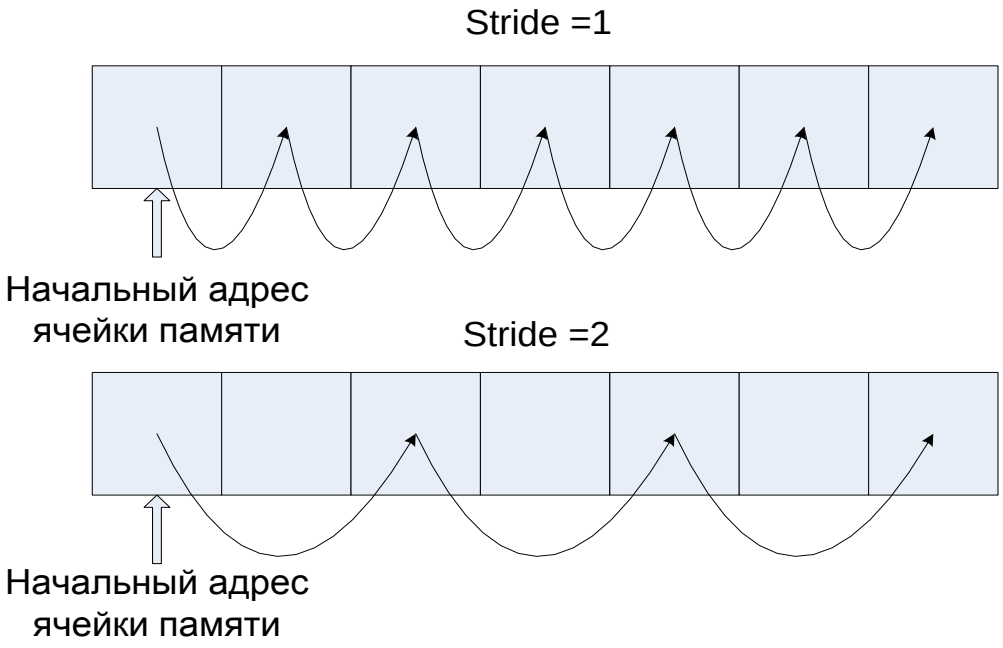
Поле **Memory** обеспечивает возможность выбора области памяти, из которой необходимо считать данные (внутренняя память процессора разбита на несколько отдельных областей, подробно этот вопрос освещен в пункте 9).

В поле **Address** задается начальный адрес массива данных, который будет отображаться на экране.

Кнопка **Browse** позволяет напрямую выбрать начальный адрес массива данных из списка начальных адресов переменных, расположенных во внутренней памяти процессора.

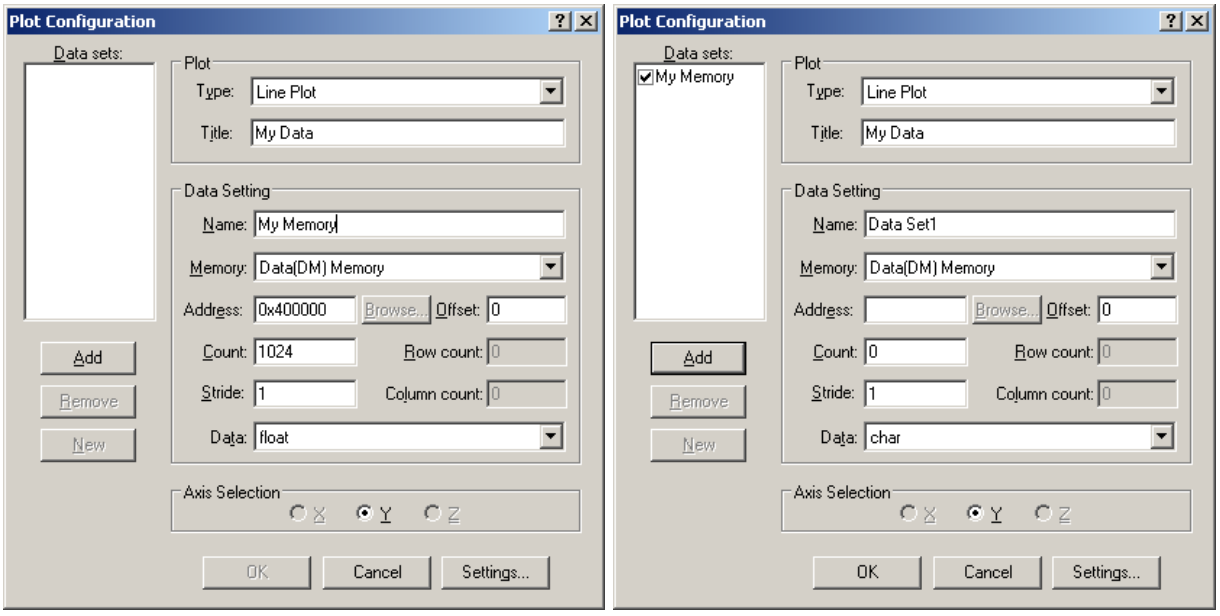
Поле **Count** определяет, сколько слов данных, начиная с начального адреса, необходимо отобразить на графике.

Поле **Stride** определяет, с каким шагом, начиная с начального адреса, будут считываться данные из памяти.



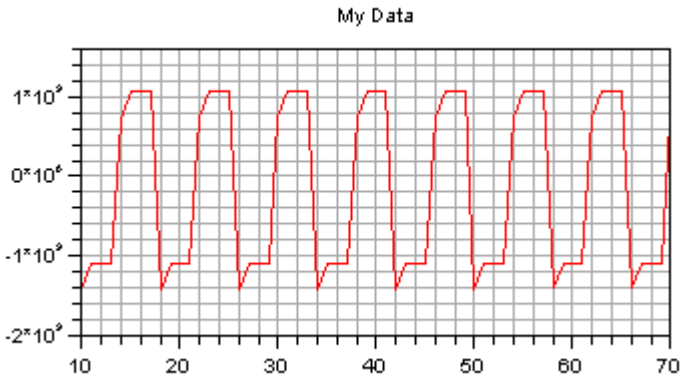
Поле **Data** определяет тип отображаемых данных (это может быть один из стандартных типов данных char, int, float, double, long и т.д).

После того как все поля заполнены, необходимо нажать кнопку **add**, чтобы добавить массив в список отображаемых данных **Data Sets**.



Чтобы отобразить содержимое области памяти на экране монитора, необходимо нажать кнопку ОК.

Для примера приведено окно, в котором отображается периодический сигнал, предварительно записанный в память процессора.



Использование данного пункта меню чрезвычайно полезно при отладке программы, просмотре данных и создании отчетов по лабораторным работам.

5. Меню **Project**

- **New** – создает новый проект,

- **Open** – открывает существующий проект,

- **Save As** – сохраняет проект под новым именем,

- **Close** – закрывает проект,

- **Set Active Project** – делает один из проектов активным,

- **Add To Project** – добавляет файл к проекту,

- **Remove Selection** – удаляет выделенные файлы или папки из проекта,

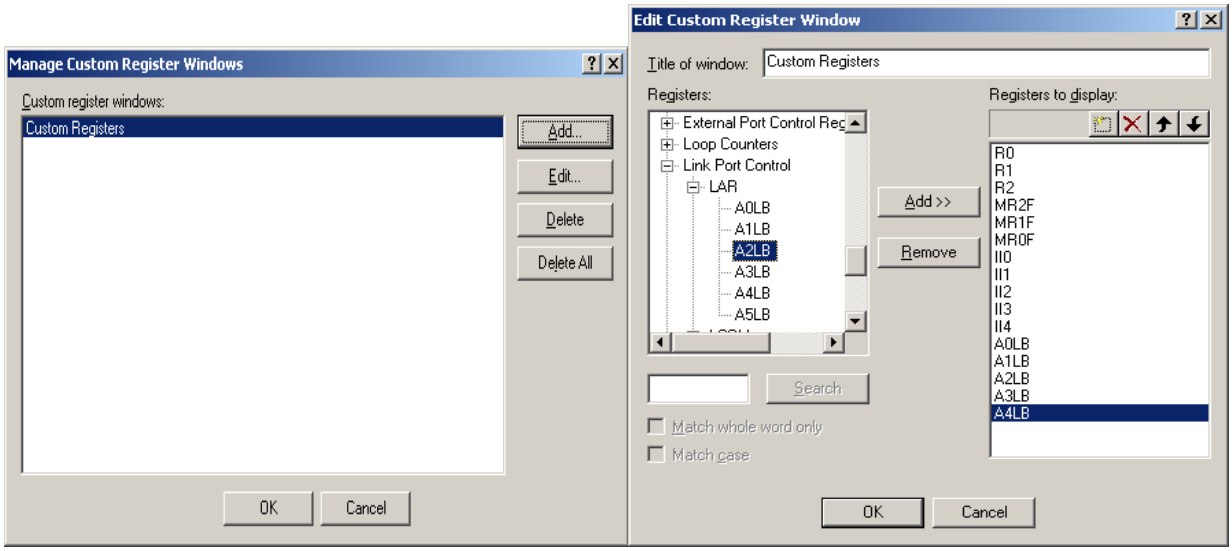
- **Build File** – компилирует отдельный файл проекта,

- **Build Project** – компилирует весь проект,

- **Rebuilt all** – перекомпилирует все файлы и все проекты (поскольку одновременно может быть открыто более одного проекта).

6. Меню **Register**

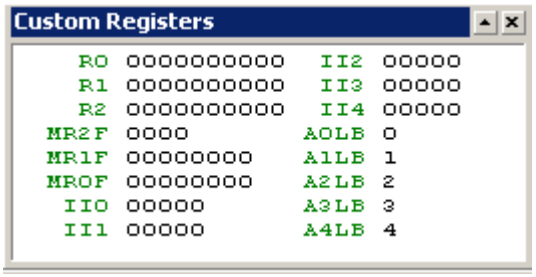
- **Register->Custom->manage** – позволяет задать набор регистров, содержимое которых будет отображаться на экране монитора на отдельной пользовательской панели. При выборе данного пункта меню на экране появляется окно **Manage Custom Register Windows**. Чтобы произвести выбор регистров, необходимо нажать на кнопку **Add,** и в появившемся окне настроить содержимое новой панели, после чего нажать кнопку **OK**.



Под регистром будем понимать ячейку памяти заданной разрядности (8, 16, 24, 32, 48), содержащую некоторое числовое значение. Содержимое

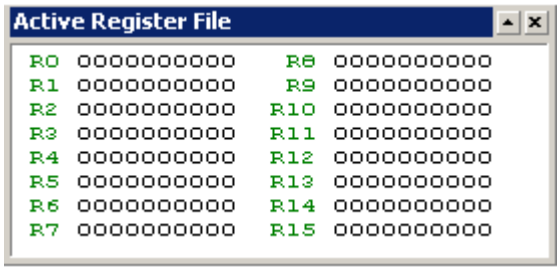
регистра может быть использовано процессором в арифметической или логической операции как команда для перехода в нужный режим работы или адрес ячейки памяти.

- **Register->Custom->Custom Registers** – выводит на экран пользовательскую панель, отображающую содержимое регистров, выбранных в окне **Custom->manage.**



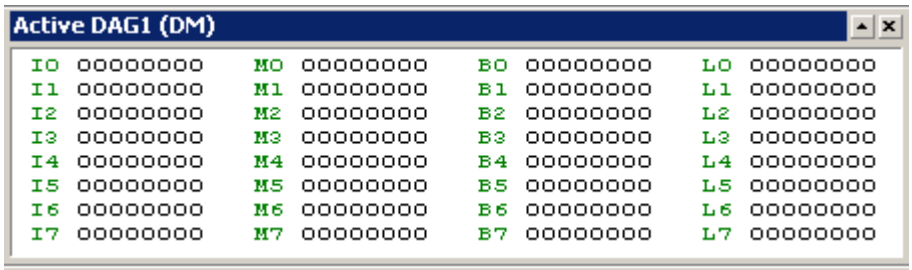
Панель **Custom Registers** является одним из наиболее удобных средств отладки ассемблерных программ, поскольку позволяет объединять наборы совместно используемых регистров.

- **Core->Register File** – выводит на экран окно с регистрами R0-15.



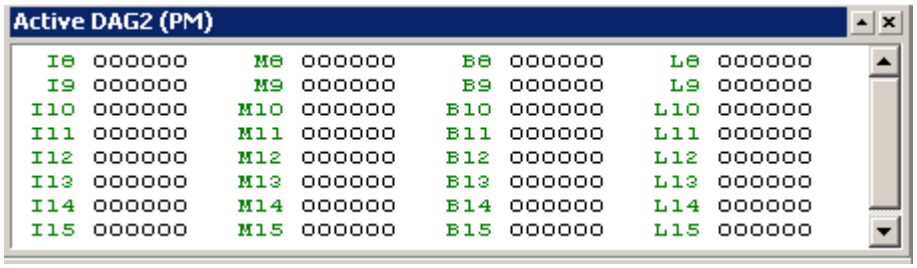
Регистры R0-15 наиболее часто используются при разработке ассемблерных программ. Основное их назначение – хранить результаты промежуточных вычислений до момента помещения результата в память процессора или внешнюю память.

- **Register->Core->DAG1(DM)** – выводит на экран окно с регистрами, содержащими 32-разрядные адреса памяти данных.



Данный набор 32-разрядных регистров используется при обращениях к памяти данных процессора (чтение и запись), при создании циклических буферов. Адресное пространство внешней памяти ограничено 2^{32} слов. Слово может быть от 1 до 4 байт.

- **Register->Core->DAG2(PM)** – выводит на экран окно с регистрами, содержащими 24-разрядные адреса памяти программы.



Данный набор 24-разрядных регистров используется при обращениях к внутренней памяти программы процессора (чтение и запись), при создании циклических буферов. Отсюда следует ограничение объема памяти программы 2^{24} слов (каждое слово в памяти программы имеет размер 6 байт).

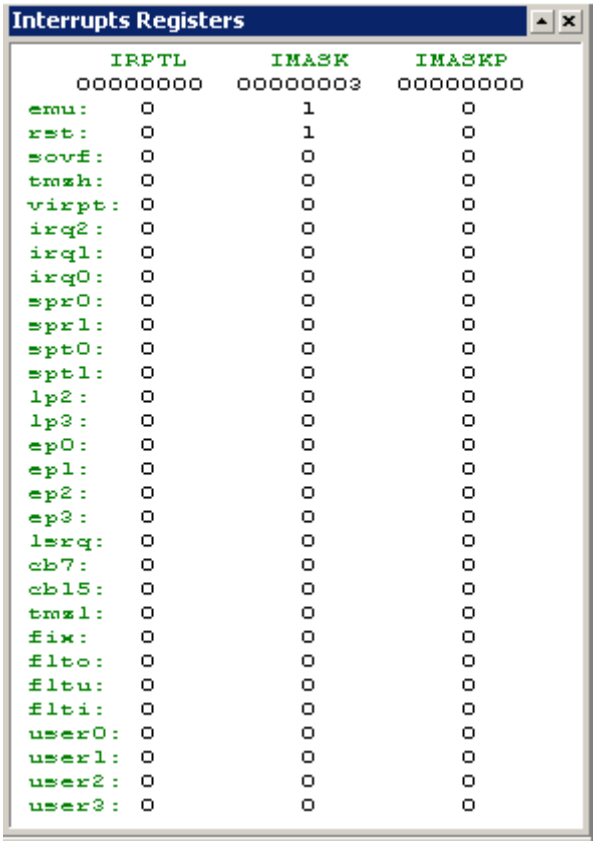
- **Register->core->Interrupts**

Окно содержит список всех прерываний процессора, за каждое из которых отвечает один из битов регистров IRPTL – регистр фиксирования прерываний, IMASK – маска прерывания, IMASKP – указатель маски прерывания (для вложенности).

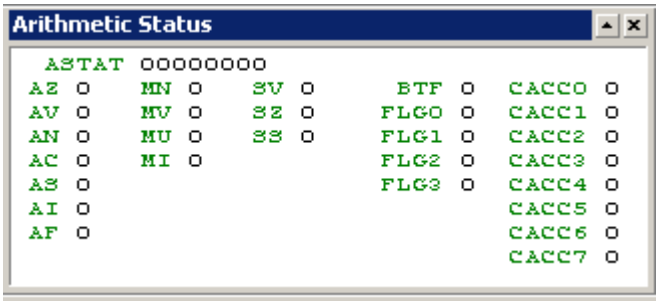
Прерывание необходимо, чтобы в определенных условиях прерывать работу основной программы, исполнять подпрограмму, состоящую из нескольких ассемблерных команд, а затем продолжить выполнение основной программы.

Регистр фиксирования прерывания (IRPTL) – это 32-разрядный регистр, который фиксирует прерывание. Он указывает на все прерывания, которые были обработаны, а также на те, обработка которых была отложена. Поскольку этот регистр доступен для чтения и записи, любое прерывание (за исключением сброса) может быть установлено или сброшено программно.

Не производите запись в бит сброса (бит 1) регистра IRPTL, потому что это переведет процессор в запрещенное состояние. Все прерывания, кроме сброса, могут быть маскированы. Маскирование означает запрещение генерации прерывания. Маскированные прерывания фиксируются в регистре IRPTL и, в случае их разрешения в дальнейшем, обрабатываются. Маскирование прерываний задается битами в регистре IMASK. Биты в IMASK точно соответствуют битам регистра IRPTL. Например, бит 10 в IMASK маскирует или разрешает то же самое прерывание, которое фиксируется битом 10 в IRPTL. Если бит в IMASK установлен в 1, то прерывание не маскировано и может быть сгенерировано. Если бит установлен в 0, прерывание маскировано и произойти не может. 32-разрядные регистры прерываний IRPTL, IMASK, IMASKP, каждому биту которых соответствует свое прерывание, изображены на рисунке (не маскированы только прерывания **emu**, **rst**).



- **Core->Status->ASTAT** – выводит на экран побитово содержимое регистра ASTAT.



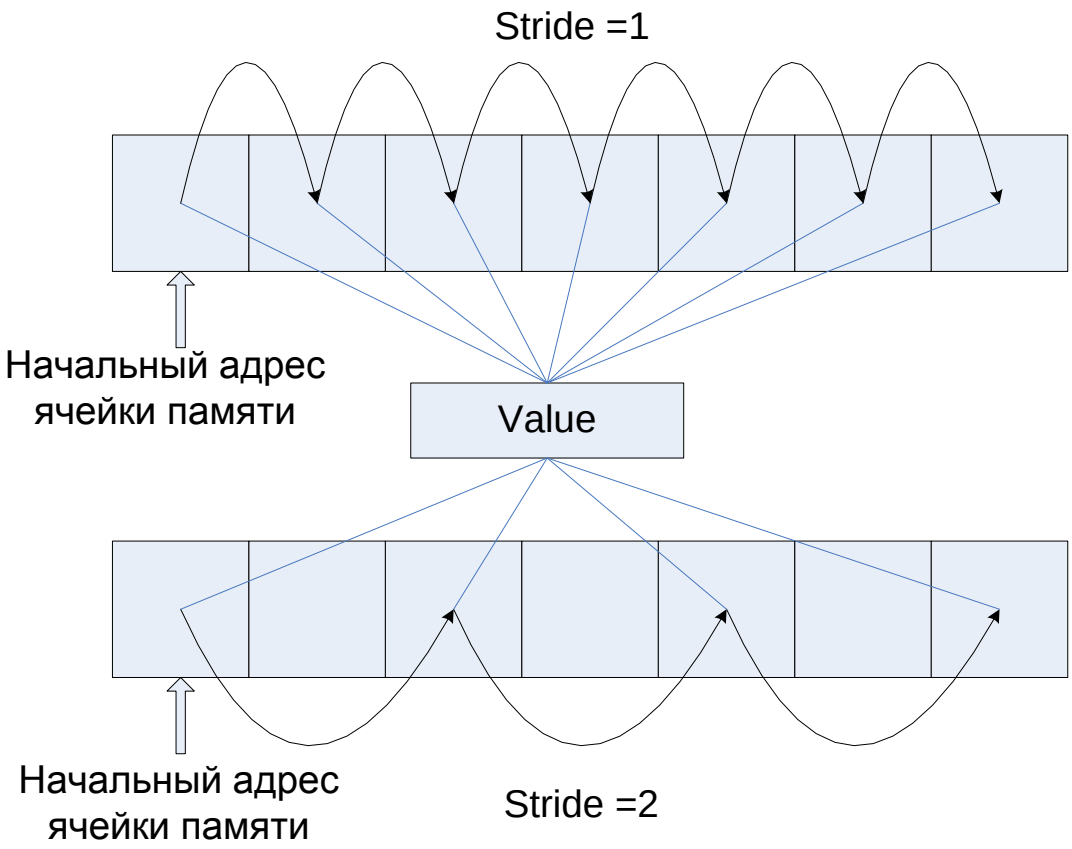
Флаги исключительных ситуаций (AZ, AV, AN, AC, AS, AI, AF) в регистре ASTAT, соответствующие определенным арифметическим операциям, проверяются после того, как операция выполнена. Например: после сложения, вычитания или операции сдвига процессор проверяет, больше или меньше нуля результат операции, не возникло ли в результате операции переполнение и т.д. На основании проверок выставляется соответствующий флаг в регистре ASTAT. ALU обновляет семь флагов исключительных ситуаций в регистре ASTAT в конце каждой операции (список всех операций АЛУ приведен в Приложении Б). Состояния этих флагов отражают результат самой последней операции ALU. Состояния флагов исключительных ситуаций можно использовать для отыскания скрытых ошибок в программе.

7. Меню **Memory**

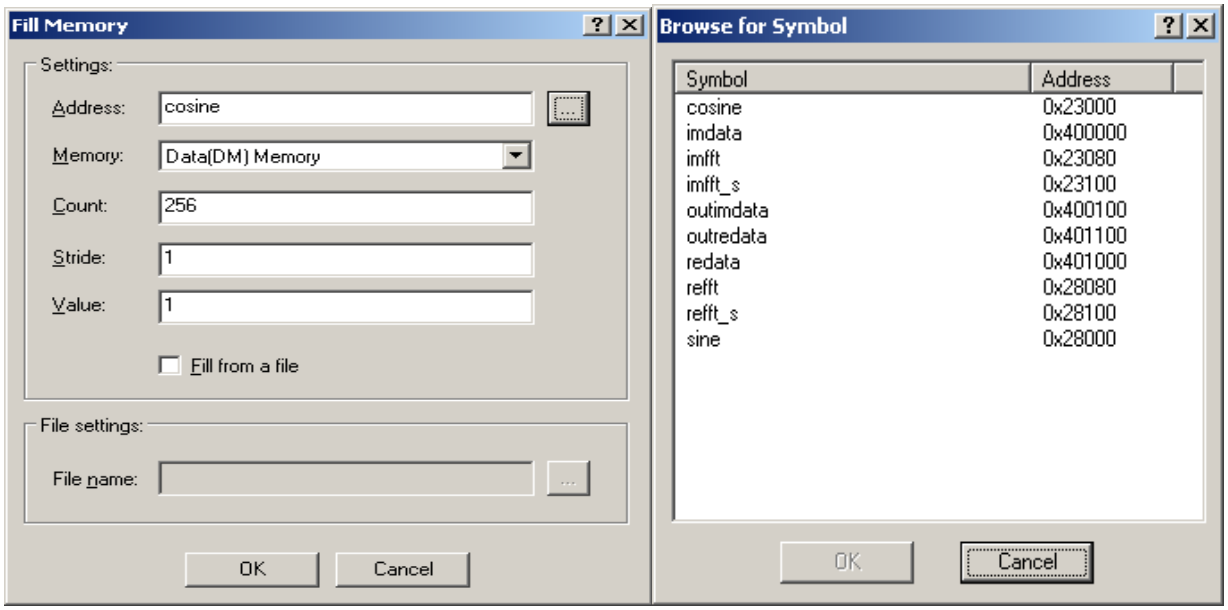
Наиболее важными элементами данного пункта меню являются:

- **Fill** – позволяет поместить по определенному адресу во внутреннюю или внешнюю память необходимое число или массив чисел.

В поле **Address** необходимо с помощью клавиатуры ввести адрес начала блока данных, в который будет производиться запись, либо нажать кнопку  и выбрать начальный адрес одного из массивов, объявленных в исходном тексте программы. Элемент управления **Memory** позволяет осуществлять смену областей памяти, что ускоряет поиск сегмента данных. Поля **Count** и **Stride** аналогичны полям в окне **Plot Configuration**. **Count** определяет количество ячеек памяти, которые необходимо заполнить значениями, содержащимися в поле **Value. Stride** определяет, через сколько ячеек памяти, начиная с начального адреса, будет производиться заполнение.

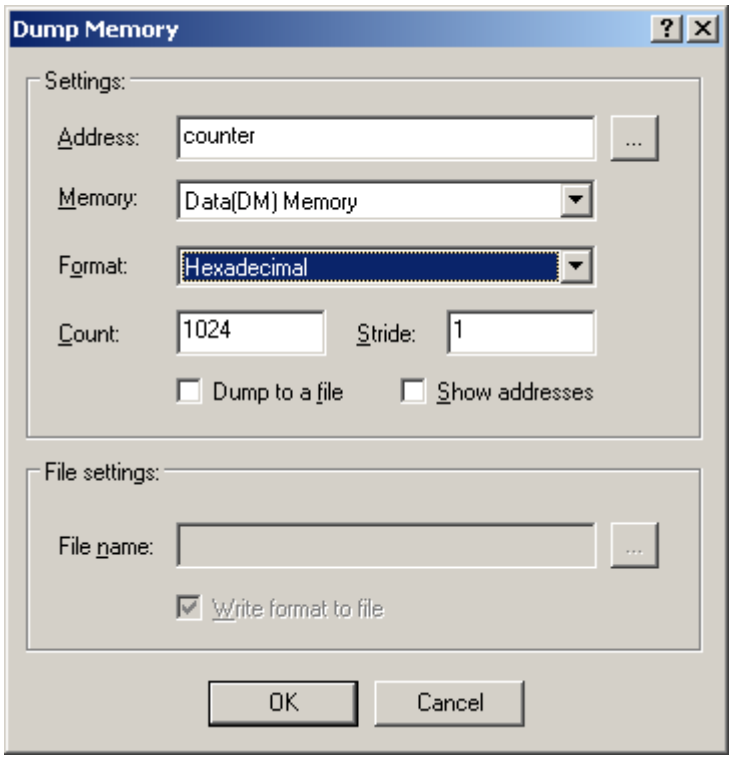


На следующем рисунке изображен случай, в котором, начиная с адреса переменной cosine (адрес 0x23000), расположенной в **Data Memory**, в 256 ячейках памяти последовательно (с шагом 1) будет записано значение 1.



Для записи сложных сигналов в память необходимо использовать элемент **Fill From File.** При этом данные будут записывать не из поля **Value**, а напрямую из файла с образцом сигнала.

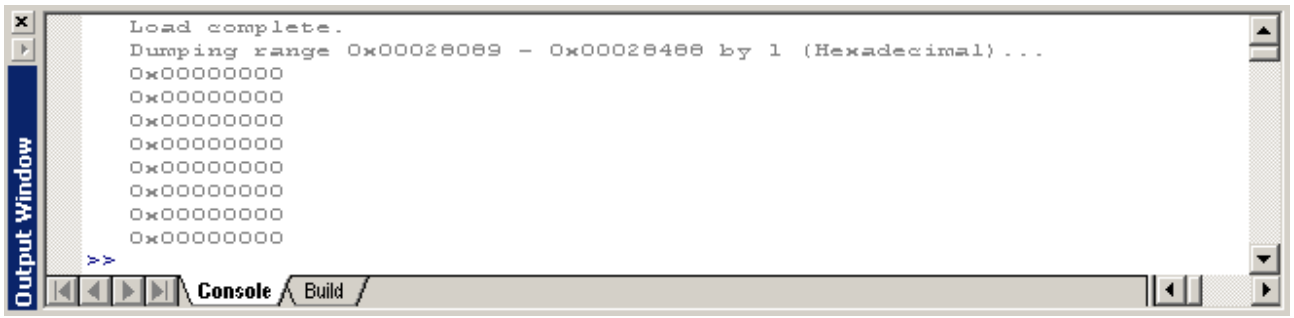
- **Dump** – позволяет считать содержимое определенного адреса внутренней или внешней памяти.



Окно **Dump Memory** имеет структуру, похожую на структуру окна **Fill Memory**, с той лишь разницей, что **Dump Memory** позволяет считывать данные из памяти процессора в окно **Output Window** либо напрямую в текстовый файл. Данные представляются в формате:

<адрес ячейки памяти> <содержимое ячейки памяти>.

Пример вывода содержимого области памяти в окно Output Window приведен на рисунке.



8. Меню **Debug**

- **Run** – запуск программы,

- **Halt** – остановка выполнения программы в текущем состоянии,

- **Reset** – сигнал программного сброса процессора,

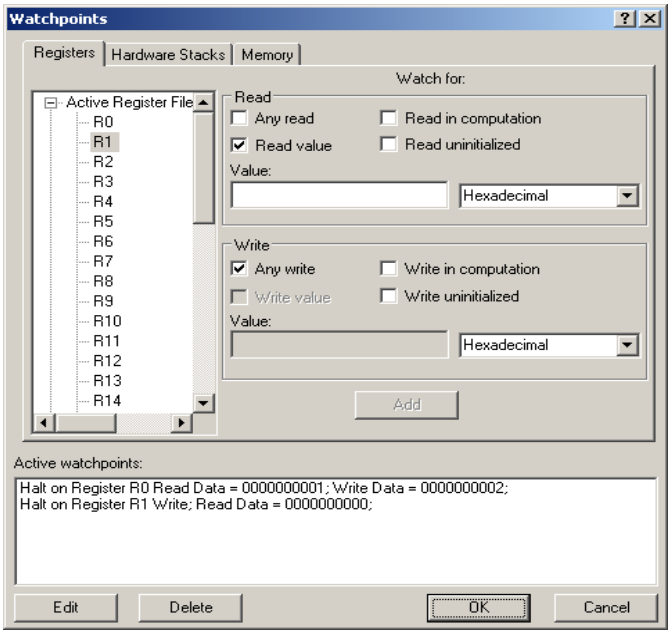
- **Restart** – повторный запуск программы,

- **Multiprocessor** – содержит элементы для отладки проектов для многопроцессорных систем.

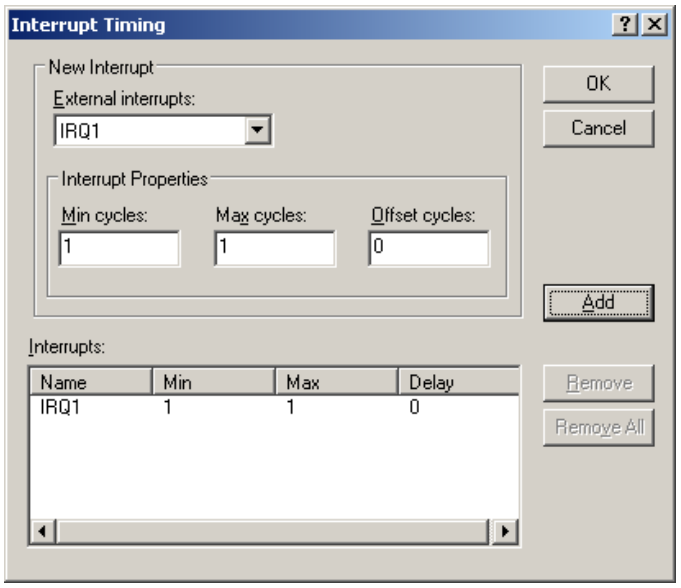
9. Меню **Settings**

- **Breakpoints** – позволяет установить в произвольном месте кода программы контрольную точку. Наличие контрольной точки означает, что при ее достижении работа программы временно приостанавливается до тех пор, пока программист не нажмет клавишу **F5** и работа программы не продолжится в нормальном режиме. Контрольные точки – одно из удобнейших средств отладки проектов.

- **Watch points** – позволяет приостановить работу программы при выполнении условия, предварительно заданного пользователем. Чтобы задать условие, необходимо на вкладке **Registers** окна **Watchpoints** выбрать имя регистра, содержимое которого будет фигурировать в условии. В поле **Value** вводится значение содержимого регистра, при котором программа должна остановить работу. После этого нажимается кнопка **Add**, и условие появляется в поле **Active watchpoint**. Для рисунка справедливо, что если чтении из регистра R0 значения 1 или записи в него 2 работа программы будет приостановлена. Даная опция полезна для обработки исключительных ситуаций.

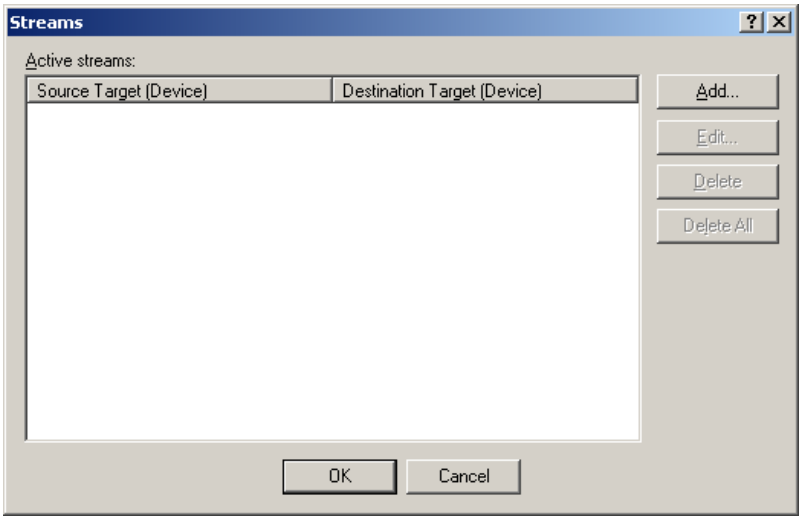


- **Interrupts** – позволяет имитировать любое прерывание для каждого из процессоров, контролировать время и частоту его появления.

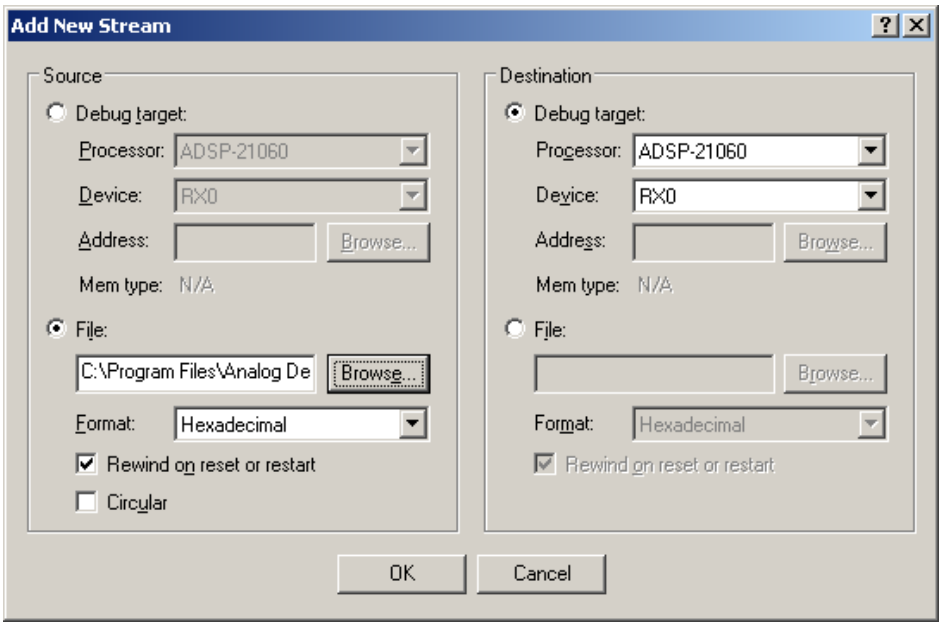


В элементе управления **External interrupt** окна **Interrupt Timing** выбирается прерывание, которое необходимо программно генерировать. Затем в элементах панели **Interrupt Properties** устанавливается интервал времени относительно момента запуска программы, в котором должно быть сгенерировано прерывание. Время возникновения случайно, отсчитывается в тактах процессора и распределено по равномерному закону в интервале от значения в поле **MIN_cycles** до значения в поле **Max_cycles.**

- **Streams** – позволяет моделировать потоки данных от различных устройств к процессору, и наоборот. Окно, появляющееся при выборе данного пункта меню, представлено на рисунке.



Для того чтобы создать новый поток, необходимо нажать кнопку **Add** в окне **Streams**, после чего на экране монитора появится окно **Add New Stream**.



На панели **Source** выбирается устройство, порождающее поток данных. Это может быть другой процессор, файл с данными либо любое другое устройство, передающее данные по одному из портов процессора. На панели **Destination** выбирается устройство – потребитель данных. Это может быть процессор, внешнее устройство либо файл. После настройки нажимается кнопка **OK.** Поток данных будет подключен к программе.

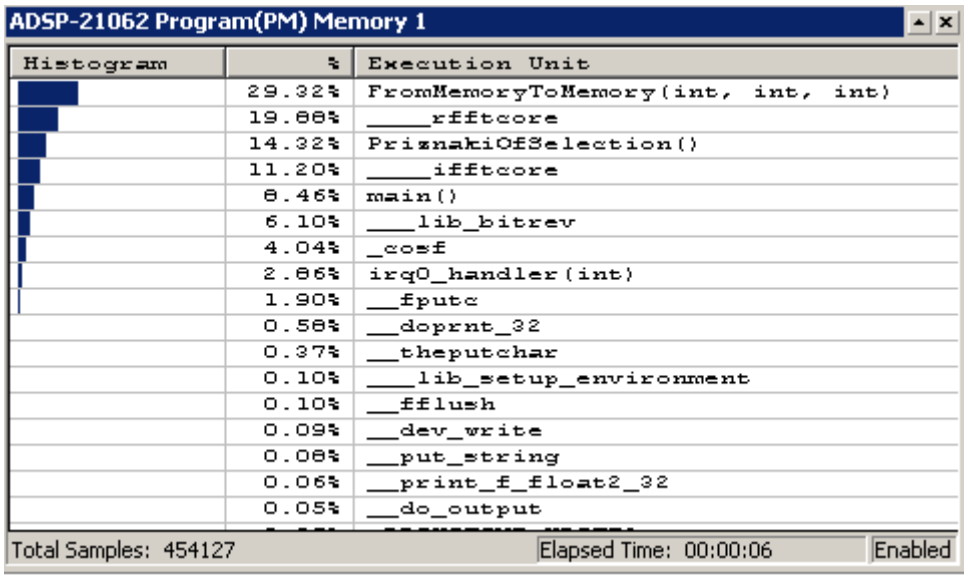
- **Preferences** – позволяет программисту определить свои настройки VisualDSP++. Данный пункт касается в основном визуальных эффектов и легко может быть освоен самостоятельно.

10. Меню **Tools**

- **Tools->Linear Profiling->New Profile** – создает окно **ADSP-2106x Program(PM) Memory 1**, в котором можно наблюдать гистограмму количества обращений программы по определенному адресу. В поле Histogramm непосредственно строится гистограмма. В поле «%» отображается процентное соотношение обращений в определенную область программы.

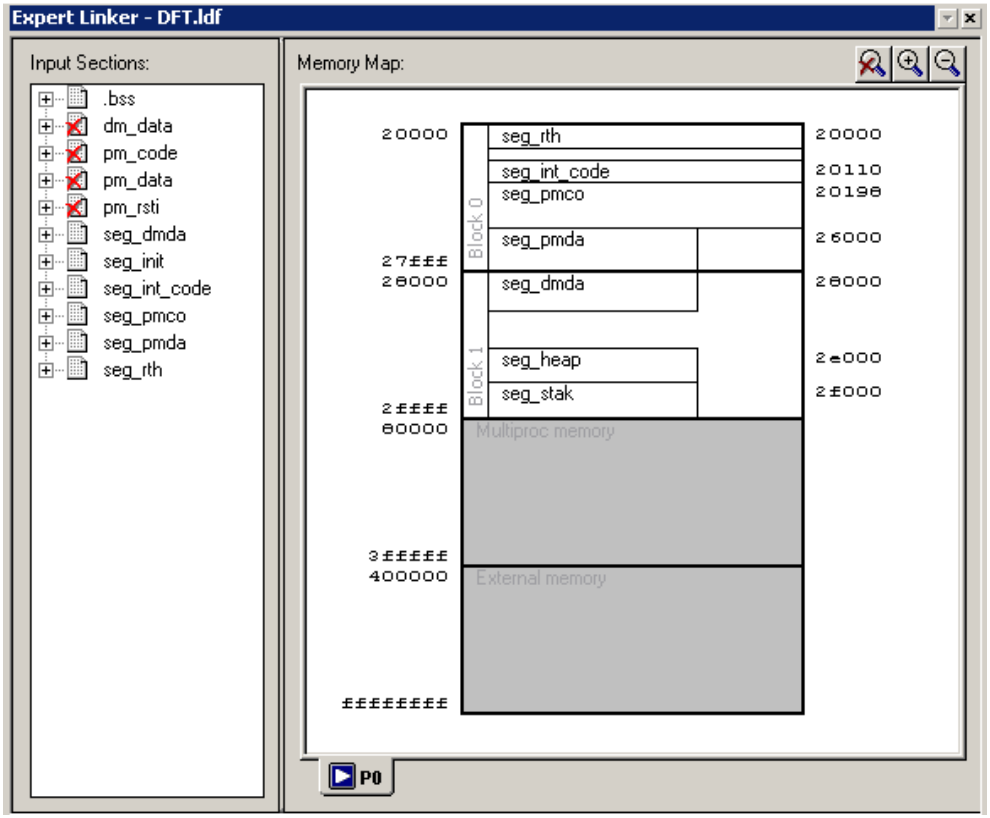
В поле **Execution Unit** отображается имя модуля программы.

Гистограмма строится по результатам полного либо частичного выполнения программы. Параметр **Total Samples** показывает время, прошедшее с момента запуска программы до момента остановки. Время задается в тактах DSP. С помощью параметра **Total Samples** удобно оценивать время работы определенного участка кода и путем последовательных итераций пытаться минимизировать его.

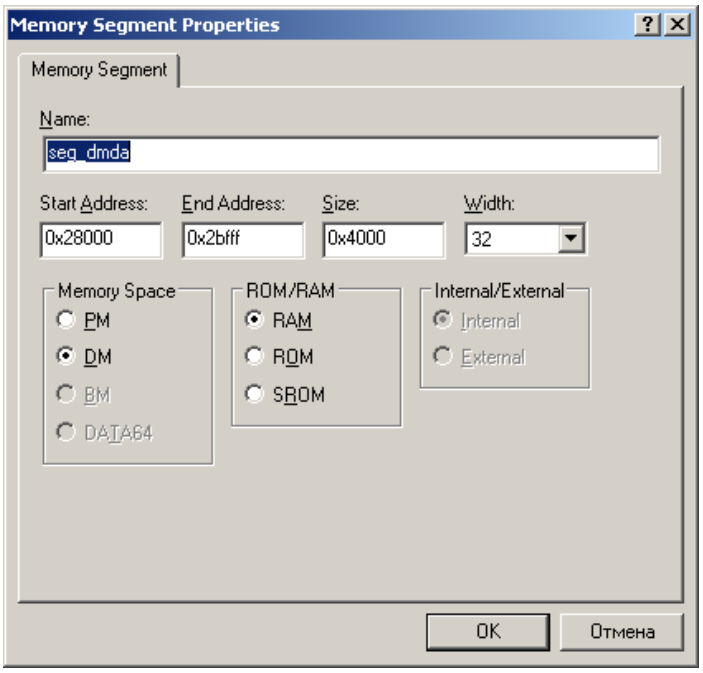


- **Tools->Expert Linker->Create LDF** – позволяет создать карту памяти процессора, разделить внутреннюю память процессора на секции и сегменты, каждая (каждый) из которых выполняет свою функцию.

При запуске данного пункта меню появится окно мастера создания ***.ldf** файла. Для создания карты памяти по умолчанию необходимо в каждом из последовательно появляющихся окон мастера нажать кнопки **ОК**. Результатом работы мастера будет окно **Expert Linker**. В поле **Input Section** будут описаны секции и сегменты, на которые будет разбито все адресное пространство внутренней памяти процессора. В поле **Memory Map** приведено физическое размещение сегментов во внутренней памяти с указанием адресов. Затененная часть карты памяти с адресами 80000 – 3fffff отведена для многопроцессорной системы. Адреса 400000 – ffffffff отведены под внешнюю память.



Параллельно с окном Expert Linker создается ***.ldf*** файл, который содержит описание карты памяти процессора, и автоматически присоединяется к проекту. Размеры сегментов внутренней памяти можно принудительно изменять. Для этого достаточно кликнуть правой кнопкой мышки по нужному сегменту и во всплывающем меню выбрать свойство Properties. В результате на экране ЭВМ появится окно, в котором можно изменить название сегмента, начальный и конечный адреса, установить размер слова данных в пределах сегмента. С помощью дополнительных переключателей можно изменить размещение сегмента. Можно создать сегмент как во внешней, так и во внутренней памяти.



На панели инструментов располагаются элементы управления, которые облегчают работу программиста, делают ее более эффективной и простой. Наиболее полезные из них приведены в таблице

	показать окно, отображающее ассемблерный код
	поставить точку принудительной остановки программы
	имитация сигнала Reset (аппаратный сброс)
	запустить программу (F5)
	собрать проект
	настройки проекта
	открыть проект
	добавить файл в директорию
	установить новую закладку
	перейти к следующей закладке
	убрать все закладки

1.5. Итоги раздела

В результате прочтения раздела мы получили общее представление о базовых возможностях, предоставляемых интерфейсной частью пакета Analog Devices VisualDSP 3.5. Научились считывать данные из памяти, записывать их в память, выводить временные и спектральные формы сигналов на экран монитора, моделировать потоки данных от различных устройств, создавать собственные панели, изучили удобные средства отладки. Полученные знания, дополненные материалами последующих разделов, позволят создать целостное впечатление о возможностях пакета.

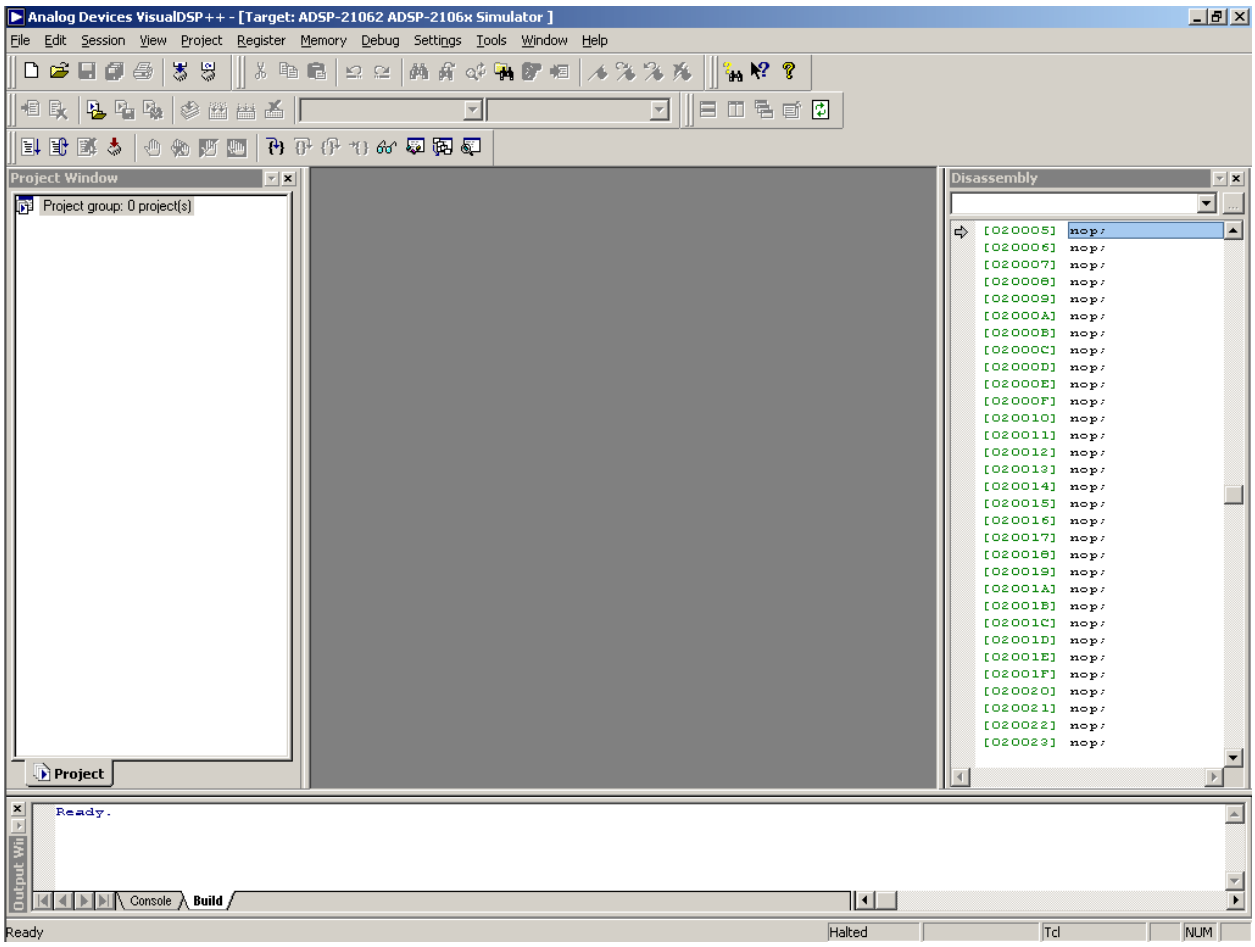
2. Основы программирования ЦСП

2.1. Цель раздела

После того как мы познакомились с пакетом Analog Devices VisualDSP 3.5 и изучили пользовательский интерфейс, настала пора освоить функциональную сторону пакета. В этом разделе нам предстоит создать свой проект, настроить его, подключить к проекту ассемблерный файл, рационально распределить память процессора. По окончании раздела мы должны создать типовой шаблон проекта, который будет использоваться во всех без исключения лабораторных работах и может быть использован при разработке реальных программ для процессоров фирмы Analog Devices.

2.2. Создание проекта

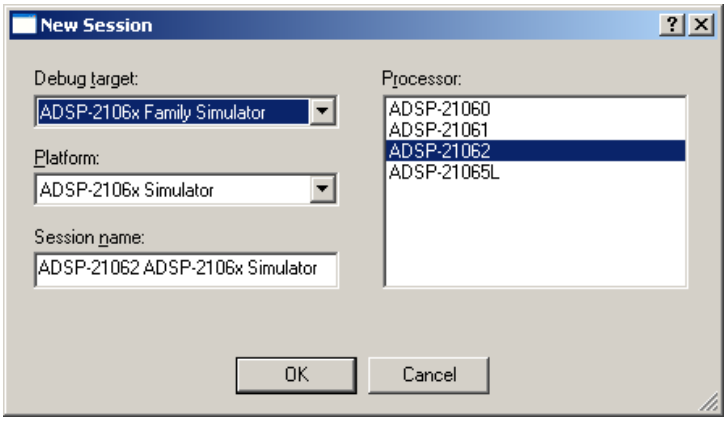
Приступая к созданию нового проекта, необходимо запустить программу Analog Devices VisualDSP 3.5. В результате на экране появится окно программы. Если программа запускается первый раз, то все дочерние окна программы будут пусты, только в окне **Disassembly** будут выведены адреса внутренней памяти процессора и **NOP** напротив каждого адреса.



NOP напротив адресов команд в окне **Disassembly** означает, что в ячейке памяти нет команды.

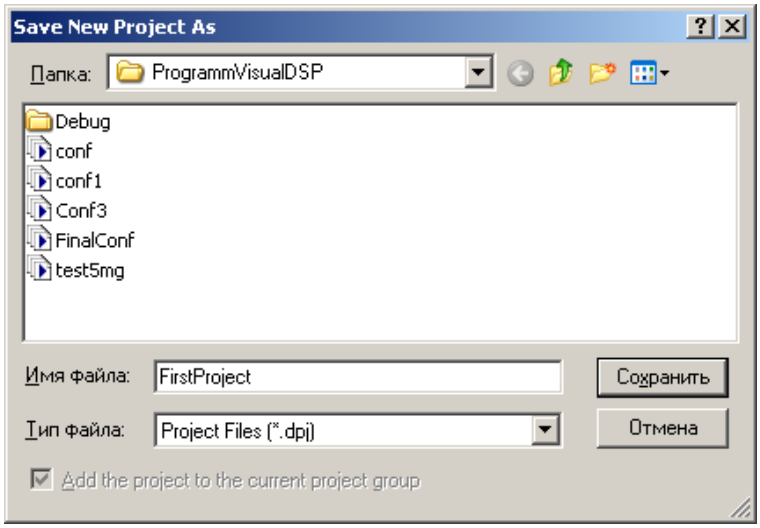
Программируя процессор, следует понимать, каким образом работает записанная в него программа. Программа представляет собой набор команд, которые процессор должен последовательно выполнять. Команды представляют 48-разрядные слова, которые могут быть записаны во внутреннюю память процессора. В каждую ячейку памяти процессора может быть записана только одна команда. После загрузки в процессор программа представляет собой последовательность 48–разрядных слов, хранящихся во внутренней памяти процессора. У каждой ячейки памяти процессора имеется адрес, обращаясь к которому процессор может считать, а затем и выполнить команду. Работа программы начинается с того, что процессор обращается к начальному адресу, в котором лежит первая команда программы. Оканчивается работа программы исполнением последней команды и переходом процессора в режим пониженного потребления питания.

Перед созданием нового проекта необходимо выбрать процессор, для которого проект будет создаваться. Для этого необходимо выбрать пункт меню **Session->NewSession**, в появившемся окне выбрать процессор **ADSP-21062** и нажать кнопку **ОК**.



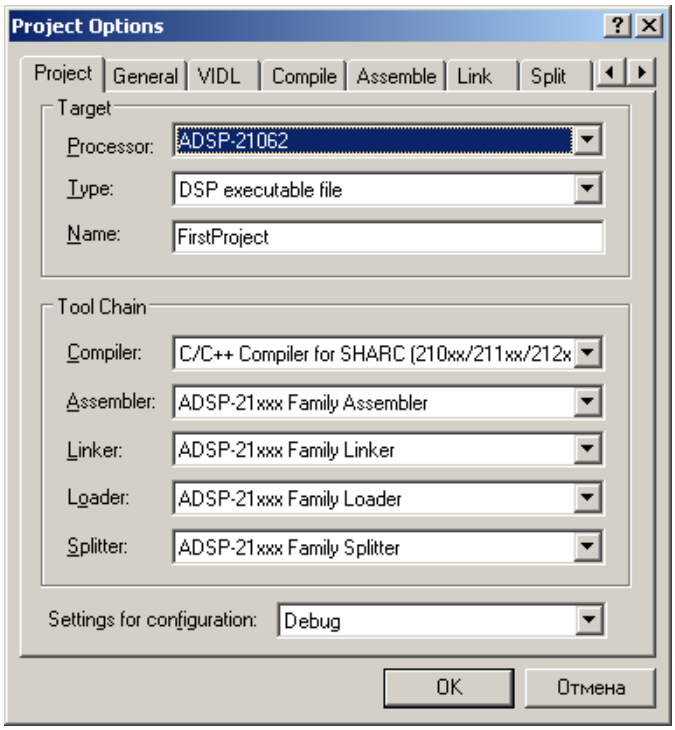
После этого процессор **ADSP-21062** воспринимается пакетом **VisualDSP** как процессор, для которого создается проект. Изменится также содержимое всех пунктов меню, зависящих от типа процессора. Особенно это касается пунктов Memory и Register.

Для того чтобы создать новый проект, необходимо в меню **Project** выбрать пункт **New Project,** в результате этого на экране монитора появится диалоговое окно.



В данном окне необходимо ввести название проекта **FirstProject** и нажать кнопку **«Сохранить».**

После чего на экране появится окно настроек проекта. Количество настроек велико, но в рамках проведения курса лабораторных работ достаточно остановиться на описании наиболее важных.



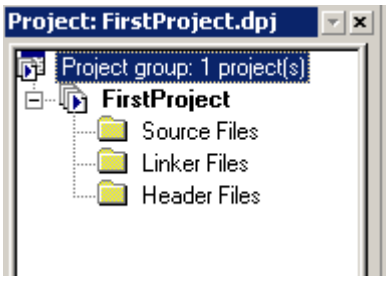
Элемент управления **Project->Processor** позволяет выбрать процессор семейства **SHARC:**

ADSP-21020	ADSP-21060	ADSP-21061	ADSP-21062
ADSP-21065L	ADSP-21160	ADSP-21161	ADSP-21261
ADSP-21262	ADSP-21266	ADSP-21267	ADSP-21363

Для создания проекта выберите один из процессоров, помеченных в таблице, например **ADSP-21062**. Они отличаются лишь объемом внутренней памяти. У процессора **21060** общая внутренняя память 4 Мбита, а у процессора **21062** – 2 Мбита.

Элемент управления **Project->Type** позволяет выбрать тип файла, который будет создаваться при компиляции проекта. Возможен выбор между исполняемым файлом, загрузочным модулем, объектным модулем или библиотекой. С точки зрения освоения курса лабораторных работ, наиболее важными являются исполняемый и загрузочный файлы.

После нажатия кнопки **ОК** окно **Project** перестанет быть пустым и в нем появится новый проект **FirstProject**.

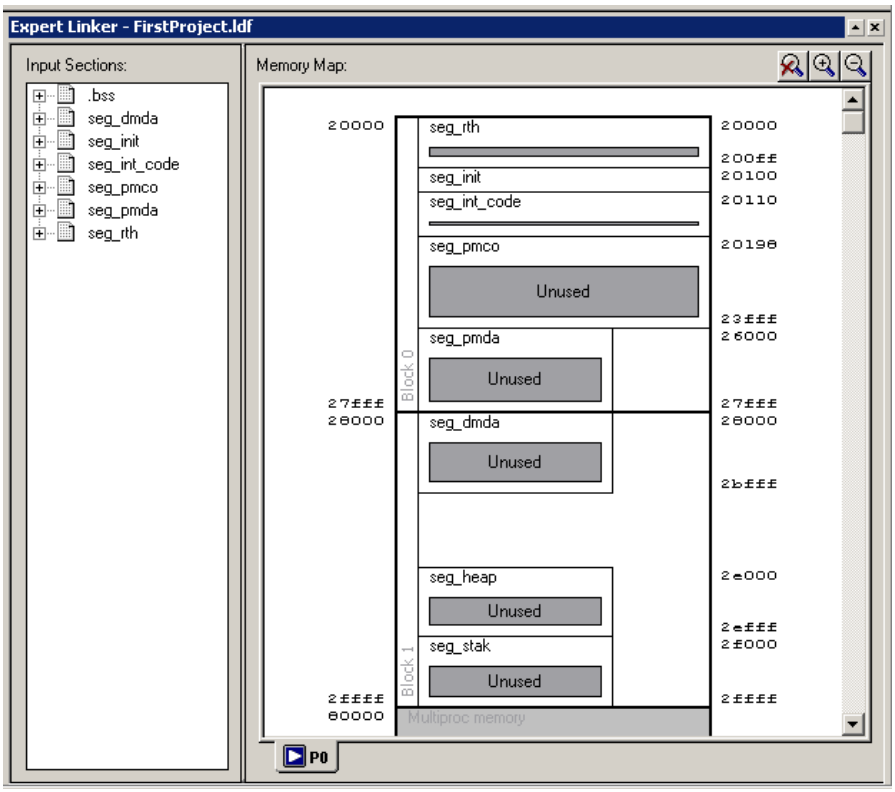


На данном этапе проект **FirstProject** включает в себя три пустых папки. Первая из них **Source Files** предназначена для файлов исходного кода проекта (.asm, .cpp, .c). Вторая **Linker Files** предназначена для хранения файла карты памяти (.ldf). Третья **Header Files** содержит заголовочные файлы проекта. В заголовочных файлах обычно хранятся описания констант, стандартных обозначений, переменных, классов.

Чтобы проект можно было скомпилировать и превратить в рабочую программу, необходимо создать файл **CodeFirst.asm**, который будет содержать исходный ассемблерный код, и добавить его к проекту. Для создания файла необходимо выбрать пункт меню **File->New**. В результате на экране монитора появится окно нового документа.

Используя команду **File->Save As,** необходимо сохранить документ под именем **CodeFirst.asm**. Расширение **.asm** указывать обязательно, иначе файл будет сохранен без расширения. После этого, воспользовавшись командой **Project->Add To Project->File** или иконкой  , добавить файл **CodeFirst.asm** к проекту. Он автоматически появится в проекте в папке **Source Files**.

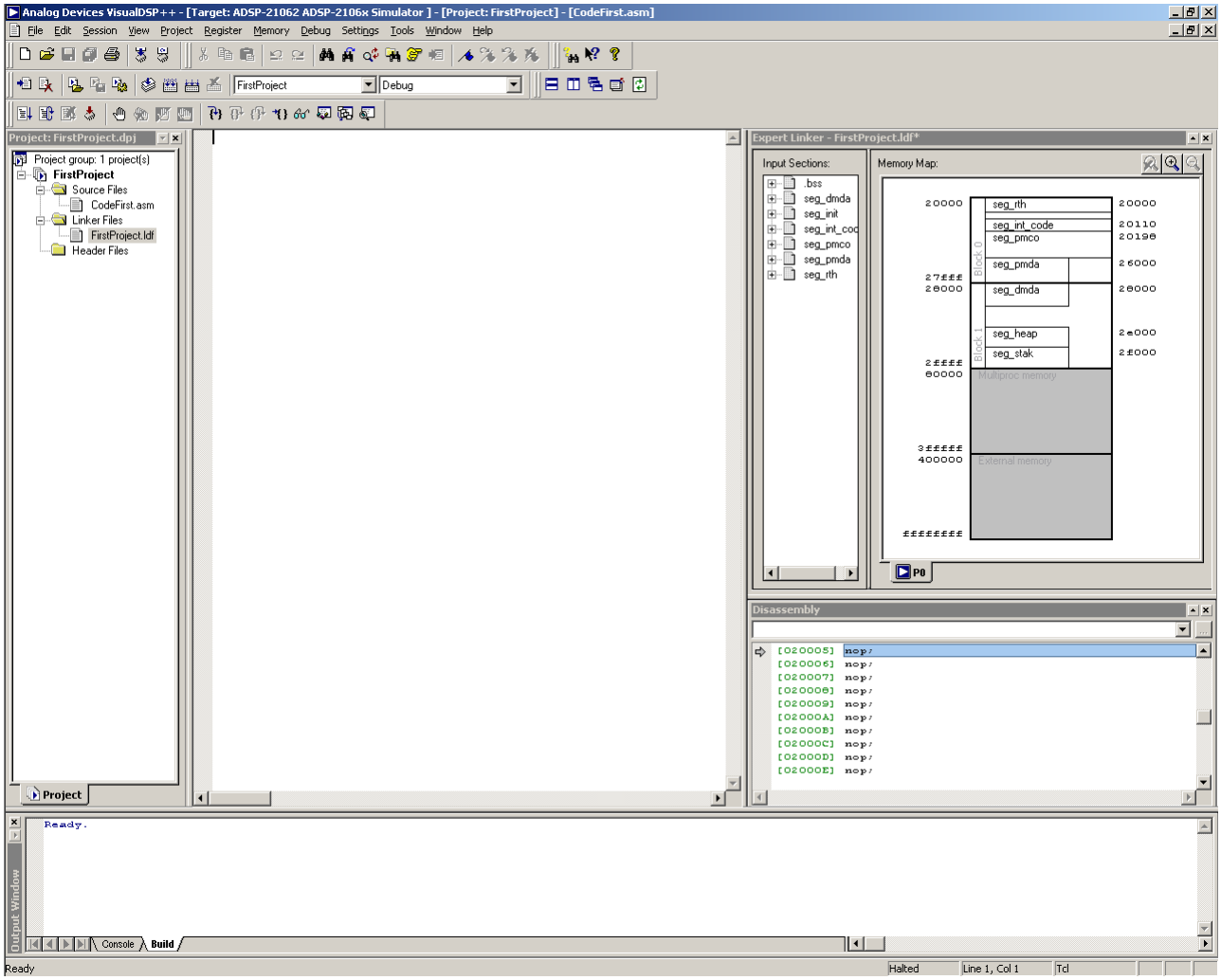
Для создания файла компоновщика необходимо выбрать пункт меню **Tools->Expert Linker->Create LDF**. Далее необходимо выполнять указания мастера создания файла компоновщика, во всех окнах последовательно нажимая кнопки **ОК** или **Next**. Результатом работы мастера будет файл **FirstProject.ldf** в каталоге проекта **Linker Files** и окно **Expert Linker** на экране монитора.



В данном окне можно найти полное описание внутренней памяти процессора, которая разбита на ряд сегментов, назначение которых более подробно будет рассмотрено в следующем разделе. Из рисунка видно, что все сегменты обозначены как **Unused** − «Неиспользуемые». Это нормально, поскольку мы не написали еще ни одной строчки исходного кода проекта и не использовали ни один сегмент памяти программы или данных.

Создание «скелета» проекта можно считать завершенным.

Если после всех операций окно **VisualDSP 3.5** на экране вашего монитора имеет такой же вид, как на рисунке, то можно считать, что все выполнено правильно и проект готов к написанию исходного кода.



Теперь можно закрыть окно Expert Linker и приступить к созданию первой программы на языке ассемблер.

2.3. Написание первой программы на ассемблере

2.3.1. Постановка задачи, отыскание методов решения

Для того чтобы написать программу на языке ассемблер, необходимо четко представлять структуру будущей программы. Задача, решением которой является программа, должна быть четко сформулирована. До написания первой строчки исходного кода необходимо иметь полное представление о том, каким образом программа будет реализована.

Допустим, по условию задания программа должна:

- сигнализировать о своей работоспособности внешним устройствам;

- считать данные по адресу 0x28000 и 0x28001;

- произвести простейшие арифметические операции со считанными данными (сложение, вычитание, умножение, сдвиг влево и вправо на 1 разряд);

- перейти в режим пониженного энергопотребления.

Каким образом решить поставленные задачи?

Шаг первый

Необходимо внимательно изучить описание процессора, для которого пишется программа (в нашем случае **ADSP-21062**). В результате необходимо понять, по какому адресу во внутренней памяти процессора должна быть загружена программа. Из описания мы узнаем, что после загрузки программы в процессор и подачи сигнала **RESET** должен быть осуществлен переход в начало программы. Для процессоров **ADSP-21062** начальным адресом программы считается 0x20198 (начальный адрес сегмента **seg_pmco**). Этот переход на языке ассемблер необходимо описывать программисту. Из описания процессора и команд встроенного ассемблера, мы узнаем, что при включении питания процессор начинает последовательно считывать команды с адреса 0x20004. Данный адрес расположен в сегменте прерываний (**seg_rth**). Из этого следует, что после загрузки процессора необходимо совершить переход от адреса 0x20004 к 0x20198. Решение первой задачи ясно: необходимо найти встроенную команду перехода процесса выполнения программы с одного адреса на другой. Такой командой является команда **JUMP**. В программе команда **JUMP** должна описываться следующим образом:

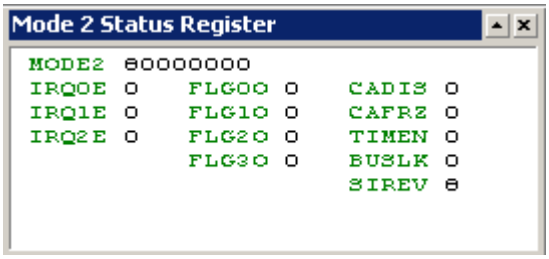
JUMP(addr), где addr − 24-разрядный адрес ячейки памяти программы.

JUMP(Label), где Label − имя метки, помещенной в тексте программы по определенному адресу.

Шаг второй

Первое действие, которое должна совершить будущая программа после начала работы − это сигнализировать о том, что она загружена. Как это реализовать? Для этого опять читаем описание процессора и узнаем, что процессор может использовать один из четырех выводов, называемых флагами, для сообщения внешним устройствам о своем состоянии. Также мы узнаем, что флаги управляются посредством регистра **ASTAT**. В данном регистре под это отведены четыре бита. Помимо этого, можно настраивать флаги как входы или выходы. В случае если флаги настроены как входы, процессор получает информацию от внешних устройств; если как выходы, то процессор сам сигнализирует внешним устройствам.

Из описания процессора мы узнаем, что для настройки флага как вход необходимо установить бит **FLG0O, FLG1O, FLG2O, FLG3O** в регистре **Mode2** в нуль. Для настройки флага как выход необходимо установить соответствующий бит в единицу. В исходном состоянии все биты **FLG(0−3)O** нулевые. Чтобы посмотреть содержимое регистра **Mode2**, необходимо активировать пункт меню **Register->Core->Status->Mode2**. В результате на экране ЭВМ появится окно **Mode 2 Status Register:**



Для установки одного из битов регистра используется команда:

BIT SET <Имя регистра > <Имя Бита>.

Для примера настроим **FLG0O** и **FLG1O** как выходы, а **FLG2O** и **FLG3O** как входы:

bit set mode2 FLG0O; //флаг настроен как выход

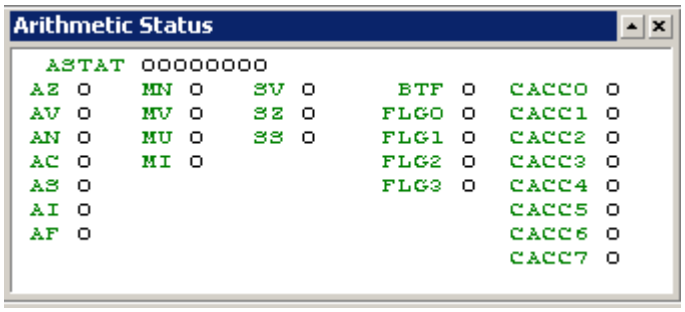
bit set mode2 FLG1O; // флаг настроен как выход

bit clr mode2 FLG2O; // флаг настроен как вход

bit clr mode2 FLG3O; // флаг настроен как вход

Для того чтобы сигнализировать внешним устройствам, что программа загружена, необходимо установить бит, соответствующий первому и второму флагам в регистре **ASTAT**.

Чтобы посмотреть содержимое регистра **ASTAT**, необходимо активировать пункт меню **Register->Core->Status->ASTAT**. В результате на экране ЭВМ появится окно **Arithmetic Status:**



Для установки битов необходимо по аналогии с регистром **Mode 2** использовать команду **BIT SET**, только в качестве имени регистра использовать **ASTAT**, а в качестве имени битов **ASTAT_FLG0** и **ASTAT_FLG1**:

bit set astat ASTAT_FLG0; // установка флага 0

bit set astat ASTAT_FLG1; // установка флага 1

Задача сигнализации решена.

Шаг третий

Как считать данные с внутренней памяти процессора? Куда их поместить? Для этой цели в процессоре предусмотрены регистры специального назначения, позволяющие обращаться к адресам ячеек памяти, хранить считанные значения и результаты вычислений.

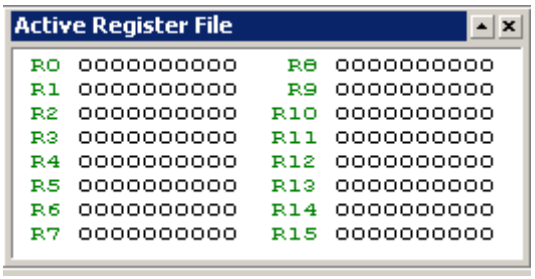
К таким регистрам относятся:

1) Активные регистры данных

R0–15 – для хранения данных с фиксированной точкой,

F0–15 – для хранения данных с плавающей точкой.

Чтобы просмотреть содержимое регистров общего назначения, необходимо использовать пункт меню **Register->Core->Register File**:



2) Активные регистры адреса

Но как поместить содержимое ячейки памяти в регистр общего назначения? Для этого предусмотрены генераторы адреса данных. Генераторы адреса данных (**DAG**) обеспечивают адресацию при передаче данных между памятью и регистрами. Два генератора адреса данных позволяют процессору одновременно выводить адреса для двух операндов при выполнении операций считывания или записи. **DAG1** генерирует 32-разрядные адреса памяти данных. **DAG2** генерирует 24-разрядные адреса памяти программы. Каждый генератор содержит восемь регистров адреса **I(0–7)**, восемь регистров модификации **M(0–7)**. Указатель, используемый для косвенной адресации, может модифицироваться значением определенного регистра перед обращением к памяти (предмодификация) или после обращения к памяти (постмодификация).

I(0–7) – может хранить адрес ячейки памяти данных (**DAG1**),

M(0–7) – может хранить целочисленный модификатор для адреса памяти данных,

I(8-15) – может хранить адрес ячейки памяти программы (**DAG2**),

M(8-15) – может хранить целочисленный модификатор для адреса памяти программы.

В процессоре предусмотрены операции чтения из памяти данных.

R1=DM(I0,M0) – операция постмодификации адреса.

В данном случае процессор считает данные из ячейки памяти с адресом **I0**, поместит их в регистр **R1**, а затем значение **I0** увеличится на значение, содержащееся в **M0**.

R1=DM(M0,I0) – операция предмодификации адреса.

В данном случае процессор изменит значение адреса, содержащееся в **I0** на величину **M0**, а затем считает данные из ячейки памяти с адресом **I0+M0** и поместит их в регистр **R1**.

В процессоре предусмотрены операции чтения из памяти программы:

R1=DM(I8,M8) – операция постмодификации адреса,

R1=DM(M8,I8) – операция предмодификации адреса.

Теперь у нас есть всё необходимое для того, чтобы считать данные из памяти в регистры общего назначения.

Шаг четвертый

Как реализовать вычислительные операции сложения, вычитания, умножения, сдвига влево и вправо на 1 разряд? В справочнике вычислительных операций процессора приведены все перечисленные операции:

- операция суммирования **Rn = Rx + Ry** (АЛУ);
- операция вычитания **Rn = Rx – Ry** (АЛУ);
- операция умножения **Rn = Rx *Ry** (УМНОЖИТЕЛЬ);
- операция сдвига **Rn= ASHIFT Rx BY <data8>** (АЛУ).

Операция сдвига требует пояснения: содержимое регистра **Rx** будет сдвинуто влево на **data8**. Результат сдвига будет помещен в регистр Rn.

Пример:

0000 0000 0000 0000 0000 0000 0000 *0110*

Rn = ASHIFT Rx BY 8;

0000 0000 0000 0000 0000 *0110* 0000 0000

Шаг пятый

Каким образом включить режим пониженного энергопитания? Для этого в процессоре предусмотрены специальные команды IDLE и IDLE16, которые останавливают ядро процессора до прихода внешнего прерывания или прерывания по таймеру.

Теперь мы готовы начать написание исходного кода программы.

2.3.2. Написание исходного текста программы

Начнем написание программы с описание сегмента, хранящего вектор прерываний. Отметим, что в векторе прерываний зарезервировано место под 64 прерывания.

Листинг программы, описывающей вектор прерываний, приведен ниже.

```
//описывает начало сегмента, в котором хранится вектор прерываний
  .segment/pm seg_rth;
    //0x20000 четыре ячейки памяти без команд, резервное прерывание
 nop; nop; nop; nop;
    //0x20004 четыре ячейки памяти прерывания по сигналу RESET
 ___lib_RSTI:
    nop;
    // переход к метке first в сегменте seg_pmco по адресу 0x20198
    jump first;
    nop;
    nop;
    //0x20008 четыре ячейки памяти, резервное прерывание
    nop; nop; nop; nop;
    //Вектор  переполнения стека
___lib_SOVFI:   RTI;RTI;RTI;RTI;
    // Вектор прерывания по таймеру (высшего приоритета)
___lib_TMZHI:  RTI;RTI;RTI;RTI;
    //Векторы  внешних прерываний:
___lib_VIRPTI:  RTI;RTI;RTI;RTI;
___lib_IRQ2I:   RTI;RTI;RTI;RTI;
___lib_IRQ1I:   RTI;RTI;RTI;RTI;
___lib_IRQ0I:   RTI;RTI;RTI;RTI;
    // Зарезервированное прерывание
```

```
    NOP;NOP;NOP;NOP;
    //Векторы для каналов DMA последовательного порта:
___lib_SPR0I:   RTI;RTI;RTI;RTI;
___lib_SPR1I:   RTI;RTI;RTI;RTI;   //link buffer0
___lib_SPT0I:   RTI;RTI;RTI;RTI;
___lib_SPT1I:   RTI;RTI;RTI;RTI;   //link buffer1
    //Векторы для каналов портов связи:
___lib_LP2I:    RTI;RTI;RTI;RTI;
___lib_LP3I:    RTI;RTI;RTI;RTI;
    //Векторы для каналов DMA внешних портов:
___lib_EP0I:    RTI;RTI;RTI;RTI;   //link buffer4
___lib_EP1I:    RTI;RTI;RTI;RTI;
___lib_EP2I:    RTI;RTI;RTI;RTI;
___lib_EP3I:    RTI;RTI;RTI;RTI;
    //      Link
___lib_LSRQ:    RTI;RTI;RTI;RTI;
    //  Вектор DAG1 буфера 7 (кругового)
___lib_CB7I:    RTI;RTI;RTI;RTI;
    //  Вектор DAG1 буфера 7 (кругового)
___lib_CB15I:   RTI;RTI;RTI;RTI;
    // Вектор прерывания по таймеру (низшего приоритета)
___lib_TMZLI:   RTI;RTI;RTI;RTI;
_FIXI:      RTI;RTI;RTI;RTI;   // 24 − Fixed-point overflow
_FLTOI:     RTI;RTI;RTI;RTI;   // 25 − Floating-point overflow exception
_FLTUI:     RTI;RTI;RTI;RTI;   // 26 − Floating-point underflow exception
_FLTII:     RTI;RTI;RTI;RTI;   // 27 − Floating-point invalid exception
_SFT0I:     RTI;RTI;RTI;RTI;   // 28 − User software interrupt 0
_SFT1I:     RTI;RTI;RTI;RTI;   // 29 − User software interrupt 1
_SFT2I:     RTI;RTI;RTI;RTI;   // 30 − User software interrupt 2
_SFT3I:     RTI;RTI;RTI;RTI;   // 31 − User software interrupt 3
```

.endseg; *// окончание сегмента*

После того как вектор прерываний заполнен, необходимо начать описание основной части программы. Начать ее следует с того, чтобы поместить в ячейки памяти с адресами 0x28000 и 0x28001 данные, например 7 и 8. Для этого необходимо описать в программе сегмент памяти **seg_dmda**, содержащий всего две переменные **index** и **index1**.

.SECTION/DM seg_dmda;

.var index=7;

.var index1=8;

.endseg;

Теперь можно приступать к написанию основного тела программы.

.segment/PM seg_pmco;

first:

bit set mode2 FLG0O; *//направление флага на передачу*

bit set mode2 FLG1O; *//направление флага на передачу*

bit clr mode2 FLG2O; *//направление флага на прием*

bit clr mode2 FLG3O; *//направление флага на прием*

bit set astat ASTAT_FLG0; *//установление флага 0*

bit set astat ASTAT_FLG1; *//установление флага 1*

i1=0x28000; *// Запись адреса 0x28000 в регистр i1*

m1=1; *// В регистр m1 значение 1*

r1=dm(i1,m1); *// чтение данных из ячейки памяти с адресом 0x28000, при этом значение i1 увеличивается на m1 и становится равным 0x28001*

r2=dm(i1,m1); *// чтение данных из ячейки памяти с адресом 0x28001, при этом значение i1 увеличивается на m1 и становится равным 0x28002*

r3=r1+r2;

r4=r1-r2;

r5= r1*r2;

r6= ASHIFT R1 BY 0x1;

r7= ASHIFT R2 BY 0x1;

idle;

.endseg;

Программа написана и включена в проект. Теперь можно собрать проект. Для этого нажимаем клавишу **F7** и ждем до тех пор, пока в окне **Output Window** не появится надпись:

Loading:

"X:\Level_3\MicroPocess\ProgMICRO\FirstProg\Debug\FirstProject.dxe"...

Load complete.

В данном случае проект собран без ошибок и готов к работе.

После нажатия клавиши F7 в окне Output Window может появиться надпись:

Previous errors prevent assembly

Assembler totals: 1 error(s) and 0 warning(s)

Tool failed with exit/exception code: 1.

Build was unsuccessful.

Необходимо проверить исходный код программы, устранить ошибку **1 error(s)** и нажать клавишу **F7** заново.

2.3.3. Как устранить ошибку в исходном коде программы?

Вопрос: Как устранить ошибку?

Ответ: При ошибке в исходном коде программы в окне **Output Window** появится надпись:

----------Configuration: FirstProject - Debug----------

"C:\Program Files\Analog Devices\VisualDSP 3.5 32-Bit\easm21k.exe" .\CodeFirst.asm -proc ADSP-21062 -g -o .\Debug\CodeFirst.doj

[Error ea1007] ".\CodeFirst.asm":86 [column 11] Syntax Error at "T".

Previous errors prevent assembly

Assembler totals: 1 error(s) and 0 warning(s)

Tool failed with exit/exception code: 1.

Build was unsuccessful.

Надпись говорит о том, что в тексте программы одна ошибка **1 error(s)** и ноль предупреждений об опасности **0 warning(s)**.

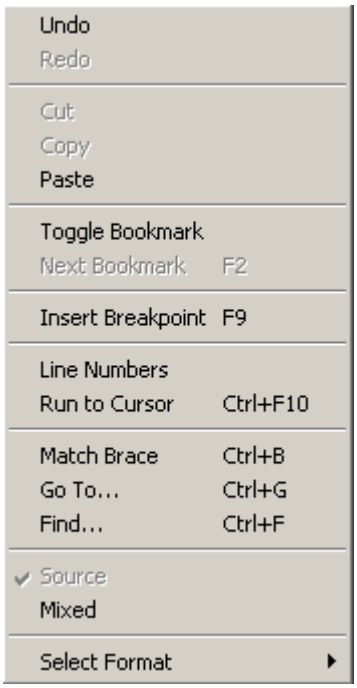
Вопрос: Где искать ошибку?

В этом нам поможет подсказка, которая сигнализирует, на какой строке и в каком столбце исходного кода обнаружена ошибка.

[Error ea1007] ".\CodeFirst.asm":86 [column 11] Syntax Error at "T".

В данном случае ошибка обнаружена на 86-ой строке и 11-ом столбце.

Теперь необходимо найти 86-ю строку, для этого можно просчитать строки с начала файла, либо наведя курсор мыши на поле редактора кода нажать правую кнопку. В результате чего появится всплывающее меню вида:



Выбрать пункт Line Numbers. При этом в окне редактора кода слева от каждой строчки появится ее номер. После чего находим 86-ю строку, 11-й столбец и ищем ошибку.

```
82
83 r3=r1+r2;
84 r4=r1-r2;
85 r5= r1*r2;
86 r6= ASHIF T R1 BY 0x1;
87 r7= ASHIFT R2 BY 0x1;
88 idle;
89 .endseg;
```

В тексте программы умышленно вместо команды **ASHIFT** было написано **ASHIF T**, что и породило ошибку. Исправив ошибку, необходимо нажать кнопку **F7** и убедиться в том, что проект скомпилирован без ошибок.

2.3.4. Проверка правильности работы программы

Результатом работы программы должны быть:

- Флаги **ASTAT_FLG**0 и **ASTAT_FLG1** в регистре **ASTAT** равны единице;

- считанные из памяти значения должны быть равны 7 и 8 и помещены в регистры **R1** и **R2**.

- результаты вычислительных операций должны быть следующими:

R3 = 7+8 =15;

R4 = 7-8 = -1;

R5 = 7*8 = 56;

R6 = 14 (в двоичном виде 1110) = 7 (в двоичном виде 111)<<1;

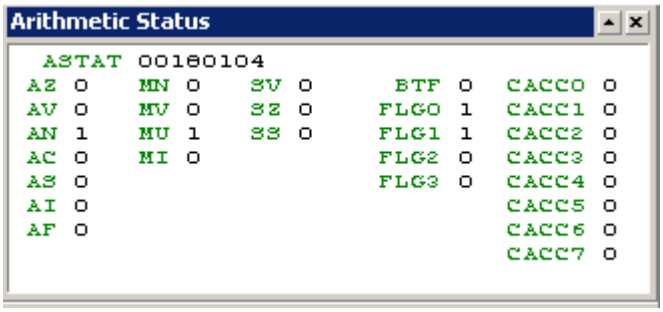
R7 = 16 (в двоичном виде 10000) = 8 (в двоичном виде 1000)<<1;

- окончанием работы программы должна быть команда **IDLE**.

Чтобы проверить правильность программы, необходимо запустить программу на исполнение нажатием клавиши **F5**. Затем в меню **Debug** выбрать пункт **Halt** (Shift+F5) и кликнуть по нему курсором мыши. В результате выполнение программы приостановится на команде, которую она выполняла в момент выбора пункта меню. Поскольку наша первая программа чрезвычайно мала, то к моменту выполнения команды **Halt** она будет полностью выполнена и должна остановиться на команде **IDLE:**

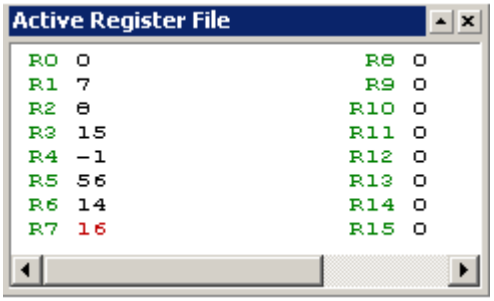
```
r3=r1+r2;
r4=r1-r2;
r5= r1*r2;
r6= ASHIFT R1 BY 0x1;
r7= ASHIFT R2 BY 0x1;
idle;
.endseg;
```

Если первая проверка пройдена успешно, то необходимо посмотреть содержимое регистра **ASTAT**. Для этого необходимо активировать пункт меню **Register->Core->Status->ASTAT**. В результате на экране ЭВМ появится окно **Arithmetic Status**:



Если биты **FLG0** и **FLG1** равны единице, то программа корректно сигнализировала о том, что она загружена и готова к работе.

Подтверждением правильности считывания данных из внутренней памяти и правильности вычислительных операций служит содержимое окна Active Register после окончания работы программы.



2.4. Итоги раздела

После прочтения раздела и повторения действий, приведенных в пункте 2.2, на ЭВМ мы овладели методикой быстрого создания универсального «скелета» проекта для любого процессора семейства **SHARC**.

Написанная на языке ассемблер программа позволила применить теоретические знания о пакете VisualDSP на практике. Основным выводом раздела можно считать утверждение, что написание программы для процессора невозможно без полного знания структуры процессора, регистров управления, методов управления регистрами и памятью. В следующих разделах на примерах различных программ мы ближе познакомимся с возможностями процессора. Рассмотрим различные виды вычислительных операций и команд для процессора ADSP-21062.

Следует обратить внимание на то, чтобы в названиях проекта, файлов проекта и полного пути до него использовались исключительно буквы латинского алфавита. В противном случае могут возникать ошибки и работоспособность проекта будет невозможна.

3. Описание процессоров семейства SHARC

3.1. Общие сведения

Процессор ADSP-2106x – высокопроизводительный 32-разрядный цифровой сигнальный процессор – является представителем семейства ADSP-21000. Он может использоваться для обработки речи, звука, графики и др. Для формирования полноценной системы на кристалл добавлены двухпортовое статическое оперативное запоминающее устройство (SRAM) и интегрированные периферийные устройства ввода/вывода (Input/Output – I/O), которые работают со специализированной шиной I/O. Процессор может выполнять почти все команды за один цикл, используя расположенный на кристалле кэш команд. Супергарвардская архитектура процессора ADSP-2106x включает четыре независимых шины для операций с данными, командами и для ввода/вывода, а также коммутатор шин. В процессоре ADSP-2106x реализуется новый уровень интеграции для цифровых сигнальных процессоров, который заключается в комбинации высокопроизводительного ядра цифрового сигнального процессора с плавающей точкой с интегрированными на кристалл устройствами: интерфейсом хост-процессора (хост-интерфейс), контроллером прямого доступа в память (DMA), последовательными портами, линк портами. Кроме того, в процессоре предусмотрена возможность создания многопроцессорных сетей с применением прямого соединения шин без дополнительного оборудования. На рис. 3.1 показана супергарвардская архитектура ADSP-2106x: коммутатор шин, соединяющий ядро числового процессора с независимым устройством ввода/вывода, двухпортовой памятью и параллельным портом системной шины.

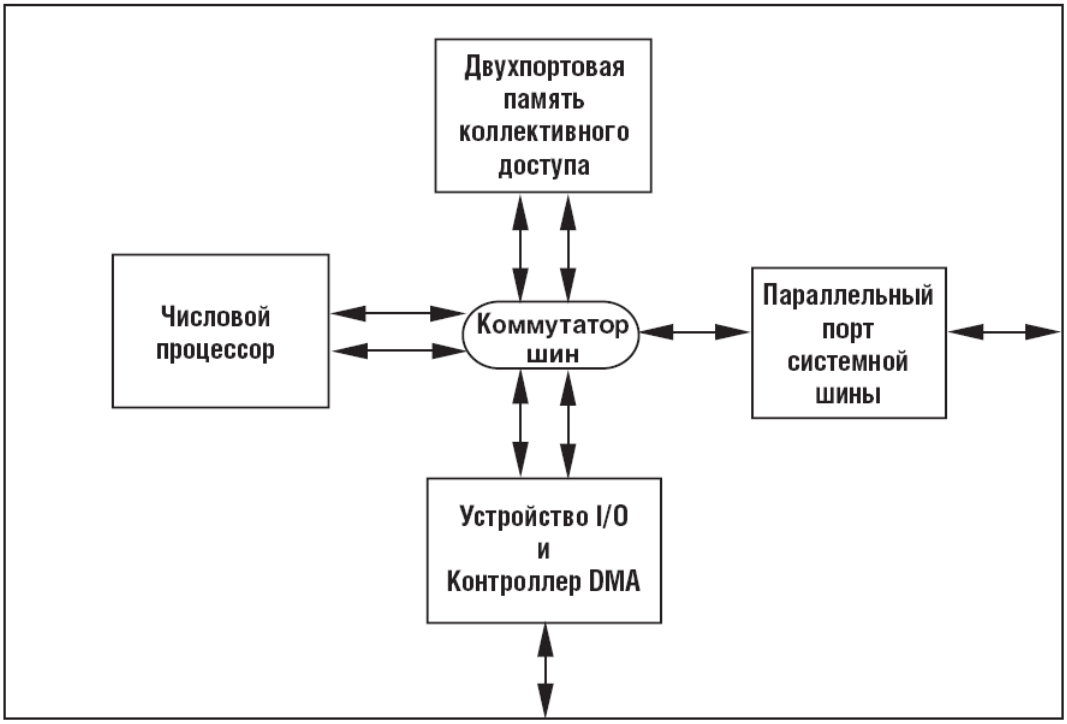


Рис. 3.1. Супергарвардская архитектура процессора ADSP-2106x

На рис. 3.2 показана детальная блок-схема процессора, где отображены следующие устройства:

- 32-разрядные вычислительные устройства с плавающей точкой стандарта IEEE: умножитель, арифметико-логическое устройство (ALU) и устройство сдвига;
- регистровый файл данных;
- генераторы адреса данных (DAG1, DAG2);
- программный автомат (program sequencer) с кэшем команд;
- таймер;
- двухпортовое статическое оперативное запоминающее устройство (SRAM);
- внешний порт для взаимодействия с внешней памятью и периферийными устройствами;
- интерфейс хост-процессора и многопроцессорный интерфейс;
- контроллер DMA;
- последовательные порты;

- линк-порты;
- JTAG тест-порт.

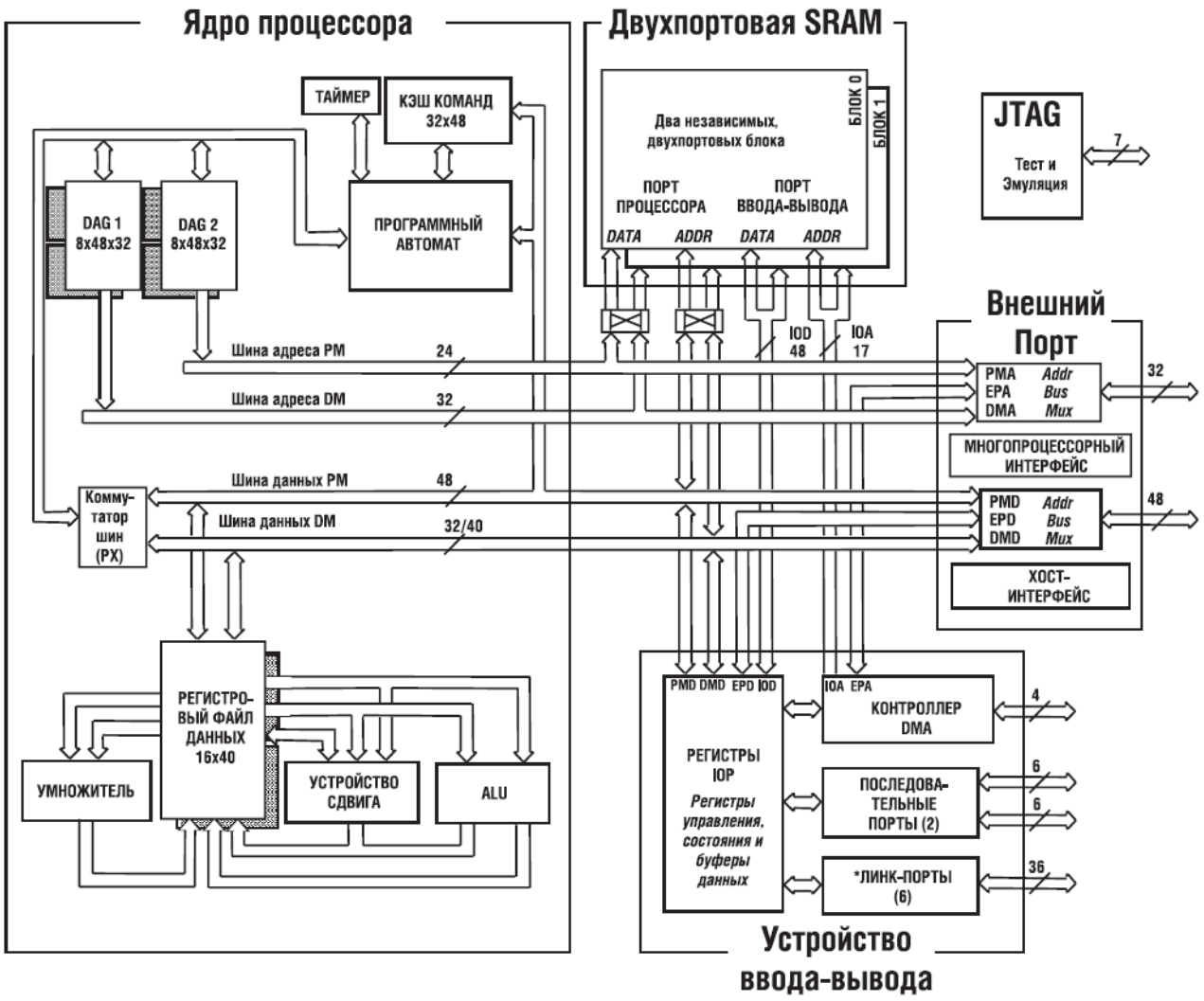


Рис. 3.2. Блок-схема процессора ADSP-2106x

На рис. 3.2 показаны три расположенных на кристалле шины: шина памяти программы (Program Memory – PM), шина памяти данных (Data Memory – DM) и шина ввода/вывода (I/O). Шина PM используется для обращения к любым командам или данным. В течение одного цикла процессор может выбрать два операнда (один по шине PM и один по шине DM), команду (из кэша) и выполнить передачу данных с использованием прямого доступа в память. Внешний порт ADSP-2106x обеспечивает интерфейс процессора с внешней памятью, устройствами I/O, отображенными в карте памяти, хост-процессором и другими ADSP-2106x в

многопроцессорной системе. Внешний порт выполняет арбитраж внутренней и внешней шин, а также обеспечивает сигналы управления для совместно используемой глобальной памяти и устройств ввода/вывода.

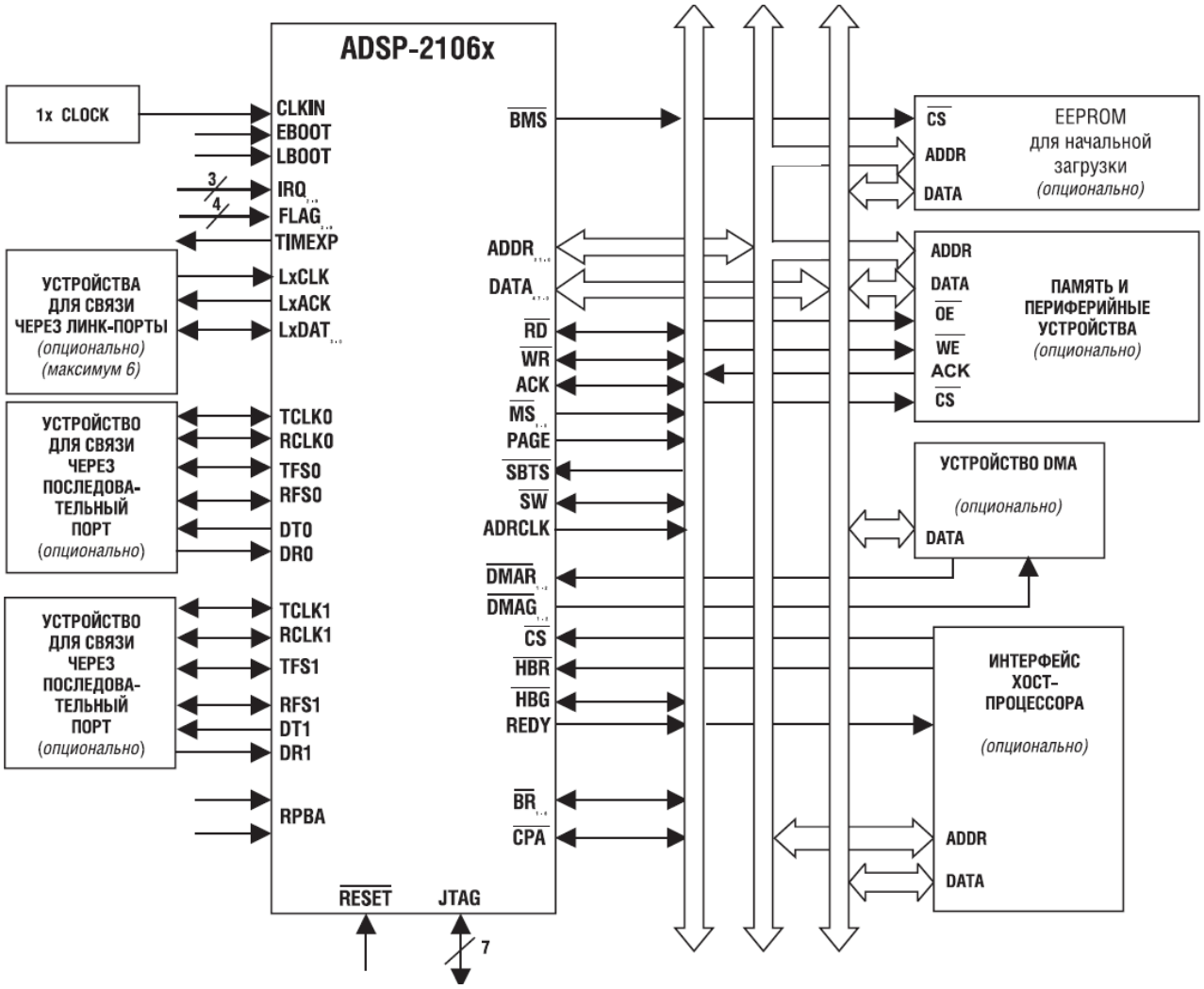


Рис. 3.3. Основная система ADSP-2106x

Процессоры ADSP-21060, ADSP-21062 и ADSP-21061 фактически идентичны. ADSP-21060 имеет 4 Мбита статической оперативной памяти, ADSP-21062 имеет 2 Мбита и ADSP-21061 – 1 Мбит (все три процессора программно и функционально совместимы с процессором ADSP-21020). Помимо этого, существуют четыре отличия между ADSP-21060, ADSP-21062 и ADSP-21061.

В ADSP-21061:

- нет линк-портов;

- 6 каналов DMA – 4 для последовательного и 2 для внешнего портов (вместо 4);
- имеются дополнительные возможности прямого доступа к памяти для последовательного порта;
- введена новая команда idle 16 для снижения потребляемой мощности.

3.2. Характеристики семейства ADSP-21000

Процессоры ADSP-2106x принадлежат к семейству цифровых сигнальных процессоров ADSP-21000 с плавающей точкой. В архитектуре семейства ADSP-21000 получили дальнейшее развитие пять основных требований, которым удовлетворяли 16-разрядные цифровые сигнальные процессоры с фиксированной точкой семейства ADSP-2100:

- быстрая, гибкая арифметика вычислительных устройств;
- непрерывный поток данных в вычислительные устройства и из них;
- повышенная точность и расширенный динамический диапазон в вычислительных устройствах;
- два генератора адреса;
- эффективная работа программного автомата.

Быстрая, гибкая арифметика. Процессоры семейства ADSP-21000 выполняют все команды за один цикл. Они поддерживают высокую тактовую частоту, а также полный набор арифметических операций, включающий, помимо традиционных умножения, сложения, вычитания и комбинированного умножения/сложения, примитивы деления (1/X), операции Min, Max, Clip, Shift и Rotate. Процессоры совместимы со стандартом IEEE представления чисел с плавающей точкой и допускают как

обработку прерывания при арифметическом исключении, так и обработку зафиксированных состояний исключения.

Непрерывный поток данных. ADSP-2106x имеет расширенную гарвардскую архитектуру, объединенную с 10-портовым регистровым файлом данных. В каждом цикле:

- два операнда могут быть считаны или записаны в регистровый файл или из него;
- два операнда могут быть переданы в ALU;
- два операнда могут быть переданы в умножитель;
- два результата могут быть приняты от ALU и умножителя.
- Процессор может параллельно выполнять передачу данных и арифметические операции в одной команде.

40 разрядная повышенная точность. Процессоры семейства ADSP-21000 обрабатывают данные в 32-разрядном формате с плавающей точкой, 32-разрядные целые и дробные форматы (в дополнительном коде и беззнаковые) и данные в 40-разрядном формате повышенной точности стандарта IEEE с плавающей точкой. Процессор перемещает данные повышенной точности между своими вычислительными устройствами, ограничивая ошибку усечения промежуточных данных. При обработке данных 32-разрядная мантисса повышенной точности может передаваться во все вычислительные устройства и из них. При желании 40-разрядная шина данных может быть выведена за кристалл. В процессоре есть 80-разрядный аккумулятор для выполнения вычислений с 32-разрядными данными с фиксированной точкой.

Два генератора адреса. Процессоры семейства ADSP-21000 имеют два генератора адреса данных (DAG), которые обеспечивают прямую или косвенную адресацию (пред- и постмодификацию адреса). Операции

адресации по модулю и битреверсные операции поддерживаются без ограничений на размещение буфера данных.

Эффективная работа программного автомата. Кроме циклов с нулевыми непроизводительными затратами, процессоры семейства ADSP-21000 поддерживают инициализацию циклических операций (циклов) и выход из них за один такт. Циклы могут быть и вложенными (шесть аппаратных уровней), и прерываемыми. Процессор поддерживает задержанный и незадержанный переходы.

3.2.1. Дополнительные характеристики системы

Процессоры семейства ADSP-21000 имеют несколько особенностей, упрощающих разработку системы. Дополнения внесены в три ключевые области:

- архитектурные особенности, поддерживающие языки высокого уровня и операционные системы;
- последовательное сканирование и эмуляция на кристалле с помощью порта 1149.1 JTAG;
- поддержка форматов с плавающей точкой стандарта IEEE.

Языки высокого уровня. Архитектура семейства ADSP-21000 имеет несколько характерных особенностей, которые позволяют поддерживать компиляторы языков высокого уровня и операционные системы:

- регистровые файлы данных и адреса общего назначения;
- собственные 32-разрядные типы данных;
- большое адресное пространство;
- адресация с пред- и постмодификацией;
- размещение циклического буфера данных без ограничений на его расположение в памяти;

- стек счетчика команд, стеки цикла и стек состояния, расположенные на кристалле.

Кроме того, процессоры семейства ADSP-21000 поддерживают расширенный стандарт ANSI языка Си, векторные типы данных и операторы для числовой и сигнальной обработки.

Возможности последовательного сканирования и эмуляции. Процессоры семейства ADSP-21000 поддерживают стандарт IEEE P1149.1 JTAG (Joint Test Action Group) для отладки системы. Этот стандарт определяет метод последовательного сканирования состояний I/O каждого компонента системы.

Форматы данных стандарта IEEE. Процессоры семейства ADSP-21000 поддерживают форматы данных с плавающей точкой стандарта IEEE. Это означает, что приложения, разработанные для IEEE-совместимых процессоров, можно легко использовать в процессорах семейства ADSP-21000.

3.2.2. Почему процессор с плавающей точкой?

Формат данных цифрового сигнального процессора определяет качество обработки сигналов с различными точностью, динамическим диапазоном и отношением сигнал/шум. Однако также важно рассматривать условия рынка и критерий простоты использования.

Точность. Число разрядов аналого-цифровых преобразователей продолжает увеличиваться, следовательно, возникают требования дальнейшего увеличения точности и частоты дискретизации.

Динамический диапазон. Алгоритмы компрессии и декомпрессии данных традиционно используются для обработки сигналов с заданным диапазоном. При разработке алгоритмов должны учитываться критерии низкой стоимости и простоты реализации. Адаптивная фильтрация и воспроизведение изображения являются двумя приложениями, для которых требуется широкий динамический диапазон.

Отношение сигнал/шум. В радиолокации, гидролокации и других коммерческих приложениях, таких как распознавание речи, требуется широкий динамический диапазон, чтобы различать сигналы на фоне шумов.

Критерий простоты использования. 32-разрядный процессор с плавающей точкой более прост в использовании и позволяет быстрее найти коммерческое применение, чем 16-разрядный процессор с фиксированной точкой. Причина этого заключается в архитектуре процессора с плавающей точкой. Согласованность с моделированием на рабочих станциях стандарта IEEE и устранение необходимости масштабирования – два явных преимущества простоты использования. Когда реализуется алгоритм обработки сигнала, потраченное на разработку время при использовании языков высокого уровня при большом адресном пространстве и широком динамическом диапазоне будет меньше времени, требуемого при кодировании на ассемблере.

3.3. Архитектура ADSP-2106x

3.3.1. Ядро процессора

Ядро процессора ADSP-2106x состоит из трех вычислительных устройств, программного автомата, двух генераторов адреса данных, таймера, кэша команд и регистрового файла данных.

Вычислительные устройства. Процессор ADSP-2106x содержит три независимых вычислительных устройства: арифметико-логическое устройство (ALU), умножитель с аккумулятором с фиксированной точкой и устройство сдвига. Вычислительные устройства обрабатывают данные в трех форматах: 32-разрядном с фиксированной точкой, 32-разрядном и 40-разрядном с плавающей точкой. Операции с плавающей точкой – одиночной точности стандарта IEEE. 32-разрядный формат с плавающей точкой соответствует стандарту IEEE, а 40-разрядный формат стандарта IEEE повышенной точности имеет восемь дополнительных младших разрядов мантиссы.

ALU выполняет стандартный набор арифметических и логических операций в форматах и с плавающей, и с фиксированной точкой.

Умножитель выполняет умножение с фиксированной и с плавающей точкой, а также операции умножение/сложение и умножение/вычитание с фиксированной точкой.

Устройство сдвига выполняет логические и арифметические сдвиги, операции с битами, внесение поля битов и его извлечение, а также операцию нахождения порядка 32-разрядных операндов. Вычислительные устройства выполняют операции за один цикл; нет вычислительного конвейера. Они соединяются между собой параллельно. Выход любого устройства может быть входом любого другого в следующем цикле. При многофункциональных вычислениях ALU и умножитель выполняют операции независимо и одновременно.

Регистровый файл данных. Файл универсальных регистров данных используется для передачи данных между вычислительными устройствами и шинами данных, а также для хранения промежуточных результатов. Для быстрого контекстного переключения регистровый файл имеет два набора регистров (первичный и дополнительный) по шестнадцать регистров каждый. Все регистры 40-разрядные. Регистровый файл в процессоре

обеспечивает непрерывный поток данных между вычислительными устройствами и внутренней памятью.

Программный автомат и генераторы адреса данных. Два специальных генератора адреса и программный автомат обеспечивают адресацию при обращении программы к памяти. Программный автомат и генераторы адреса данных отвечают за выполнение вычислительных операций с максимальной эффективностью, поскольку сами вычислительные устройства занимаются исключительно обработкой данных. При помощи кэша команд процессор ADSP-2106x может одновременно выбирать команду (из кэша) и два операнда данных (из памяти). Генераторы адреса данных позволяют организовывать циклические буферы данных.

Программный автомат выполняет адресацию команд в памяти программы. Он управляет итерациями цикла и поддерживает условные команды. Благодаря внутреннему счетчику цикла и стеку цикла процессор выполняет программу цикла с нулевыми потерями. Не требуется никаких явных команд перехода для организации циклов или декремента и проверки счетчика.

Высокая скорость выполнения команд в процессоре достигается посредством последовательности циклов выборки, декодирования и выполнения. Если бы не имелось цикла декодирования, то при использовании внешней памяти для завершения обращения потребовалось бы больше времени.

Генераторы адреса данных (DAG) обеспечивают адресацию при передаче данных между памятью и регистрами. Два генератора адреса данных позволяют процессору одновременно выводить адреса для двух операндов при выполнении операций считывания или записи. DAG1 генерирует 32-разрядные адреса памяти данных. DAG2 генерирует 24-разрядные адреса памяти программы. Каждый генератор содержит восемь регистров адреса, восемь регистров модификации и восемь регистров длины.

Указатель, используемый для косвенной адресации, может модифицироваться значением определенного регистра перед обращением к памяти (предмодификация) или после обращения к памяти (постмодификация). Регистр длины используется для выполнения автоматической адресации по модулю для циклического буфера данных, причем циклический буфер может размещаться в произвольных границах в памяти. Каждый регистр DAG имеет дополнительный регистр, который может быть активизирован для быстрого контекстного переключения.

Циклические буферы позволяют эффективно реализовывать линии задержки и другие структуры, которые требуются в цифровой обработке сигналов, и обычно используются в цифровых фильтрах и преобразованиях Фурье. Генераторы автоматически обрабатывают циклический переход указателя адреса, уменьшая затраты, увеличивая производительность и упрощая реализацию.

Кэш команд. Программный автомат имеет кэш команд размером 32 слова, который обеспечивает работу с тремя шинами для выборки команды и двух значений данных. Кэш избирателен: кэшируются только те команды, при выборке которых возникает конфликт с обращением к данным памяти программы. Это обуславливает высокое быстродействие при выполнении основных циклических операций типа умножения-накопления в цифровом фильтре и «бабочке» БПФ.

Прерывания. ADSP-2106x имеет четыре внешних аппаратных прерывания: три универсальных прерывания IRQ 2−0 и специальное прерывание для сброса. Процессор также имеет внутренне генерируемые прерывания по таймеру, прерывания операций контроллера DMA, переполнения циклического буфера, переполнения стека, арифметических исключений, многопроцессорное векторное прерывание и определяемые пользователем программные прерывания.

При генерации универсальных внешних прерываний и внутреннего прерывания по таймеру ADSP-2106x автоматически помещает в стек содержимое регистров арифметического состояния (ASTAT) и режима (MODE1) и параллельно обрабатывает эти прерывания. Допускается четыре уровня вложенности прерываний.

Таймер. Программируемый таймер обеспечивает периодическую генерацию прерываний. Когда таймер активизирован, значение 32-разрядного регистра счетчика уменьшается в каждом цикле. Когда значение регистра счетчика достигает нуля, генерируется прерывание и выставляется выходной сигнал TIMEXP. Затем регистр счетчика автоматически перезагружается из 32-разрядного регистра периода, после чего продолжается счет.

Шины ядра процессора. Ядро процессора имеет четыре шины: адреса памяти программы, адреса памяти данных, данных памяти программы и данных памяти данных. В процессорах ADSP-2106x в памяти данных хранятся операнды данных, в памяти программы – и команды, и данные (например, коэффициенты фильтра). Это позволяет осуществлять двойную выборку данных, когда команда находится в кэше.

Шина адреса памяти программы (PM) и шина адреса памяти данных (DM) используются для передачи адресов команд и данных. Шина данных PM и шина данных DM используются для передачи данных или команд, хранящихся в памяти любого типа. 24-разрядная шина адреса PM обеспечивает адресацию до 16 мегаслов смешанных команд и данных. По 48-разрядной шине данных PM передаются 48-разрядные команды. Данные с фиксированной и с плавающей точкой с одинарной точностью располагаются в 32 старших битах шины данных PM.

32-разрядная шина адреса DM обеспечивает прямую адресацию до 4-х гигаслов данных. Шина данных DM 40-разрядная. Данные с фиксированной точкой и с плавающей точкой одиночной точности размещаются в 32 старших битах шины данных DM. По шине данных DM за

один цикл передается содержимое любого регистра процессора, которое перемещается в любой другой регистр или область памяти данных. Адрес памяти данных определяется одним из двух источников: абсолютным значением, определенным в коде команды (прямая адресация) или регистром генератора адреса данных (косвенная адресация).

Внутренняя передача данных. Почти каждый регистр в ядре процессора ADSP-2106x классифицируется как универсальный регистр. Существуют определенные команды для передачи данных между любыми двумя универсальными регистрами или между универсальным регистром и памятью. Это относится и к регистрам управления, и к регистрам состояния, а также к регистрам данных в регистровом файле. Регистры устройства обмена данными между шинами (PX) обеспечивают перемещение данных между 48-разрядной шиной данных PM и 40-разрядной шиной данных DM или между 40-разрядным регистровым файлом и шиной данных PM. Эти регистры содержат аппаратные средства для устранения различия в разрядности этих шин.

Контекстное переключение. Большинство регистров процессора имеет дополнительные регистры, которые могут активизироваться во время обработки прерывания, чтобы обеспечить быстрое контекстное переключение. Регистры данных в регистровом файле, регистры DAG, регистр результата умножителя имеют дополнительные регистры. Регистры, активные после сброса, называются первичными регистрами, а другие называются дополнительными (или вторичными) регистрами. Биты управления в регистре управления режимом определяют, какой набор регистров является активным в определенный момент времени.

Набор команд. Набор команд процессоров семейства ADSP-21000 обеспечивает широкие возможности для их программирования. *Многофункциональные* команды позволяют выполнять вычисления параллельно с передачей данных, а также одновременные операции умножителя и ALU. Практически каждая команда может быть выполнена за

один процессорный цикл. Алгебраический синтаксис, используемый ассемблером семейства ADSP-2106x, упрощает кодирование и читаемость. Удобный набор средств разработки облегчает программирование.

3.3.2. Двухпортовая внутренняя память

Процессор ADSP-21060 содержит 4 Мбита статической оперативной памяти, организованной как два блока по 2 Мбита, которые могут конфигурироваться для различных комбинаций хранения кода и данных. ADSP-21062 содержит 2 Мбита памяти, 2 блока по 1 Мбиту. В одном цикле к каждому блоку памяти могут независимо обращаться ядро процессора и устройство ввода/вывода или контроллер DMA. Использование двухпортовой памяти и отдельных шин позволяет выполнить за один цикл две передачи данных из ядра и одну из устройства ввода/вывода.

Обращение к памяти может выполняться к 16-разрядным, 32-разрядным или 48-разрядным словам. В ADSP-21060 память может содержать максимум 128 килослов 32-разрядных данных, 256 килослов 16-разрядных данных, 80 килослов 48-разрядных команд (и 40-разрядных данных) или комбинацию слов различной разрядности объемом до 4 Мбит. В ADSP-21062 память может содержать максимум 64 килослова 32-разрядных данных, 128 килослов 16-разрядных данных, 40 килослов 48-разрядных команд (и 40-разрядных данных) или комбинацию слов различной разрядности объемом до 2 Мбит. В ADSP-21061 память может содержать максимум 32 килослова 32-разрядных данных, 64 килослова 16-разрядных данных, 16 килослов 48-разрядных команд (и 40-разрядных данных) или комбинацию слов различной разрядности, объемом до 1 Мбита.

Поддерживается формат хранения 16-разрядных данных с плавающей точкой, что удваивает количество данных, которые могут храниться на кристалле. Преобразование между 32-разрядным форматом с плавающей

точкой и 16-разрядным форматом с плавающей точкой выполняется с помощью одной команды.

Хотя каждый блок памяти может хранить комбинации кода и данных, обращение к памяти выполняется наиболее эффективно, когда в одном блоке хранятся данные и для их передачи используется шина DM, а в другом блоке – команды и данные, а для их передачи используется шина PM. Такое использование шины DM и шины PM, где для каждой выделен блок памяти – гарантирует выполнение за один цикл двух передач данных. В этом случае команда должна быть доступна из кэша. Передача операнда данных через внешний порт также происходит за один цикл.

3.3.3. Интерфейс внешней памяти и периферийных устройств

Внешний порт процессора ADSP-2106x обеспечивает интерфейс с внешней памятью и периферийными устройствами. 4 гигаслова внешнего адресного пространства включаются в объединенное адресное пространство ADSP-2106x. Раздельные шины на кристалле – адреса PM, данных PM, адреса DM, данных DM, адреса I/O и данных I/O – объединяются во внешнем порте и образуют внешнюю системную шину с одной 32-разрядной шиной адреса и одной 48-разрядной шиной данных. Внешняя SRAM может быть как 16-, 32-, так и 48-разрядной; расположенный на кристалле контроллер DMA автоматически упаковывает внешние данные в слова соответствующей разрядности: либо 48- разрядные команды, либо 32-разрядные данные.

Адресация внешней памяти облегчается за счет декодирования старших адресных линий для генерирования сигналов выбора банка памяти. Для упрощения адресации динамической оперативной памяти (DRAM) со страничной организацией генерируются отдельные сигналы управления. Процессор ADSP-2106x использует программируемые состояния ожидания при обращении к внешней памяти и сигналы квитирования для внешней

памяти, чтобы обеспечить интерфейс с DRAM и периферийными устройствами с различным быстродействием.

3.3.4. Интерфейс хост-процессора

Интерфейс хост-процессора ADSP-2106x обеспечивает простое подклю-чение к стандартным микропроцессорным шинам – 16- и 32-разрядным. Асинхронная передача данных может осуществляться на полной тактовой частоте процессора. Интерфейс хост-процессора доступен через внешний порт процессора и отображается в карте памяти объединенного адресного пространства. Для хост-интерфейса доступны четыре канала DMA; передача кода и данных выполняется с низкими затратами ресурсов процессора. Хост-процессор может непосредственно считывать и записывать содержимое внутренней памяти ADSP-2106x, имеет доступ к настройке каналов DMA и почтовым регистрам. Поддержка векторных прерываний обеспечивает эффективное выполнение команд хост-процессора.

3.3.5. Многопроцессорная обработка

Процессоры ADSP-2106x позволяют создавать многопроцессорные системы. Объединенное адресное пространство допускает прямые межпроцессорные обращения к внутренней памяти каждого ADSP-2106x. Расположенная на кристалле логическая схема арбитража совместно используемой шины применяется для простого соединения до шести процессоров ADSP-2106x и хост-процессора в систему. Смена ведущего процессора в системе занимает один непроизводительный цикл. Схема арбитража шины может выбираться: приоритеты могут быть как постоянными, так и вращающимися. Блокировка шины процессором обеспечивает выполнение неделимой последовательности *чтение-модификация-запись* для семафоров. Для межпроцессорных команд

предусмотрено определенное векторное прерывание. Максимальная производительность передачи данных между процессорами через линк-порты или внешний порт составляет 240 Мбайт/с. *Широковещательная передача* – это одновременная передача данных во все ADSP-2106x, которая используется для реализации взаимных семафоров.

3.3.6. Устройство ввода/вывода (IOP)

Устройство ввода/вывода (IOP) процессора включает два последовательных порта, шесть 4-разрядных линк-портов и контроллер DMA.

Последовательные порты. Процессор ADSP-2106x имеет два синхронных последовательных порта, которые обеспечивают удобный интерфейс с множеством цифровых и периферийных аналого-цифровых устройств. Через последовательные порты могут передаваться данные на полной тактовой частоте процессора, при которой обеспечивается максимальная скорость передачи 40 Мбит/с. Независимость функций передачи и приема обеспечивает большую гибкость при организации связи через последовательные порты. Данные последовательного порта могут автоматически передаваться в расположенную на кристалле память и из нее с использованием DMA. Каждый из последовательных портов поддерживает многоканальный режим с временным разделением каналов (TDM).

Последовательные порты могут работать с различными форматами передачи данных: передача начинается со старших бит или с младших, у которых длина слова от 3 до 32 бит. Сигналы тактовой и кадровой синхронизации последовательного порта могут быть внутренними или внешними.

Линк-порты. Процессоры ADSP-21062 и ADSP-21060 имеют шесть 4-разрядных линк-портов, которые позволяют реализовывать ввод-вывод данных. Линк-порты могут синхронизироваться дважды за цикл, что

позволяет каждому из них передавать за один цикл 8 бит. Линк-порты особенно полезны в многопроцессорных системах для реализации связи между процессорами по схеме «точка-к-точке».

Линк-порты могут функционировать независимо и одновременно с максимальной производительностью 240 Мбайт/с. Данные линк-порта упаковываются в 32- и 48-разрядные слова и могут прямо считываться ядром процессора или передаваться по DMA во внутреннюю память. Каждый линк-порт имеет собственные двухбуферные входные и выходные регистры. Сигналы квитирования (тактовый/подтверждение связи) управляют передачей данных через линк-порт. Пересылки данных могут программироваться на передачу и на прием. Процессор ADSP-21061 не имеет линк-портов.

Контроллер DMA. Расположенный на кристалле контроллер DMA ADSP-2106x позволяет выполнять передачу данных с нулевыми потерями без вмешательства процессора. Контроллер DMA функционирует независимо по отношению к ядру процессора и не влияет на его работу. Это позволяет выполнять операции прямого доступа в память одновременно с выполнением программы ядром процессора. И код, и данные могут быть загружены в ADSP-2106x с использованием передач по DMA.

Передачи по DMA могут происходить между внутренней памятью ADSP-2106x и внешней памятью, внешними периферийными устройствами или хост-процессором. Передачи по DMA могут также осуществляться между внутренней памятью ADSP-2106x и его последовательными портами или линк-портами. Еще один вариант – передачи по DMA между внешней памятью и внешними периферийными устройствами. В течение передач по DMA автоматически выполняется упаковка данных внешней шины в 16-, 32- и 48-разрядные слова.

В ADSP-21060 и ADSP-21062 доступны десять каналов DMA – два через линк-порты, четыре через последовательные порты и четыре через

внешний порт процессора (для любых передач хост-процессора, других ADSP-2106x). Четыре дополнительных канала DMA линк-порта используются совместно последовательным портом 1 и внешним портом. В ADSP-21061 доступны шесть каналов DMA – четыре через последовательные порты и два через внешний порт. Асинхронные периферийные устройства могут управлять двумя каналами DMA, используя линии DMA запрос/предоставление (DMAR1-2, DMAG1-2). К другим возможностям относятся генерирование прерывания после завершения передачи по DMA и выполнение цепочки операций DMA.

Десять каналов DMA ADSP-21060 и ADSP-21062 описаны ниже:

# канал DMA	*Буфер данных*	*Описание*
0-й канал DMA	RX0	Последовательный порт 0 приема
1-й канал DMA	RX1(*или LBUF0*)	Последовательный порт 1 приема (*или буфер 0 линк-портов)*
2-й канал DMA	TX0	Последовательный порт 0 передачи
3-й канал DMA	TX1(*или LBUF1*)	Последовательный порт 1 передачи (*или буфер 1 линк-портов)*
4-й канал DMA	LBUF2	Буфер 2 линк-портов
5-й канал DMA	LBUF3	Буфер 3 линк-портов
6-й канал DMA	EPB0(*или LBUF4*)	Буфер 0 FIFO внешнего порта (*или буфер 4 линк-портов)*
7-й канал DMA*	EPB1(*или LBUF5*)	Буфер 1 FIFO внешнего порта (*или буфер 5 линк-портов)*
8-й канал DMA*	EPB2	Буфер 2 FIFO внешнего порта
9-й канал DMA	EPB3	Буфер 3 FIFO внешнего порта

DMAR1 и DMAG1 − сигналы квитирования 7-го канала DMA

DMAR2 и DMAG2 − сигналы квитирования 8-го канала DMA

Начальная загрузка. При включении питания системы внутренняя память загружается из 8-разрядного программируемого запоминающего

устройства EPROM или хост-процессора. Процессоры ADSP-21060 и ADSP-21062 могут также загружаться через один из линк-портов. Выбор источника начальной загрузки управляется сигналами на выводах, EBOOT и LBOOT. Для начальной загрузки могут использоваться и 16-, и 32-разрядные хост-процессоры.

3.4. Средства разработки

Процессор ADSP-2106x сопровождается полным набором программных и аппаратных средств разработки; этот набор включает EZ-LAB® Evaluation Board, EZ-ICE® In-Circuit Emulator и программные средства, которые используются для программирования и отладки приложений, написанных на ассемблере и Си.

EZ-ICE Emulator служит для отладки аппаратно-программного обеспечения.

К программным средствам относится компилятор языка Си стандарта ANSI, который содержит в себе стандарт Numerical C extensions, базирующийся на работе ANSI NCEG committee (Nimerical C Extensions Group). В стандарте Numerical C обеспечиваются новые возможности языка Си, связанные с выбором массивов, операциями векторной математики, комплексными типами данных, циклическими указателями и динамическими массивами. Другие компоненты программных средств – это C Runtime Library с определенными функциями DSP, отладчик, ассемблер, библиотека ассемблера, компоновщик и симулятор.

EZ-ICE Emulator использует тест-порт JTAG IEEE 1149.1 процессора для контроля и управления процессором разрабатываемой платы во время эмуляции. EZ-ICE обеспечивает оптимальный режим эмуляции, проверяет и модифицирует память, регистры и стеки процессора. Использование интерфейса JTAG процессора гарантирует внутрисистемную внутрисхемную эмуляцию, которая не влияет на загрузку и синхронизацию системы. Более

подробную и структурированную информацию можно найти в перечне технических характеристик *ADSP-21000 Family Hardware & Software Development Tools*. Этот перечень можно заказать в представительстве Analog Devices или ее дистрибьютора.

3.5. Многопроцессорная сеть

Многопроцессорная сеть – это система параллельной обработки данных, обладающая высокой производительностью, гибкостью и простотой программирования. Такая организация системы поддерживается специальными устройствами в процессорах ADSP-21060 и ADSP-21062. Многопроцессорные сети применяются в широком спектре приложений, к которым относятся радиолокационные системы освещения воздушной обстановки, виртуальная реальность, техническое моделирование, нейронные сети, решение больших систем линейных уравнений и т. д.

3.6. Дополнительная литература

К дополнительным материалам, рекомендуемым для изучения, относятся:

ADSP-21060/62 SHARC Data Sheet;

ADSP-21061 SHARC Data Sheet;

ADSP-21000 Family Hardware & Software Development Tools Data Sheet;

ADSP-21000 Family Assembler Tools & Simulator Manual;

ADSP-21000 Family C Tools Manual;

ADSP-21000 Family C Runtime Library Manual;

ADSP-21000 Family Applications Handbook, Vol. 1.

По вопросам приобретения этих и других дополнительных материалов можно обращаться в представительства фирмы Analog Devises или к ее официальным дистрибьюторам.

4. Примеры рабочих программ для процессора

4.1. Цель раздела

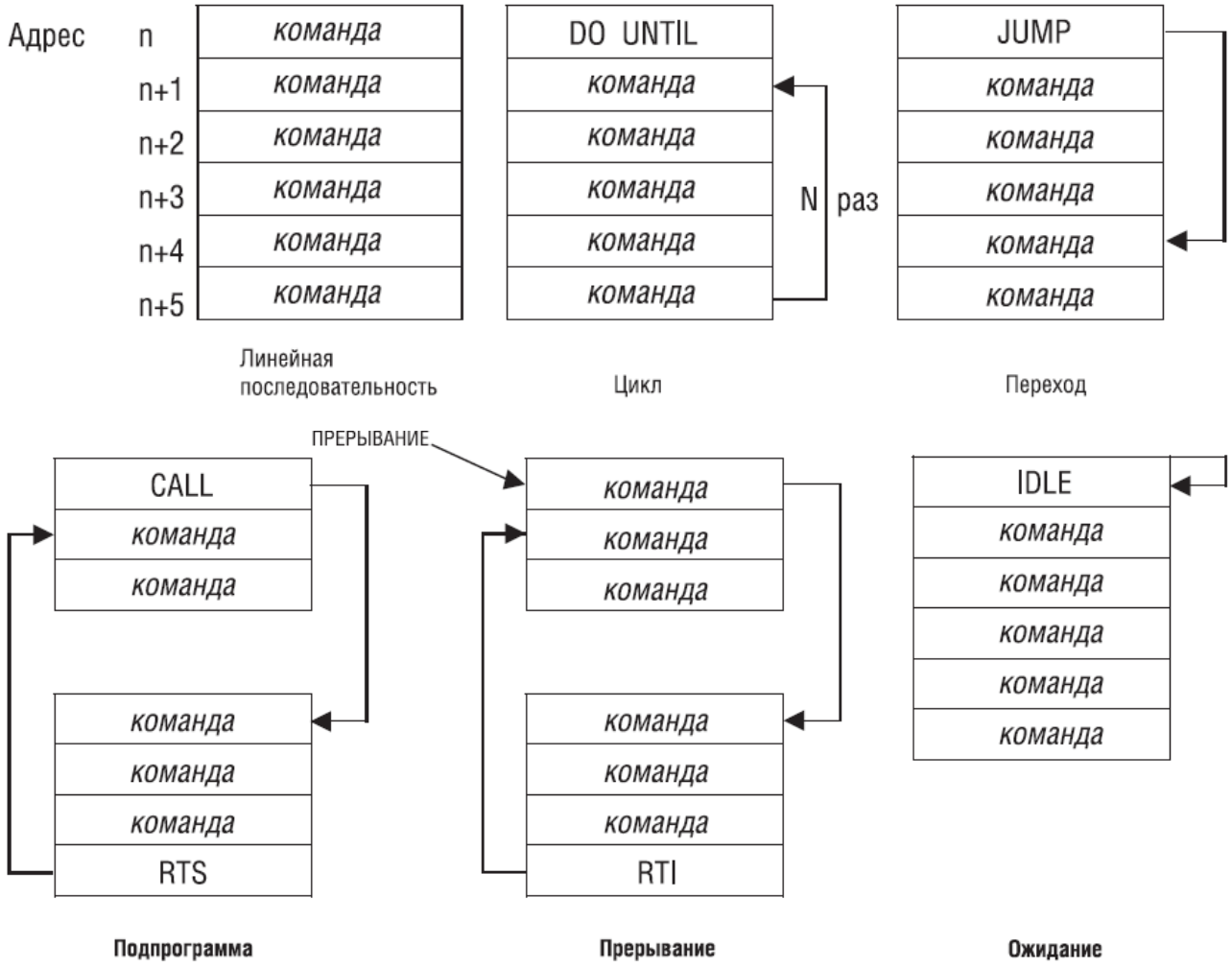
Основной целью раздела можно считать освоение программирования на языке ассемблер для процессора ADSP-21062 на примере трех демонстрацион-ных программ. Кроме того, в процессе знакомства с программами хотелось бы более глубоко раскрыть возможности различных средств отладки и визуализации. К каждой программе будет приложена теоретическая часть, без знания которой реализовать программу не представляется возможным.

4.2. Последовательность выполнения операций для процессоров SHARC

Процесс выполнения программ в ADSP-2106x, как правило, линейный: процессор выполняет команды программы последовательно. Изменения в этом линейном потоке обуславливаются следующими структурами программы:

- *Циклы*. Последовательность команд повторяется некоторое время без непроизводительных затрат.
- *Подпрограммы*. Процессор временно прерывает последовательное выполнение программы для выполнения команды из другой части памяти программы.
- *Переходы*. При выполнении программы происходит безусловный переход к команде в другой части памяти программы.
- *Прерывания*. Частный случай подпрограмм, когда выполнение основной программы прерывается событием (но не командой программы), которое происходит во время ее выполнения.
- *Ожидание*. Специальная команда, по которой процессор прекращает работу и переходит в состояние пониженного потребления

мощности. Когда поступает прерывание, процессор обрабатывает его и продолжает стандартное выполнение программы.



Управление выполнением этих программных структур осуществляет программный автомат. Программный автомат выбирает адрес следующей команды, генерируя большинство этих адресов самостоятельно, а также выполняет следующие функции:

- приращение адреса выборки,
- управление стеками,
- проверка условий,
- декремент счетчика циклов,
- вычисление новых адресов,
- управление кэшем команд,
- обработка прерываний.

Программирование для процессоров на языке ассемблер невозможно без применения циклов, подпрограмм, переходов и прерываний. Перед знакомством с примерами программ необходимо посмотреть Приложение А, чтобы сформировалось представление о данных структурах программы.

4.2. Программа 1. Простейшая программа на ассемблере.

Рассмотренная в разделе программа чрезвычайно проста в реализации и понимании. Реализуя программу, мы познакомимся с операциями:

- чтения из памяти,
- записи в память,
- инициализации переменных,
- написания макросов,
- получения значения адреса переменной,
- получения значения размера массива,
- реализация счетчика с использованием цикла,
- безусловного перехода.

Текс программы приведен ниже.

```
#include "def21062.h" // Подключаем файл, в котором определены все
                      //ключевые слова для процессора ADSP 21062
#define N 10          // Определяем символ N таким образом, что если он будет использован в тексте программы, то компилятор заменит его числом 10
.segment/pm   seg_rth; // Описываем начало сегмента, хранящего вектор прерываний
    // зарезервированное прерывание адреса 0x20000-0x20004
    nop;          // адрес 0x20000 внутренней памяти
    nop;          // адрес 0x20001 внутренней памяти
    nop;          // адрес 0x20002 внутренней памяти
    nop;          // адрес 0x20003 внутренней памяти
    ___lib_RSTI:  // прерывание СБРОС (Reset), с которого начинается работа процессора
    nop;          // адрес 0x20004 внутренней памяти
```

```
jump first;          // адрес 0x20005 внутренней памяти  (команда перехода по метке First)
nop;                 // адрес 0x20006 внутренней памяти
nop;                 // адрес 0x20007 внутренней памяти
.endseg;             // Конец сегмента, хранящего вектор прерываний

.SECTION/DM  seg_dmda; // Описываем начало сегмента, хранящего данные
.VAR sum[N];         // Объявляем массив переменных из N слов
.endseg;             // Конец сегмента, хранящего данные

.segment/PM  seg_pmco;
first:               // метка, по которой осуществляется переход из прерывания Reset
I6= sum;             // в регистр генератора адреса DAG1 записывается адрес переменной  sum
    M6=1;            // значение модификатора M6 принимается равным 1
    M5=0;            // значение модификатора M5 принимается равным 0
    R0=@sumbol;      // в регистр общего назначения записывается значение длинны массива sum в словах
                     // Реализация счетчика с помощью цикла
  LCNTR=length(sumbol), DO outer UNTIL LCE; // начало цикла
    R0=R0+1;         // содержимое регистра R0 инкрементируется на 1.
    DM(I6,M5)=R0;    // содержимое регистра  R0 записывается по адресу памяти I6, после чего адрес увеличивается на значение M5=0
    R1=DM(I6,M6);    // из адреса памяти I6+M5 = I6 считываются данные и помещаются в регистр R1, после чего адрес I6 увеличивается на значение M6, т.е становится равным I6+1, при следующем проходе цикла адрес I6 увеличится еще на 1
    outer:  // метка перед последним исполняемым оператором цикла
    nop;    // последний исполняемый оператор в цикле
    idle;   // окончание программы, режим пониженного энергопотребления
.endseg;             // окончание сегмента программы
```

Листинг программы снабжен большим количеством комментариев, но все же некоторые моменты требуют дополнительного пояснения.

1) **Вопрос**: в каком порядке выполняются команды, описанные в программе?

Ответ: еще до исполнения первой команды процессор размещает все команды и данные по соответствующим сегментам:

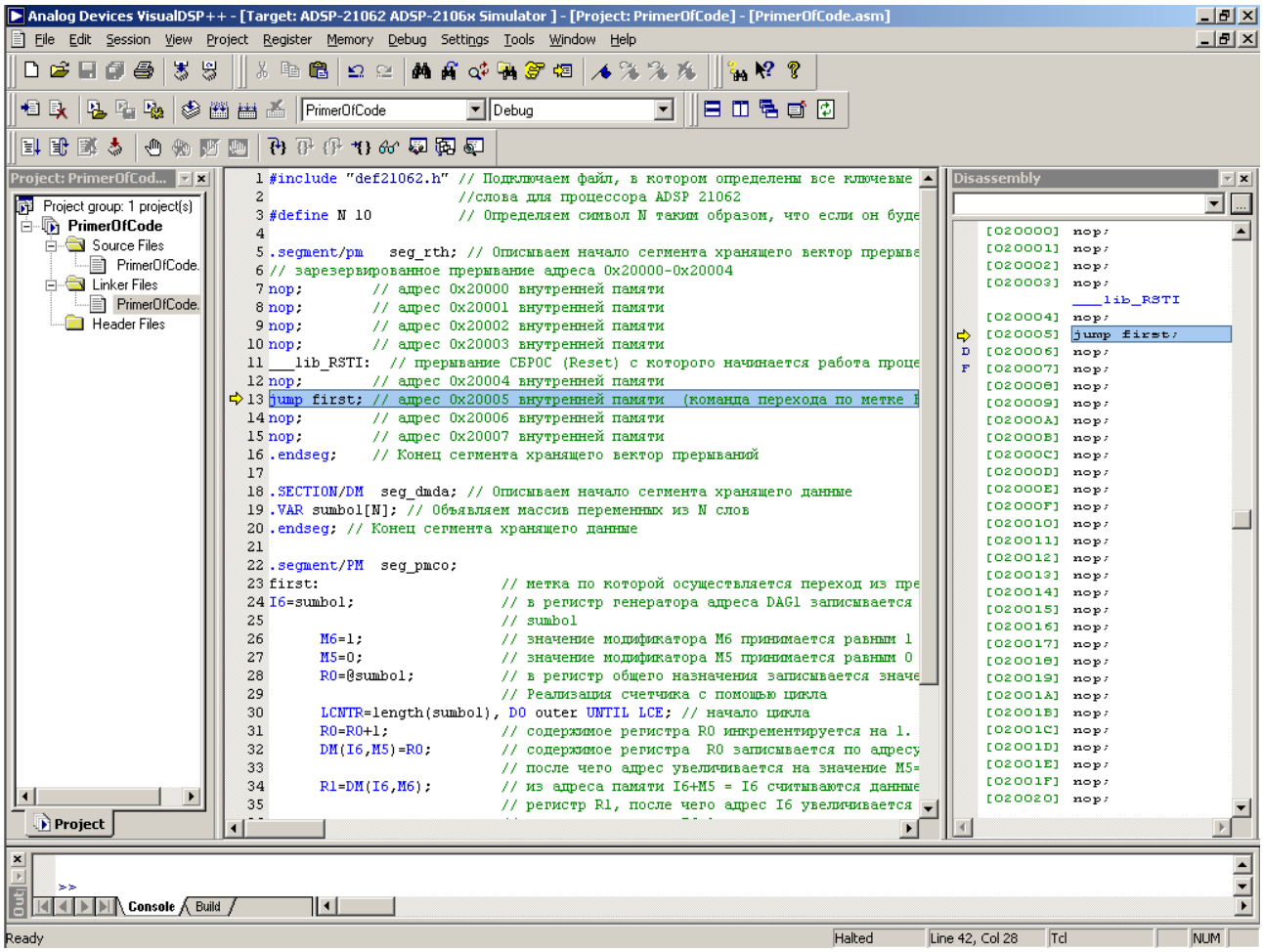
- вектор прерываний в сегмент **seg_rth** по адресам от **0x20000** до **0x20008**,

- переменную **sumbol[N]** в сегменте **seg_dmda** по адресам от **0x28000** до **0x28000+N**,

- команды, описанные в сегменте **seg_pmco**, помещает по адресам от **0x20192** до **0x20192+,** количество команд программы, начиная от метки **first** до команды **idle.** После того как программа загружена в память, процессор начинает последовательно считывать и выполнять команды с адреса **0x20004** в сегменте **seg_rth**. Дойдя до адреса **0x20005**, процессор считывает команду безусловного перехода <**jump first**>. Выполняя команду, он осуществляет переход из сегмента **seg_rth** в сегмент **seg_pmco** по адресу метки **first**. Поскольку никаких других прерываний в данной программе не предусмотрено, то ни одной команды в сегменте **seg_rth** выполнено больше не будет. Метка **first** расположена в самом начале сегмента **seg_pmco (адрес 0x20198),** процессор начинает последовательно выполнять все команды вплоть до адреса **0x201a1,** по которому расположена команда **idle**. На этом работа программы считается завершенной.

2) **Вопрос:** каким образом можно посмотреть адрес команды, описанной в тексте программы?

Ответ: для этого достаточно запустить программу, используя клавишу F7. Затем нажать клавишу F11. В результате окно пакета VisualDSP будет иметь вид:



Желтая стрелка () напротив команды в редакторе кода и напротив ячейки памяти в окне **Disassembly**, будет указывать, по какому адресу расположена команда. Нажимая клавишу **F11**, можно проследить последовательность выполнения операций.

Если программа имеет большой размер и команда, к которой вам необходимо перейти, расположена далеко от начала, используется так называемая точка остановки. Чтобы установить точку остановки, необходимо в редакторе кода поместить курсор на строчку, в которой расположена интересующая команда, и нажать клавишу **F9**. Напротив строчки появится красная точка (). Нажав клавишу **F5,** можно запустить программу. Программа начнет работу, последовательно выполняя команды. Дойдя до точки остановки, приостановит работу, и напротив команды, адрес которой необходимо определить, появится значок  . Данный значок сигнализирует, что данная точка остановки активна. Нажимая клавишу **F5** можно

продолжить выполнение программы до следующей точки. Нажимая клавишу **F11,** можно пошагово отладить программу.

3) **Вопрос:** каким образом организован счетчик?

Ответ: ответ сводится к описанию цикла.

В языке ассемблер для процессоров **SHARC** предусмотрена конструкция, эквивалентная циклу.

LCNTR=N, DO Label UNTIL LCE;

Команды;

Label:

Команда;

Параметр **LCNTR** определяет, сколько итераций должен осуществить цикл. Запись **LCNTR=N** означает, что цикл должен осуществить **N** итераций, где **N** − целое положительное число.

В конструкции **DO Label UNTIL LCE:**

- параметр **Label** − это метка, которая должна быть описана в тексте программы после объявления цикла. Появление метки **Label** в тексте программы сигнализирует о том, что следующая за ней команда является последней командой, содержащейся в теле цикла, а все команды между меткой и строчкой **DO Label UNTIL LCE:** находятся внутри цикла.

- DO, UNTIL, LCE − ключевые слова.

4.3. Программа 2. Работа с прерываниями

Программа должна обрабатывать сигналы от внешнего устройства. На время обработки программа должна временно прекращать текущие операции и выполнять операции, необходимые внешнему устройству. Данный режим реализуется путем создания обработчика прерываний **IRQ1** и **IRQ2**.

В обработчике прерывания **IRQ1** программа процессора должна устанавливать флаг **FLG0,** а в обработчике прерывания **IRQ2** программа процессора должна устанавливать флаг **FLG1.** Считывая состояния флагов,

внешнее устройство сможет определить, произошло прерывание или нет и выполнено ли требуемое от программы **ADSP-21062** действие.

Реализуя программу, мы научимся:

- описывать прерывания,
- сигнализировать, что прерывание произошло,
- имитировать прерывания в пакете **VisualDSP**,
- отслеживать произошедшие прерывания с помощью регистров,
- создавать «бесконечный» цикл.

Важной особенностью данной программы является то, что в ней описывается принцип работы с прерываниями. В рамках программы рассмотрены два прерывания − **IRQ1** и **IRQ2**.

```
#include "def21062.h"

.segment/pm    seg_rth;
nop; nop;nop;nop;
___lib_RSTI:
    nop;
    jump start;
    nop;
    nop;
    nop; nop; nop; nop;
___lib_SOVFI:   RTI;RTI;RTI;RTI;
___lib_TMZHI:  RTI;RTI;RTI;RTI;
___lib_VIRPTI:  RTI;RTI;RTI;RTI;
___lib_IRQ2I: //внешнее прерывание 2
    nop;
    bit set astat ASTAT_FLG1; // установка флага, сигнализирующая о том, что прерывание произошло
    RTI;
    RTI;
___lib_IRQ1I: //внешнее прерывание 1
    nop;
    bit set astat ASTAT_FLG0; // установка флага, сигнализирующая о том, что прерывание произошло
    RTI;
    RTI;
```

```
___lib_IRQ0I:   RTI;RTI;RTI;RTI;
.endseg;

.segment/PM  seg_pmco;
start:
bit set mode2 FLG0O;          //направление флага на передачу
bit set mode2 FLG1O;          //направление флага на передачу
bit set mode2 FLG2O;          //направление флага на передачу

bit clr astat ASTAT_FLG0;
bit clr astat ASTAT_FLG1;
bit clr astat ASTAT_FLG2;

   r1=0x0;
   imask=r1;                  // обнуление регистра IMASK
   irptl=r1;                  // обнуление регистра IRPTL
   imaskp=r1;                 // обнуление регистра IMASKP

bit set mode1 IRPTEN;         //разрешение глобальных прерываний
bit set imask IRQ1I;          //внешнее прерывание
bit set imask IRQ2I;          //внешнее прерывание

do mon until forever;         // бесконечный цикл
  bit set astat ASTAT_FLG2;
mon:
  bit clr astat ASTAT_FLG2;
.endseg;
```

Логика работы программы проста. Первым делом описывается вектор прерываний, в котором активными являются три прерывания **Reset**, **IRQ1**, **IRQ2**.

```
___lib_RSTI:
    nop;
    jump start;
    nop;
    nop;
___lib_IRQ1I:
    nop;
    bit set astat ASTAT_FLG0;
    RTI;
    RTI;
___lib_IRQ2I:
    nop;
    bit set astat ASTAT_FLG1;
    RTI;
```

RTI;

Замечание: Прерывания должны следовать друг за другом в порядке, предусмотренном структурой вектора прерываний.

Каждое из прерываний может содержать не более четырех команд в сегменте **seg_rth**. В прерывании **Reset**, как и в предыдущих программах, описана команда безусловного перехода по метке **Start**. В прерываниях **IRQ1** и **IRQ2** описаны команды установки флагов «**bit set astat ASTAT_FLG(0-1);**» для сигнализации внешним устройствам и команда **RTI**. По команде **RTI** процессор осуществляет выход из прерывания и возврат к той команде, на которой выполнение программы было прервано.

После перехода по метке **Start** программа выполняет команды:

- установки направления трех флагов командой «**bit set mode2 FLG(0-2)O;**» на передачу.

- установки флагов в нуль командой «**bit clr astat ASTAT_FLG(0-2);**», поскольку в начале программа не должна делать флаги активными.

- обнуления регистров **IMASK**, **IRPTL**, **IMASKP**, отвечающих за возможность осуществления прерываний.

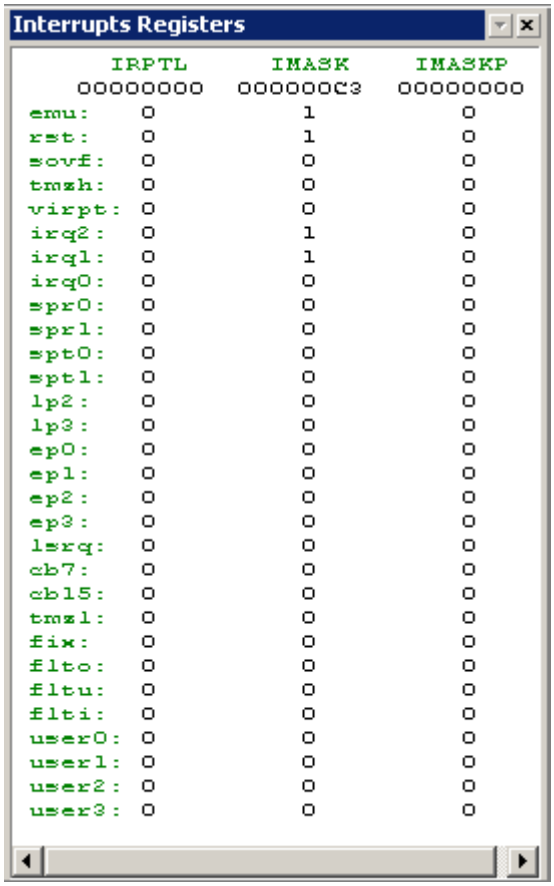
```
r1=0x0;
imask=r1;          // обнуление регистра IMASK
irptl=r1;          // обнуление регистра IRPTL
imaskp=r1;         // обнуление регистра IMASKP
```

Когда все вектора очищены, в процессоре может произойти только одно прерывание **RESET**.

После этого описываются команды, разрешающие прерывания **IRQ1** и **IRQ2**. Для этого выставляются соответствующие биты в регистре **IMASK.** Помимо этого, пишется команда глобального разрешения прерываний.

```
bit set mode1 IRPTEN;      // глобальное разрешение прерываний
bit set imask IRQ1I;       // внешнее прерывание
bit set imask IRQ2I;       // внешнее прерывание
```

Список разрешенных прерываний можно посмотреть, используя пункт меню **Register->Core->Interrupt**. Биты разрешенных прерываний в регистре **IMASK** будут равны единице.



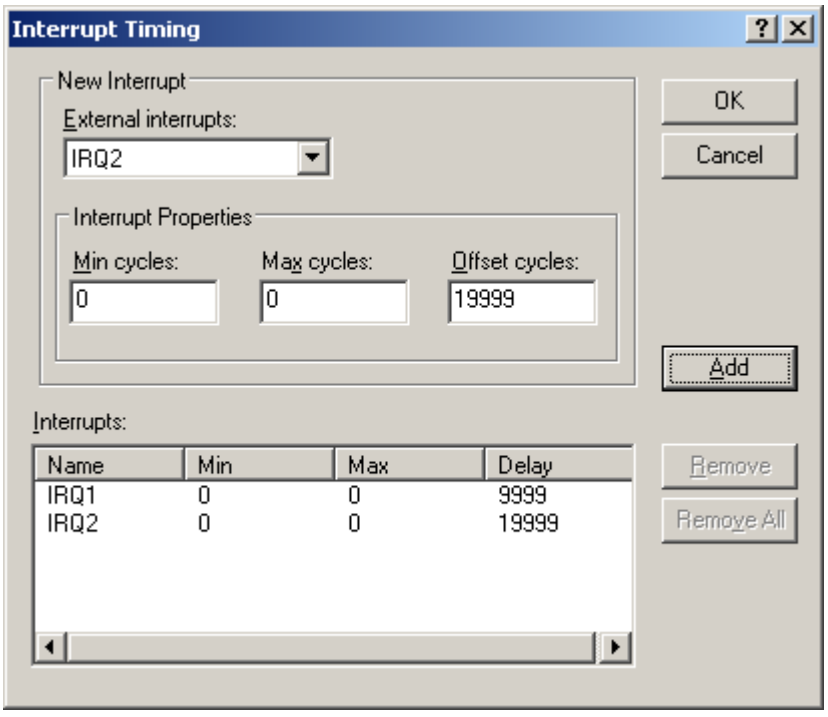
После этого в программе описывается бесконечный цикл, в котором включается и выключается флаг 2.

```
do mon until forever;                // бесконечный цикл
  bit set astat ASTAT_FLG2;
mon:
  bit clr astat ASTAT_FLG2;
```

Этот цикл переводит программу в режим ожидания прерывания и сигнализирует внешнему устройству, что программа готова к работе.

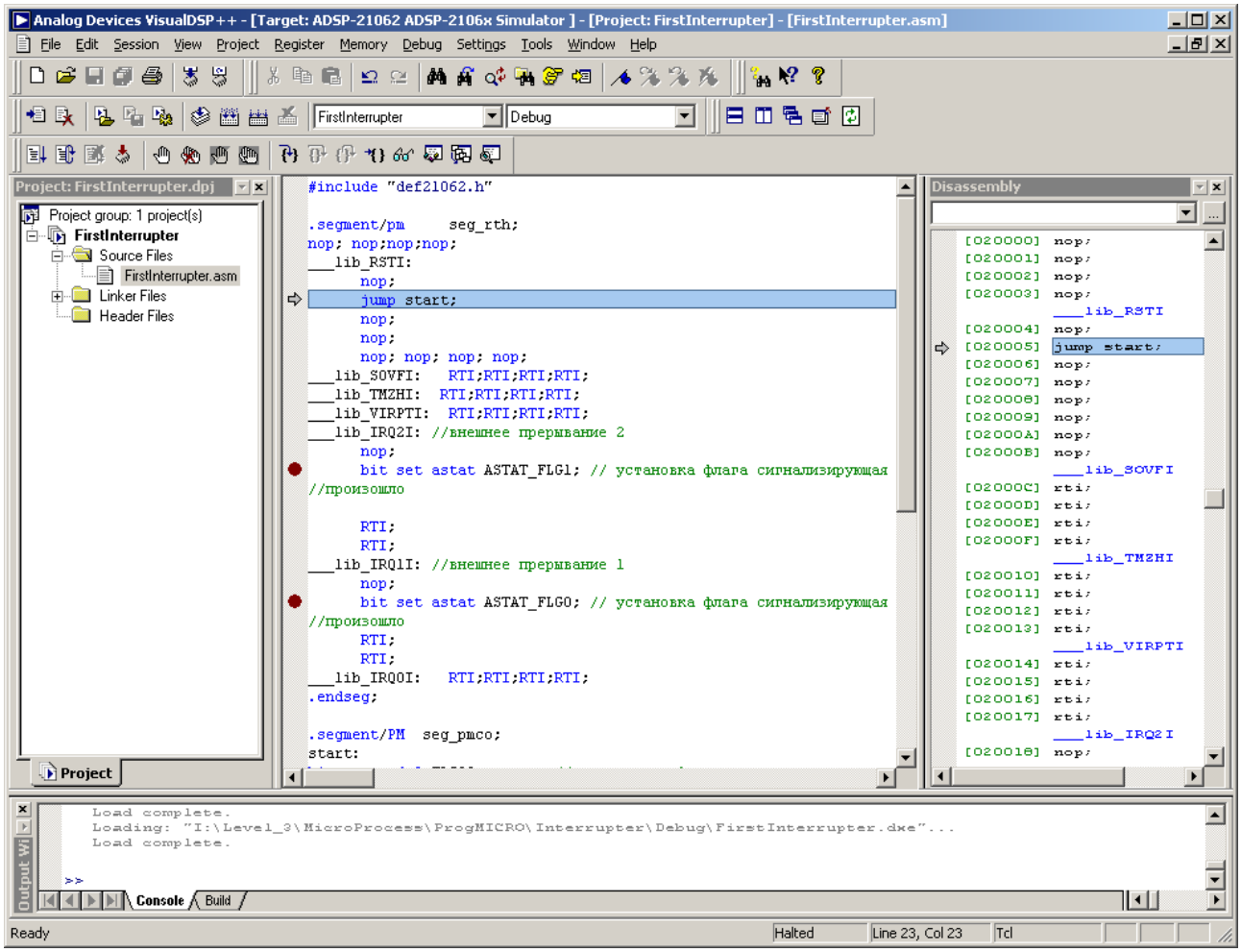
Коротко о работе программы. Программа загружается в процессор, настраивает вектор прерываний и переходит в режим ожидания (вечный цикл). По изменяющимся значениям флага **FLAG2** внешнее устройство узнает, что программа **ADSP** готова выполнять команды. Устройство посылает сигнал **SIRQ1**, при этом программа **ADSP** выходит из вечного цикла и начинает выполнять прерывание **IRQ1**. При этом выставляется флаг **FLAG0** и по команде **RTI** программа выходит из прерывания. После выхода из прерывания программа автоматически возвращается в цикл. Внешнее устройство опять узнает о готовности программы выполнять команды и посылает сигнал **SIRQ2**. При этом программа **ADSP** выходит из вечного цикла и начинает выполнять прерывание **IRQ2**. При этом выставляется флаг **FLAG1** и по команде **RTI** программа выходит из прерывания. После выхода из прерывания программа автоматически возвращается в цикл.

Проверка работоспособности программы. Для проверки необходимо смоделировать наличие внешнего устройства, посылающего команды. Используя пункт меню **Settings->Interrupts**, добиваемся появления на экране монитора окна **Interrupt Timing** и настраиваем его в соответствии с рисунком.



Т.е. мы определили два сигнала, которые имитируют сигналы от внешнего устройства. Появление данных сигналов приведет к совершению программой ADSP соответствующего прерывания.

Теперь устанавливаем контрольные точки **F9** напротив прерываний **IRQ1** и **IRQ2**, чтобы программа остановила работу в момент возникновения прерывания. Окно программы после нажатия **F7** должно выглядеть следующим образом.



Нажав клавишу F11 и удерживая ее, можно последовательно просмотреть последовательность выполнения команд программой. Дойдя до строчек

do mon until forever;

bit set astat ASTAT_FLG2;

mon:

bit clr astat ASTAT_FLG2;

программа зациклится. Чтобы заставить программу выполнять команды, не отображая их на экране монитора, необходимо нажать клавишу **F5**. После этого программа некоторое время будет находиться в цикле, пока не возникнет прерывание **IRQ1** и программа не остановится на контрольной точке. При повторном нажатии клавиши **F5** должно произойти прерывание **IRQ2**. Если это происходит, программа написана правильно, и прерывания действительно возникают.

4.4. Программа 3. Передача данных

Программа должна обеспечивать передачу данных между процессором и внешним устройством через внешний порт, не приостанавливая работу программы. Данная возможность может быть реализована с помощью прямого доступа к памяти.

Прямой доступ к памяти (Direct Memory Access – DMA) обеспечивает механизм для передачи целого блока данных. Использование расположенного на кристалле процессора контроллера DMA позволяет уменьшить нагрузку на ядро процессора при передаче данных между внутренней памятью и внешним источником данных или внешней памятью. Интегрированный в процессор контроллер DMA позволяет ядру процессора или внешнему устройству инициализировать операцию передачи данных и вернуться к выполнению основной программы, в то время как контроллер DMA выполняет передачу данных независимо от ядра процессора.

Настройка передач по DMA

Операции контроллера **DMA** могут программироваться ядром процессора, внешним хост-процессором или (внешним) ведущим **ADSP-2106x**. Для программирования операции **DMA** выполняется запись заданных

параметров передачи в отображенные в карте памяти регистры управления **DMA** и регистры параметров. Выбор канала **DMA** осуществляется в регистрах параметров **DMA**.

В регистры II, IM и C должны соответственно загружаться начальный адрес для буфера, модификатор адреса и число слов. Внешние порты, линк-порты и последовательные порты имеют в своем основном регистре управления бит разрешения **DMA** (**DEN**). Как только канал **DMA** настроен и активизирован, принятые слова данных автоматически пересылаются в буфер внутренней памяти. Когда **ADSP-2106x** готов передавать данные, слово автоматически пересылается из внутренней памяти в буферный регистр **DMA**. Эти передачи продолжаются до тех пор, пока не будет передан или принят весь буфер данных.

Когда весь блок данных передан, генерируется прерывание **DMA**. Это происходит, когда значение регистра счетчика канала **DMA (C)** уменьшится до нуля. Прерывания **DMA** фиксируются и маскируются в регистрах IRPTL и **IMASK** соответственно; эти регистры находятся в ядре процессора **ADSP-2106x**, а не в пространстве отображенных в карте памяти регистров устройства ввода/вывода.

Для того чтобы начать новую передачу по **DMA** после завершения предыдущей, в программе сначала нужно обнулить разрешающий бит **DEN**, записать новые параметры в регистры **II**, **IM** и **C**, а затем снова установить бит **DEN** в единицу.

II7, IM7, C7, CP7 − Регистры параметров 7-го канала **DMA** внешнего порта.

II8, IM8, C8, CP8 − Регистры параметров 8-го канала **DMA** внешнего порта.

Каждый канал DMA внешнего порта имеет свой регистр управления. Регистры называются **DMAC6**, **DMAC7**, **DMAC8** и **DMAC9** соответственно для каналов 6−9. Заметим, что в ADSP-21061 есть только каналы 6 и 7 **DMA** внешнего порта. В таблице показано содержимое регистров **DMACx**. Все

биты являются активными, когда установлены в единицу, если другое не оговаривается специально.

Биты управления начинают действовать во втором цикле после выполнения записи в регистр. Исключением является бит **FLSH**, который начинает действовать в третьем цикле после записи.

Регистр управления **DMA** через внешний порт (**DMACx**)

Бит	Имя	Описание
0	DEN	Разрешение DMA через внешний порт
1	CHEN	Разрешение цепочки операций DMA через внешний порт
2	TRAN	Передача/прием (1=передача, 0=прием)
3−4	PS	Состояние упаковки (только для чтения)
5	DTYPE	Тип данных (0=данные, 1=команды)
6−7	PMODE	Режим упаковки (00=нет, 01=16→32, 10=16→48, 11=32→48)
8	MSWF	Старшее слово первое при упаковке
9	MASTER	Разрешение DMA в режиме ведущего
10	HSHAKE	Разрешение DMA в режиме с квитированием
11	INTIO	Разрешение прерывания после передачи одного слова для буферов внешнего порта
12	EXTERN	Разрешение DMA во внешнем режиме с квитированием
13	FLSH	Очистка буферов DMA и битов состояния
14−15	FS	Состояние буфера внешнего порта (00=пустой, 11=полный, 10=частично заполненный)
16−31		зарезервированы

Теперь мы готовы написать программу, осуществляющую передачу по DMA через внешний порт в режиме с квитированием.

```
#include "def21062.h"

#define N 0x100

.segment/pm     seg_rth;
nop; nop;nop;nop;
___lib_RSTI:
  NOP;
  jump start;
  nop;
  NOP;
NOP;NOP;NOP;NOP;
___lib_SOVFI:   RTI;RTI;RTI;RTI;
___lib_TMZHI:   RTI;RTI;RTI;RTI;
___lib_VIRPTI:  RTI;RTI;RTI;RTI;
___lib_IRQ2I:
  nop;
  jump IRQ2_handler(db);
  r0=2;
  nop;
___lib_IRQ1I:
  nop;
  jump IRQ1_handler(db);
  r0=1;
  nop;
___lib_IRQ0I:   RTI;RTI;RTI;RTI;
// Зарезервированное прерывание
NOP;NOP;NOP;NOP;
.endseg;

.SECTION/DM  seg_dmda;    // Описываем начало сегмента, хранящего данные
.VAR DataFromDevice[N];   // Объявляем массив переменных из N слов
.VAR DataToDevice[N];     // Объявляем массив переменных из N слов
.endseg;                  // Конец сегмента, хранящего данные

.segment/PM  seg_pmco;
start:
r0=0x0;
dm(SYSCON)=r0;
nop;
nop;
nop;
nop;
bit set mode2 FLG0O;//направление флага на передачу
```

```
bit set mode2 FLG1O;//направление флага на передачу
bit set mode2 FLG2O;//направление флага на передачу
r0=0x0000;      //Инициализация всех регистров адреса нулевыми значениями

   bit clr astat ASTAT_FLG0;
   bit clr astat ASTAT_FLG1;
   bit clr astat ASTAT_FLG2;

   r1=0x0;
   imask=r1;            // обнуление регистра IMASK
   irptl=r1;            // обнуление регистра IRPTL
   imaskp=r1;           // обнуление регистра IMASKP
       bit set mode2 IRQ1E;     // разрешение прерывания
       bit set mode2 IRQ2E;     // разрешение прерывания

bit set mode1 IRPTEN;   //разрешение глобальных прерываний
bit set imask IRQ1I;    //внешнее прерывание
bit set imask IRQ2I;    //внешнее прерывание

do mon until forever;
bit set astat ASTAT_FLG2;
mon:
bit clr astat ASTAT_FLG2;

IRQ1_handler:
bit set astat ASTAT_FLG0;

r0=0x3000;//код обнуления буфера DMA в регистр общего назначения
pm(DMAC8)=r0;//очистка буфера
nop;nop;

i0=DataFromDevice;
m0=1;
r0=i0;
dm(II7)=r0;//Запись начального адреса
r0=0x400000;
dm(EI7)=r0;//начальный адрес для внешнего устройства
r0=1;
dm(IM7)=r0;//модификатор адреса
r0=@DataFromDevice;//количество слов передачи, равное размеру массива DataFromDevice в словах
dm(C7)=r0;//длительность выборки
dm(EC7)=r0;//длительность для внешнего устройства
r0=0;
```

```
dm(EM7)=r0;//регистр модификатора обнуляется
r0=0x401;//установить восьмой канал DMA на передачу данных
pm(DMAC7)=r0;//инициализация седьмого канала DMA
nop;//пустые циклы для применения изменений
nop;//пустые циклы для применения изменений
nop;//пустые циклы для применения изменений
nop;//пустые циклы для применения изменений
rti;// DMA настроено −выход из прерывания

IRQ2_handler:
bit set astat ASTAT_FLG1;

r0=0x3000;//код обнуления буфера DMA в регистр общего назначения
pm(DMAC7)=r0;//очистка буфера
nop;nop;

i0=DataToDevice;
m0=1;
r0=i0;
dm(II8)=r0;//Запись начального адреса
r0=0x400000;
dm(EI8)=r0;//начальный адрес для внешнего устройства
r0=1;
dm(IM8)=r0;//модификатор адреса
r0=@DataToDevice;//количество слов передачи, равное размеру массива DataFromDevice в словах
dm(C8)=r0;//длительность выборки
dm(EC8)=r0;//длительность для внешнего устройства
r0=0;
dm(EM8)=r0;//регистр модификатора обнуляется
r0=0x401;//установить восьмой канал DMA на передачу данных
pm(DMAC8)=r0;//инициализация восьмого канала DMA
nop;//пустые циклы для применения изменений
nop;//пустые циклы для применения изменений
nop;//пустые циклы для применения изменений
nop;//пустые циклы для применения изменений
rti;//ДМА настроено −выход из прерывания

nop;
idle;
.endseg;
```

В основе приведенной программы лежит программа 2. Т.е. описан вектор прерываний, состоящий из трех активных прерываний **Reset**, **IRQ1** и

IRQ2. В обработчике прерывания **IRQ1** программа процессора устанавливает флаг **FLG0,** а в обработчике прерывания **IRQ2** программа процессора устанавливает флаг **FLG1.** По состоянию флагов внешнее устройство определяет активное прерывание.

Внешнее устройство по номеру прерывания определяет, настроено **DMA** или нет, и в соответствии с настройкой **DMA** либо передает, либо принимает данные.

Первой особенностью данной программы является наличие обработчиков прерываний, состоящих более чем из четырех команд. Большой обработчик необходим для настройки **DMA** (порядка 20 команд). Только после этого данные будут передаваться от процессора к внешнему устройству или наоборот. Для увеличения количества команд в обработчике прерываний используется команда безусловного перехода **Jump**.

```
___lib_IRQ2I:
  nop;
  jump IRQ2_handler(db);
  r0=2;
  nop;
```

jump IRQ2_handler(db); команда перехода по метке **IRQ1_handler**, при этом выполнение прерывания продолжается до тех пор, пока не будет достигнута команда **RTI**. Только после этого произойдет выход из прерывания. Если объединить строки обработчика прерывания **___lib_IRQ2I** в одну программу, то он будет иметь вид:

```
___lib_IRQ2I: //метка
nop;
jump IRQ2_handler(db); // операция безусловного задержанного перехода
r0=2;
```

nop;

IRQ2_handler: *//метка*

… *// команды обработчика*

rti;*// выход из прерывания*

Второй особенностью данной программы является наличие универсального алгоритма настройки **DMA** через внешний порт. Отбросив промежуточные действия, алгоритм может быть записан в виде:

dm(II8)=r0;*//Запись начального адреса*

dm(EI8)=r0;*//начальный адрес для внешнего устройства*

dm(IM8)=r0;*//модификатор адреса*

dm(EM8)=r0;*//модификатор адреса внешнего устройства*

dm(C8)=r0;*//длительность выборки*

dm(EC8)=r0;*//длительность для внешнего устройства*

r0=0x401;*//установить восьмой канал* DMA *на передачу данных*

pm(DMAC8)=r0;*//инициализация восьмого канала* DMA

II8 определяет, с какого адреса необходимо начать считывать слова из внутренней памяти процессора.

EI8 определяет, в какой адрес внешней памяти будет записано считанное из внутренней памяти слово.

C8 определяет, сколько слов будет последовательно считано из внутренней памяти, начиная с начального адреса II8.

EC8 определяет, сколько слов будет последовательно записано во внешнюю память, начиная с начального адреса EI8.

IM8 определяет, с каким шагом будет модифицироваться значение адреса внутренней памяти (если он равен единице, то все слова будут считываться последовательно одно за другим; если он равен M, то будут считаны слова по адресам **II8, M+ II8, 2*M+ II8, 3*M+ II8,… , C8 *M+ II8).**

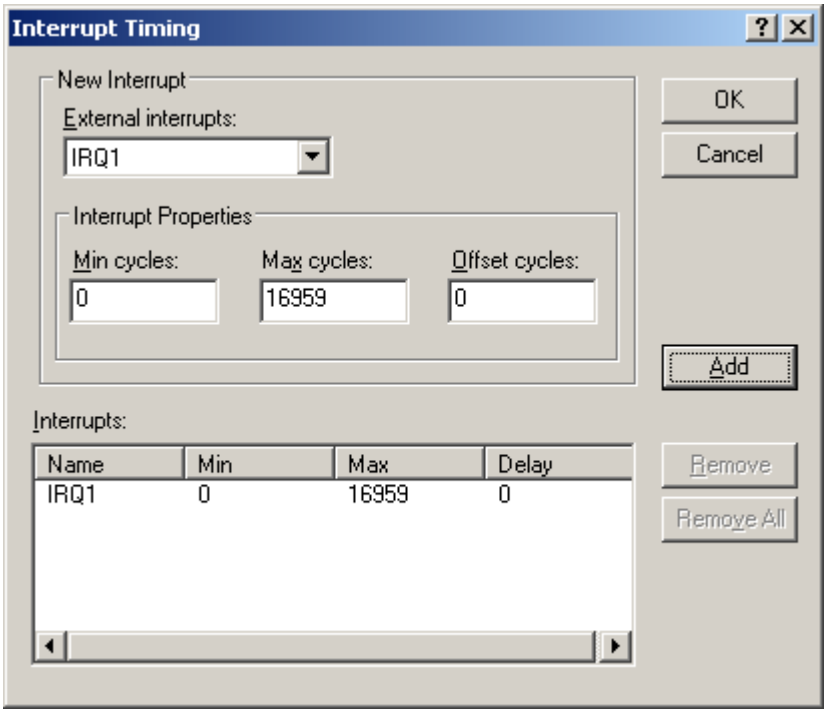
EM8 определяет, с каким шагом будет модифицироваться значение адреса внешней памяти.

Для проверки работоспособности программы необходимо произвести следующие действия:

- настроить внешние прерывания **IRQ1** и **IRQ2** по аналогии с предыдущим пунктом,
- настроить имитатор потока данных от внешнего устройства,
- настроить имитатор потока данных к внешнему устройству.
- скомпилировать проект и проверить работоспособность программы.

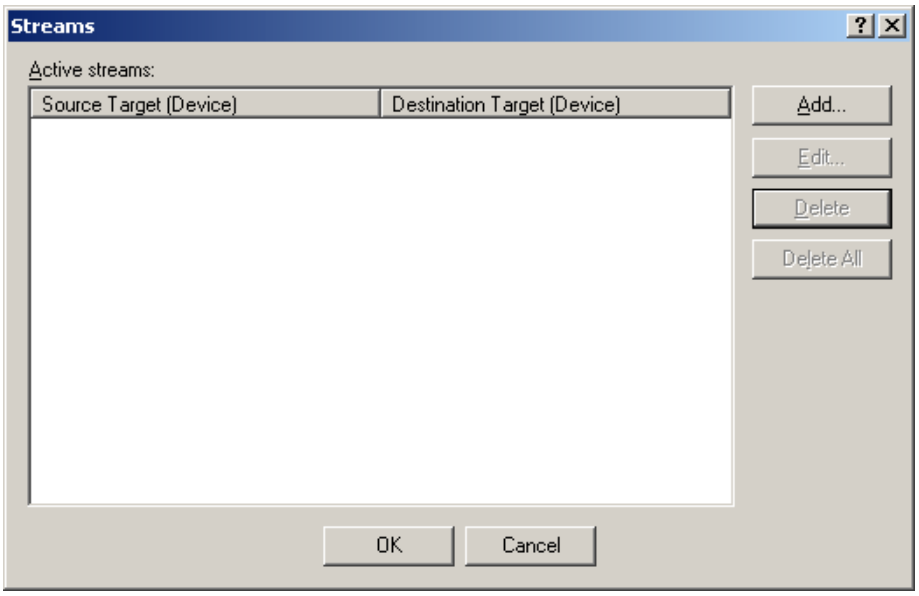
Проделаем операции последовательно:

1) Настраиваем **IRQ1**, которое должно обеспечить прием данных от внешнего устройства. Используя пункт меню **Settings->Interrupts,** добиваемся появления на экране монитора окна **Interrupt Timing**. Используя активные поля **Min Cycles**, **Max Cycles** и **External interrupt** и нажимая кнопку **OK**, добиваемся, чтобы окно **Interrupt Timing** имело следующий вид:

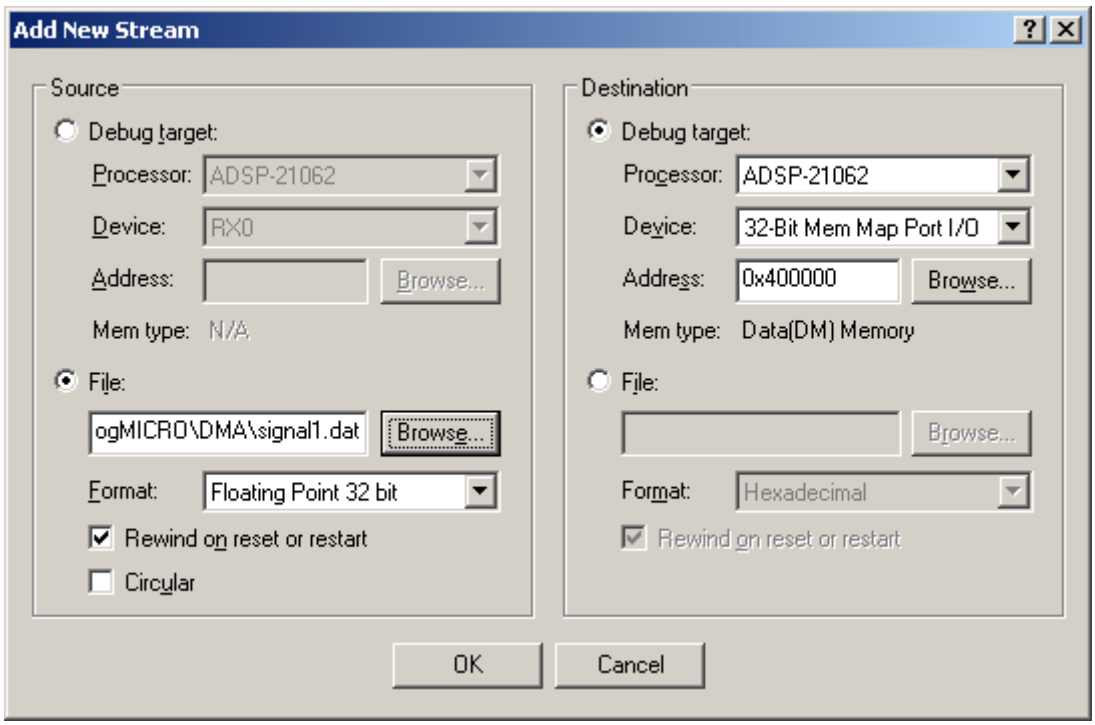


Теперь в интервале от 0 до 16959 тактов будет смоделировано прерывание от внешнего устройства.

2) Настраиваем имитатор потока данных от внешнего устройства к процессору. Используя пункт меню **Settings->Streams** добиваемся появления на экране монитора окна **Streams**.



Для настройки потока данных нажимаем кнопку **Add**, что приводит к появлению на экране монитора окна **Add New Stream**, активные поля которого необходимо заполнить в соответствии с образцом:



Пояснения к рисунку.

Источником, имитирующим поток данных, будет файл **signal.dat**. Для того чтобы подключить файл в качестве источника данных, необходимо на панели Source выбрать File: и нажать кнопку Browse... . В появившемся окне выбираем файл **signal.dat** в качестве источника и нажимаем кнопку OK.

Теперь необходимо настроить процессор на прием данных через внешний порт. Для этого на панели **Destination** выбираем Debug target: и настраиваем активные элементы управления.

Выбираем процессор **ADSP-21062**, устройство – 32-bit Mem Map Port I/O, адрес, с которого будут считываться данные,– 0x400000.

Нажимаем кнопку OK. После этого имитатор потока данных можно считать настроенным.

Для проверки работоспособности программы необходимо в редакторе кода напротив обработчика прерывания **IRQ1** установить точку остановки . Нажатием клавиши **F7** скомпилировать программу. После нажатия клавиши **F5** программа начнет работу. Когда будет сгенерировано прерывание **IRQ1,** выполнение программы приостановится. В окне редактора кода можно будет наблюдать:

```
__lib_IRQ1I: //RTI;RTI;RTI;RTI;
  nop;
  jump IRQ1_handler(db);
  r0=1;
  nop;
```

Нажимая клавишу **F11**, мы перейдем в режим пошаговой отладки (при каждом нажатии клавиши **F11** процессор будет выполнять одну команду). Нажимать клавишу **F11** будем до тех пор, пока не дойдем до строки кода **RTI**.

```
nop;//пустые циклы
rti;//ДМА настроено -выход из прерывания

IRQ2_handler:
bit set astat ASTAT_FLG1;
```

Теперь DMA настроено, и внешнее устройство начинает передавать данные в процессор.

Вопрос: каким образом можно отследить, началась передача данных или нет?

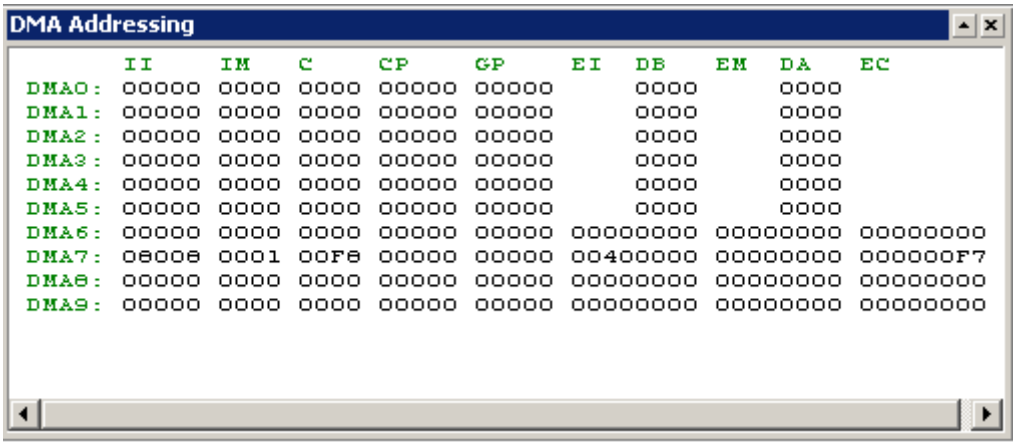
Ответ: необходимо, используя пункт меню **Register->IOP->DMA Addressing**, добиться появления на экране окна **DMA Addressing**.

DMA Addressing

	II	IM	C	CP	GP	EI	DB	EM	DA	EC
DMA0:	00000	0000	0000	00000	00000		0000		0000	
DMA1:	00000	0000	0000	00000	00000		0000		0000	
DMA2:	00000	0000	0000	00000	00000		0000		0000	
DMA3:	00000	0000	0000	00000	00000		0000		0000	
DMA4:	00000	0000	0000	00000	00000		0000		0000	
DMA5:	00000	0000	0000	00000	00000		0000		0000	
DMA6:	00000	0000	0000	00000	00000	00000000	00000000	00000000		
DMA7:	08000	0001	0100	00000	00000	00400000	00000000	000000FF		
DMA8:	00000	0000	0000	00000	00000	00000000	00000000	00000000		
DMA9:	00000	0000	0000	00000	00000	00000000	00000000	00000000		

В окне можно видеть настроенный канал **DMA7**. В регистре **II7** записано значение 0x8000, являющееся начальным адресом переменной **DataFromDevice.** В регистре **IM7** хранится значение 1. Количество слов, которые процессор должен принять от внешнего устройства, хранится в регистре **C7** (0x100 = 1024 слова).

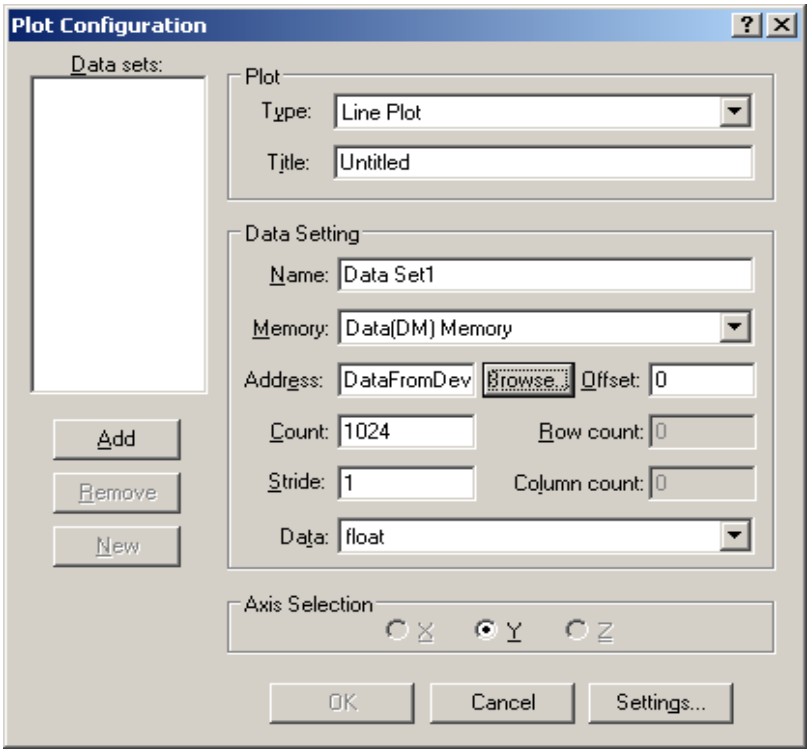
Нажимая клавишу **F11**, мы можем видеть, что содержимое окна **DMA Addressing** начинает меняться. Содержимое регистра **II7** растет, а содержимое регистра **C7** убывает на ту же величину. Это свидетельствует о том, что передача по **DMA** успешно начата.



На рисунке изображен случай, когда успешно передано 8 слов.

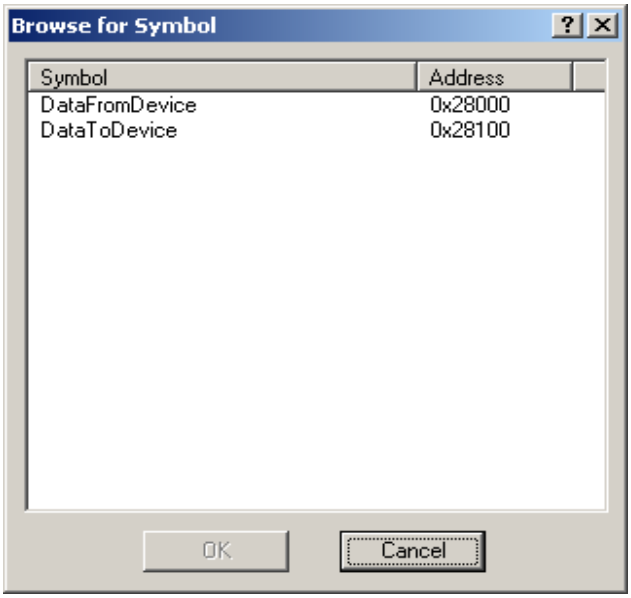
Вопрос: что же принимает процессор?

Ответ: для ответа на этот вопрос проще всего построить данные, поступающие от внешнего устройства. Для этого необходимо воспользоваться пунктом меню **View->Debug Windows->Plot->New**. В результате чего на экране появляется окно вида:



Используя активные поля, необходимо установить следующие значения:

- в поле **Address** необходимо ввести значение начального адреса переменной **DataFromDevice**. Для этого достаточно нажать кнопку **Browse** и в появившемся окне выбрать переменную **DataFromDevice**;



- в поле Name ввести имя графика **Device Data**;

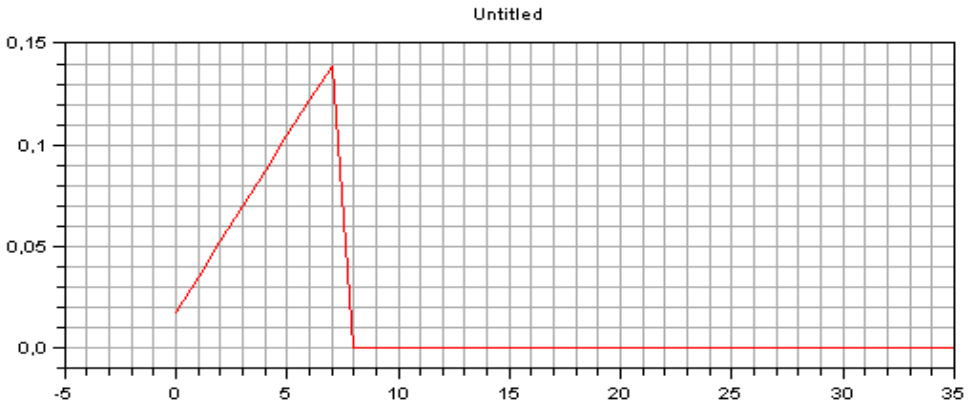
- в поле **Count** ввести значение 1024 (равное количеству слов, считанных с внешнего устройства);

- в поле **Stride** оставить 1;

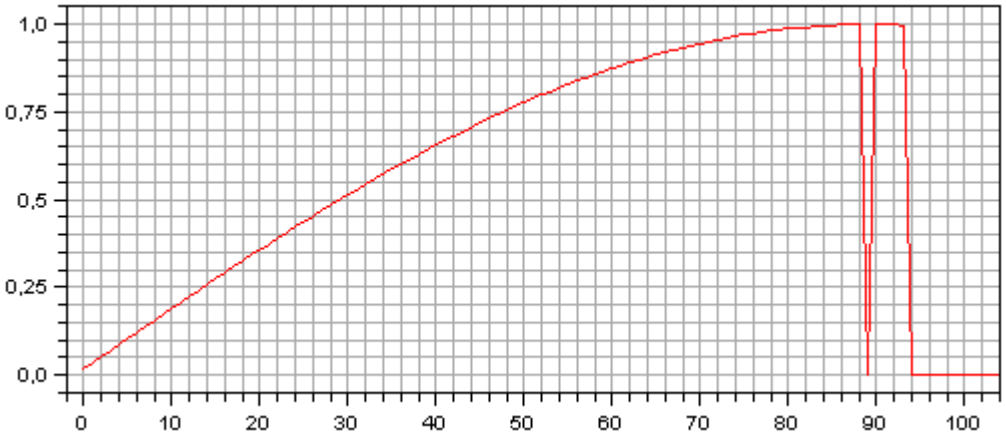
- в поле **Data** установить тип принимаемых данных **float**.

После чего нажать кнопку **Add**.

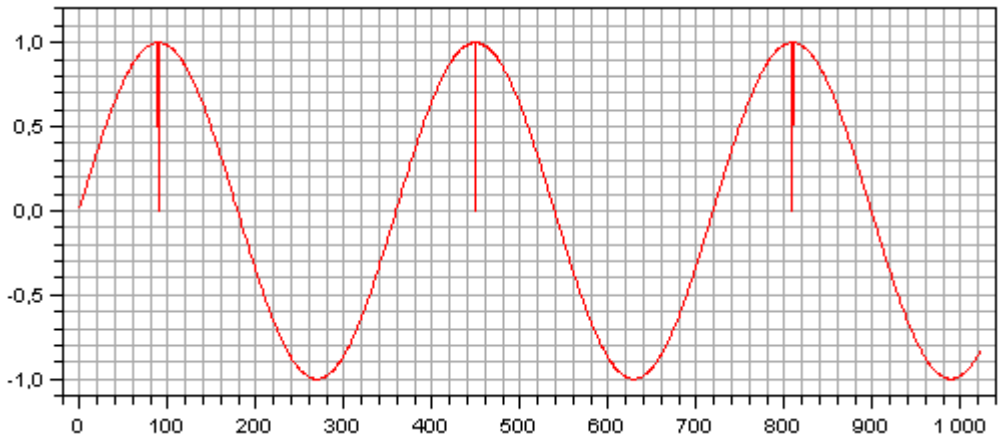
При этом в поле **Data sets** появится запись **Device Data**. Нажав кнопку **ОК**, мы можем наблюдать сигнал, принятый от внешнего устройства.



Из рисунка видно, что принято восемь отличных от нуля отсчетов сигнала. Чтобы посмотреть весь сигнал целиком, необходимо принять все 1024 отсчета сигнала. Это можно сделать, нажав клавишу **F5** либо, нажимая клавишу F11, смотреть, как последовательно появляются все новые и новые отсчеты сигнала. К примеру, на следующем рисунке принято 94 отсчета сигнала.



После приема всех 1024 слов сигнал выглядит следующим образом.



На этом описание программ на ассемблере можно считать оконченным.

Список ключевых слов, используемых при разработке программ на ассемблере.

Ключевые слова, используемые при разработке программ на ассемблере для процессоров **SHARC,** приведены в таблице				
__ADI__	__DATE__	__FILE__	__LINE__	__STDC__
__TIME__				
.ALIGN	.ELIF	.ELSE	.ENDIF	.EXTERN
.FILE	.GLOBAL	.IF	.IMPORT	.LEFTMARGIN
.LIST	.LIST_DATA	.LIST_DATFILE	.LIST_DEFTAB	.LIST_LOCTAB
.LIST_WRAPDATA	.NEWPAGE	.NOLIST_DATA	.NOLIST_DATFILE	.NOLIST_WRAPDATA
.PAGELENGTH	.PAGEWIDTH	.PRECISION	.ROUND_MINUS	.ROUND_NEAREST
.ROUND_PLUS	.ROUND_ZERO	.PREVIOUS	.SECTION	.STRUCT
.VAR	.WEAK			
ABS	ACS	ACT	ADDRESS	AND
ASHIFT	ASTAT	AV		
B0	B1	B2	B3	B4
B5	B6	B7	B8	B9
B10	B11	B12	B13	B14

B15	BB	BCLR	BF	BIT
BITREV	BM	BSET	BTGL	BTSTS
BY				
CA	CACHE	CALL	CH	CI
CJUMP	CL	CLIP	COMP	COPYSIGN
COS	CURLCNTR			
DADDR	DB	DEC	DEF	DIM
DMA1E	DMA1S	DMA2E	DMA2S	DMADR
DMABANK1	DMABANK2	DMABANK3	DMAWAIT	DO
DOVL				
EB	ECE	EF	ELSE	EMUCLK
EMUCLK2	EMUIDLE	EMUN	ENDEF	EOS
EQ	EX	EXP	EXP2	
F0	F1	F2	F3	F4
F5	F6	F7	F8	F9
F10	F11	F12	F13	F14
F15	FADDR	FDEP	FEXT	FILE
FIX	FLAGO_IN	FLAG1_IN	FLAG2_IN	FLAG3_IN
FLOAT	FLUSH	FMERG	FOREVER	FPACK
FRACTIONAL	FTA	FTB	FTC	FUNPACK
GCC_COMPIL	GE	GT		

ED				
I0	I1	I2	I3	I4
I5	I6	I7	I8	I9
I10	I11	I12	I13	I14
I15	IDLEI15	IDLE16	IF	IMASK
IMASKP	INC	IRPTL		
JUMP				
L0	L1	L2	L3	L4
L5	L6	L7	L8	L9
L10	L11	L12	L13	L14
L15	LA	LADDR	LCE	LCNTR
LE	LADDR	LCE	LCNTR	LE
L15	LA	LADDR	LCE	LCNTR
LE	LEFTO	LEFTZ	LENGTH	
LINE	LN	LOAD	LOG2	LOGB
LOOP	LR	LSHIFT	LT	
M0	M1	M2	M3	M4
M5	M6	M7	M8	M9
M10	M11	M12	M13	M14
M15	MANT	MAX	MBM	MIN
MOD	MODE1	MODE2	MODIFY	MROB
MROF	MR1B	MR1F	MR2B	MR2F

MRB	MRF	MS	MV	MROB
MROF				
NE	NOFO	NOFZ	NOP	NOPSPECIAL
NOT	NU			
OFFSETOF	OR			
P20	P32	P40	PACK	PAGE
PC	PCSTK	PCSTKP	PM	PMADR
PMBANK1	PMDAE	PMDAS	POP	POVLO
POVL1	PSA1E	PSA1S	PSA2E	PSA3E
PSA3S	PSA4E	PSA4S	PUSH	PX
PX1	PX2			
R0	R1	R2	R3	R4
RF5	R6	R7	R8	R9
R10	R11	R12	R13	R14
R15	READ	RECIPS	RFRAME	RND
ROT	RS	RSQRTS	RTI	RTS
SCALB	SCL	SE	SET	SF
SIN	SIZE	SIZEOF	SQR	SR
SSF	SSFR	SSI	SSIR	ST
STEP	STKY	STRUCT	STS	SUF
SUFR	SV	SZ		
TAG	TCOUNT	TF	TGL	TPERIOD

TRUE	TRUNC	TST	TYPE	TRAP
UF	UI	UNPACK	UNTIL	UR
USF	USFR	USI	USIR	USTAT1
USTAT2	UUF	UUFR	UUIR	UUIR
VAL				
WITH				
XOR				

Описание операторов встроенного ассемблера в порядке уменьшения приоритета.

Оператор	**Описание**
* / %	Умножение Деление Модуль
+ -	Сложение Вычитание
<< >>	Сдвиг влево Сдвиг вправо
&	Побитовое И (только для предпроцессора)
|	Побитовое ИЛИ
^	Побитовое исключающее ИЛИ (только для предпроцессора)
&&	Логическое И
||	Логическое ИЛИ

5. Лабораторные работы

При работе с симулятором не требуется наличия сигнального процессора, поскольку среда VisualDSP++ моделирует его работу. При моделировании можно управлять выполнением программы, просматривать состояние всех регистров и памяти процессора, организовывать работу устройства ввода/вывода и т.п.

Для освоения среды VisualDSP++ необходимо выполнить стандартные примеры программ Test_ALU62, Test_MAC62 и ADD_VEC62.

5.1. Лабораторная работа 1. Исследование арифметико-логического устройства

Программа исследования арифметико-логического устройства содержит примеры присвоения значений чисел с фиксированной и плавающей точкой и некоторых действий. Для запуска этой программы следует запустить среду **VisualDSP++** и создать папку для проекта **Test_ALU65**. В эту папку копируются файлы **test_alu.ASM** и **test_alu_modify.LDF**. Затем создается файл нового проекта способом, изложенным выше. С помощью пункта меню **Project->Build File** производится формирование исполняемого файла с последующей автоматической загрузкой его в симулятор.

После загрузки на экране можно отобразить исходный текст программы из файла **test_alu.ASM** и загруженный в память программы исполняемый файл. Исходный текст вызывается из окна проекта, а исполняемый − из меню View>DebugWindows>Disasembly. Текст программы содержит начало таблицы векторов прерывания, содержащее прерывание Reset и команду перехода jump main к метке start.

```
#include "def21060.h"          /* Memory Mapped IOP register definitions */

.SECTION/DM     seg_dmda;       /* Declare variables in data memory */

.SECTION/PM     seg_pmda;       /* Declare variables in program memory
*/

.SECTION/PM     seg_rth;        /* The reset vector resides in this space */
                nop;
                nop;
                nop;
                nop;
                nop;
                nop;
                nop;
                JUMP start;

.SECTION/PM     seg_pmco;
start:
                CALL testALU (DB);          /* Example delayed call instruction
*/
                nop;
                nop;

end:            IDLE;

/*_____________________Programm
ALU_____________________________*/

testALU:
/*Intialize data registers*/
R0=1;
R1=2;
R5=3;
R6=4;
R8=5;
R9=6;
R10=7;
R11=8;

F13=3.5e+20;
F14=5.0e+19;

/*Fixed point ALU operations*/
R2=R1+R0, R3=R1-R0;
```

```
R4=ABS R3;
/*Floating point ALU operations*/
F15=F13+F14;
F15=MIN(F13,F14);
F12=FLOAT R9;
F12=FLOAT R9 BY R8;
F15=FLOAT R10 BY R8;
R12=FIX F12 BY R11;

RTS;
```

Чтобы проследить выполнение программы по шагам, следует пользоваться пошаговым исполнением с помощью клавиши F11. Исполняемая строка подсвечивается синим цветом в исходном и исполняемом файлах.

После метки **test_alu** выполняется ряд присвоений чисел с фиксированной и плавающей точкой. Для наблюдения результата присвоения необходимо вызвать окно регистров с помощью меню **Register>Core>Register File**. Необходимый формат представления чисел в окне регистров устанавливается правой кнопкой мыши. Можно сократить количество выведенных в окно регистров, если использовать путь **Register>Custom**, позволяющий выбирать в окно необходимые регистры процессора. Особое внимание рекомендуется обратить на представление чисел с плавающей точкой в формате IEEE. Согласно этому формату, значение числа равно

$$(-1)^{s} (1.F)2^{e-127},$$

где s – знаковый разряд, e – 8-разрядный показатель степени (без знака), 1.F – мантисса с добавленным скрытым битом.

При изучении форматов записи и результатов выполнения операций рекомендуется использовать двоичное представление чисел в окне регистров. После выполнения всех команд программа остановится на команде **idle**. Повторный запуск программы выполняется нажатием кнопки **Restart** или с помощью меню **Debug>Restart**.

5.2. Лабораторная работа 2. Исследование умножителя

Для изучения действия умножителя-аккумулятора необходимо создать новую папку проекта и перенести в нее файлы **text_mux.ASM** и **text_mux_modify.LDF**. Затем выполняется формирование нового проекта, его построение и загрузка в симулятор. Эту программу также рекомендуется выполнить в пошаговом режиме, контролируя выполнение операций в окне регистров.

```
	#include "def21060.h"		/* Memory Mapped IOP register definitions
*/

	.SECTION/DM	seg_dmda;	/* Declare variables in data memory */
	.SECTION/PM	seg_pmda;	/* Declare variables in program memory
*/
	.SECTION/PM	seg_rth;	/* The reset vector resides in this space */
			nop;
			nop;
			nop;
			nop;
			nop;
			nop;
			nop;
			JUMP start;

	.SECTION/PM	seg_pmco;
	start:
			jump testMUX (DB);	/* Example delayed jump instruction
*/
			nop;
			nop;

	end:		IDLE;

	/*_____________________Programm
MUX_____________________________*/

	testMUX:

	/* Init data values */
	r0=0x2;
	r1=0x1234;
```

```
r2=0xfffffffe;
r3=0x40000000;
r4=0x33333333;

/*Fixed point integer multiplies*/
mrf=r0*r1 (ssi);
r8=r0*r1 (ssi);
mrb=r1*r2 (ssi);
mrb=r1*r2 (sui);
mrb=sat mrb (si);
r9=r1*r2 (sui);

/*Fixed point fractional multiplies*/
mrf=r3*r4 (ssf);
mrb=r3*r4 (ssfr);
mrb=r3*r4 (usf);
r10=r3*r4 (usf);

/*Floating point multiplies*/
f11=f5*f6;
/*change RND32 in MODE1 window to 1*/
f12=f5*f6;
f13=f5*f2;

jump start;
```

Особенность работы умножителя заключается в том, что алгоритм умножения с фиксированной точкой зависит от вида числа. Поэтому команда умножения чисел с фиксированной точкой всегда содержит модификатор. Сомножителю со знаком соответствует обозначение s, без знака – u, умножение целых чисел обозначается – i, дробных – f, получение округленного результата – r.

5.3. Лабораторная работа 3. Сложение векторов

В качестве более сложной программы рассмотрим программу сложения векторов. Для изучения программы необходимо создать новую папку проекта и перенести в нее файлы ADDVEC.ASM (Листинг 2.3) и EZLAB_21061.LDF.

Затем выполняется формирование нового проекта, его построение и загрузка в симулятор.

```
# define n 10

.SECTION/PM seg_pmda;
.var    x_input[n] = "xin.dat";

.SECTION/DM seg_dmda;
.var    y_input[n] = "yin.dat";
.var    rezalt[n];

.SECTION/PM seg_rth;
nop;
nop;
nop;
nop;
nop;
jump start;
/*.ENDSEG;*/

.SECTION/PM   seg_pmco;

start:

/*Intialize data registers*/

I8=x_input;
m8=1;
l8=0;
i1=y_input;
m1=1;
l1=0;
I4=rezalt;
m4=1;
l4=0;
LCNTR=n-1;
 r8=PM(I8,M8), r13=dm(i1,m1);
DO next UNTIL LCE;
 r3=r8+r13, r13=dm(i1,m1),r8=pm(i8,m8);
 next: dm(i4,m4)=r3;
r3=r8+r13;
dm(i4,m4)=r3;
```

idle;

.ENDSEG;

Программа начинается с директивы предпроцессора # define, которая присваивает значение размерности векторов, равное 10.

Суммируемые векторы записываются в буферы x_input[n] и y_input[n], расположенные в блоках памяти программы и данных. Использование двух разных блоков памяти позволяет в течение одного цикла процессора одновременно извлечь значения этих двух векторов. Директива описания переменных **.var** содержит также указание, что численные значения векторов вводятся линкером из файлов **xin.dat** и **yin.dat**. Эти файлы должны быть составлены заранее и содержатся в папке задачи. При отсутствии этих файлов исполняемый файл не формируется из-за ошибки линкера. Вектор суммы формируется в буфере **rezult**.

Перед пошаговым выполнением программы необходимо посмотреть распределение исходных массивов векторов в памяти и содержимое регистров, используемых в задаче.

Для просмотра памяти необходимо из пункта меню **Memory->Two ColumnData** вызвать окно отображения содержимого области памяти данных. Затем после нажатия правой кнопки мыши выбираем **Go To** и с помощью клавиши браузера **Browse** находим в памяти нужные переменные. Для просмотра содержимого регистров используется пункт меню **Register>Custom**. Кроме используемых вычислительных регистров необходимо выбрать регистры генератора адреса (i1, i4, i8).

При пошаговом исполнении программы наблюдаем инициализацию индексных регистров генератора адреса и счетчика циклов LCNTR. После этого выполняем операции цикла суммирования. Особенность этой программы заключается в использовании многофункциональной команды, содержащей операцию сложения и два обращения к памяти. Эта команда

выполняется за один цикл процессора. Во всех сигнальных процессорах считывание данных из регистров выполняется в начале цикла процессора, а запись − в конце. По этой причине перед началом цикла выполняется присвоение вычислительных регистров r8 и r13, а последнее суммирование выполняется после завершения цикла. Из-за подобной особенности многофункциональных операций длина цикла равна n-1 и вычисление последней составляющей суммарного вектора выполняется вне цикла.

5.4. Лабораторная работа 4. Запись в память процессора

Для изучения процедуры записи во внутреннюю память процессора необходимо создать новую папку проекта и перенести в нее файлы Int_memory.ASM и Int_memory.LDF. Затем выполняется формирование нового проекта, его построение и загрузка в симулятор. Эту программу также рекомендуется выполнить в пошаговом режиме, контролируя выполнение операций в окне регистров.

```
#include "def21060.h"
#define N 100

.SECTION/DM     seg_dmda;
.VAR DMinput[N];
.SECTION/PM     seg_pmda;
.VAR PMinput[N];

.SECTION/PM     seg_rth;
                nop;
                nop;
                nop;
                nop;
                nop;
                JUMP start;
                nop;
                nop;

.SECTION/PM     seg_pmco;
start:
i0=DMinput;
```

```
        m0=1;
        i8=PMinput;
        m8=1;
        r0=5;
        r1=3;
        start1:
                        call testDM (DB);          /* Example delayed call instruction */
                        nop;
                        nop;
                        jump testPM (DB);          /* Example delayed jump instruction
*/
                        nop;
                        nop;

        end:            IDLE;

        /*_________________Programm
Memory_____________________________*/
        testDM:
        r0=r0+r1;
        dm(i0,m0)=r0;
        RTS;

        testPM:
        r1=r1-r0;
        pm(i8,m8)=r1;
        jump start1;
```

Программа начинается с директивы **#define N 100,** которая определяет размеры областей памяти, куда будут записаны данные. В памяти программы и памяти данных процессора определяются две переменные **PMinput[N]** и **DMinput[N]**, в которые будет непосредственно производиться запись. Программа реализует процедуру записи в память процессора с помощью команд **pm()** и **dm()**. цикличность записи реализована с помощью команд jump и call. Для понимания последовательности выполнения команд необходимо запустить программу в пошаговом режиме и последовательно пройти ее, следя за изменениями в окнах **Active Register File**, **Active DAG1**, **Active DAG2**, **Data(DM) Memory** и **DataPlot**. Окна откроются автоматически при выполнении команды **File>WorkSpace>Open>Space.vdw**.

5.5. Лабораторная работа 5. Формирование задержки сигнала

Работа с отладочной платой отличается от работы с симулятором выбором сессии **Session> EZ kit LITE 21065l**. После переключения сессии компьютер просит нажать кнопку Reset на отладочной плате.

Проект формирования задержки сигнала содержит файлы **ezkit_65.ldf**, **ISR_65.asm**, **init_65.asm**, **SDRAM_65.asm**, **Clear_reg.asm**, **AD1819_65.asm**, **Codec.asm** и **delay_65.asm**. В системном файле **ezkit_65.ldf** необходимо закомментировать директиву **SHT_NOBITS**, т.к. в программе задержки используется внешняя динамическая память. Программа задержки содержит две подпрограммы.

```
/* определение бит регистров ADSP-21060 */

#include      "def21065l.h"
#include      "new65Ldefs.h"
#define TAPS 1000000

.GLOBAL          talk_through;
.GLOBAL          Init_User;
.EXTERN   Left_Channel_In;
.EXTERN   Right_Channel_In;
.EXTERN   Left_Channel_Out;
.EXTERN   Right_Channel_Out;

.segment/dm seg_sdram;
        .var dline[TAPS];
.segment /pm pm_code;
Init_User:


b0=dline; l0=TAPS; m0=0; m1=1; / *инициализация циклического
 буфера задержки */
rts;

talk_through:
        r0 = dm(Left_Channel_In);    /*прием левого канала */
        r1 = dm(Right_Channel_In);          /*прием правого канала */
        if not flag1_in jump mem;
```

```
        dm(i0,m0) = r0;    /* запись левого сигнала в буфер */
mem:

        r0=dm(i0,m1);     /* считывание задержанного сигнала */
        dm(Left_Channel_Out)  = r0; /*передача в левый канал
задержанного сигнала */
        dm(Right_Channel_Out) = r1;/*передача в правый канал
сигнала без задержки */
        rts;

.endseg;
```

Подпрограмма Init_User вызывается при инициализации платы с помощью модуля Init_65. В этой подпрограмме выполняется инициализация циклического буфера, расположенного во внешней динамической памяти. Длина этого буфера может составлять 1000000.

Подпрограмма talk_through вызывается с частотой приема данных в последовательном порту. В этой подпрограмме принятые данные правого канала передаются на выход без задержки. Принятые данные из левого канала при нажатии кнопки flag1_in поступают на запись в циклический буфер. Для заполнения циклического буфера длиной l0=1000000 при тактовой частоте 48 кГц требуется производить запись в течение 20 с. При отпускании кнопки флага запись прекращается, и на правый канал передачи поступают задержанные данные, повторяющиеся с периодом 20 с.

5.6. Лабораторная работа 6. Фильтр с конечной памятью

Проект фильтра с конечной памятью содержит файлы **ezkit_65.ldf**, **ISR_65.asm**, **init_65.asm**, **SDRAM_65.asm**, **Clear_reg.asm**, **AD1819_65.asm**, **Codec.asm** и **fir_65.asm**. В системном файле **ezkit_65.ldf** необходимо снять комментарий с директивы **SHT_NOBITS**, т.к. в программе фильтра не используется внешняя динамическая память. В модуле **fir_65.asm** содержится три подпрограммы.

В подпрограмме **Init_User** производится инициализация циклических

буферов задержки принятых данных и коэффициентов фильтра. Коэффициенты фильтра поступают из файла **band_fir2.dat**. Число коэффициентов определяется директивой предпроцессора

#define FILTER_TAPS 397.

При проектировании фильтра длина фильтра и название файла коэффициентов устанавливаются пользователем. Подпрограмма завершается очисткой буфера задержки.

Прием данных, вызов подпрограммы фильтрации и передача результата фильтрации выполняется в подпрограмме **talk**. Подпрограмма фильтра подобна, рассмотренной при описании отладочной платы процессора **ADSP-21061** и в листинге отсутствует.

```
/* определение бит регистров ADSP-21060 */
#include      "def21065l.h"
#include      "new65Ldefs.h"

#define FILTER_TAPS 397

.GLOBAL   Init_User;
.GLOBAL          talk;
.EXTERN   Left_Channel_In;
.EXTERN   Right_Channel_In;
.EXTERN   Left_Channel_Out;
.EXTERN   Right_Channel_Out;

.segment /dm    dm_data;
.VAR    dline[FILTER_TAPS];      /* линия задержки сигнала */
.endseg;

.segment /pm          pm_data;
.VAR    coef[FILTER_TAPS] = "band_fir2.dat";   /* объявление и
 присвоение коэффициентов */
.endseg;

.segment /pm pm_code;
Init_User:

   b0 = dline;     l0 = FILTER_TAPS;    /* индекс и длина
буфера задержки */
```

```
  m1 = 1;
  b8 = coef;      l8 = FILTER_TAPS;     /* индекс и длина
 буфера коэффициентов */
  m9 = 1;
  f12=0.0;                  /* очистка буфера задержки*/
  i3=dline;       l3=0;
  lcntr=FILTER_TAPS, do clear until lce;
clear:    dm(i3,1)=f12;

  rts;

talk:
        r14 = dm(Right_Channel_In);     /* прием правого канала */
        r1 = -31;                       /* масштабирование */
        f0=float r14 by r1;             /* преобразование в число
 с плавающей точкой */
        call fir;
        r1=31;                          /* обратное преобразование
в число с фиксированной точкой*/
        r8 = fix f8 by r1;
        r0=fix f0 by r1;
        dm(Right_Channel_Out) = r8;     /*передача в правый */
        dm(Left_Channel_Out)  = r8;     /*и левый каналы */
        rts;

.endseg;
```

Принятые данные преобразуются из формата с фиксированной точкой в формат с плавающей точкой и поступают в подпрограмму фильтра. Выходные данные фильтра перед передачей подвергаются обратному преобразованию.

Приложение А. Набор команд

А.1. Обзор

В этом и следующем приложениях подробно описывается набор команд **ADSP-2106x**. В этом приложении рассматриваются типы команд, синтаксис ассемблера и код операции, по которому транслируется команда. Многие типы команд имеют поле для определения операции вычисления. Это операции, которые используют **ALU**, умножитель или устройство сдвига. Таких команд много, поэтому они описываются отдельно в Приложении Б. (Заметим, что пересылка данных между регистрами **MR** и регистровым файлом рассматривается как операция умножителя.)

В этом разделе приводится характеристика каждой команды: синтаксис, функция, дается один или два примера на ассемблере и описываются различные поля ее кода операции. Команды группируются по четырем категориям:

I. Команды *вычисления и пересылки или модификации*, которые определяют параллельное выполнение операции вычисления с одной или двумя пересылками данных или с модификацией индексного регистра.

II. Команды *управления последовательностью выполнения программы*, которые определяют различные типы переходов, вызовов, возвратов и циклов. Некоторые из этих команд могут также содержать операцию вычисления и/или пересылки данных.

III. Команды *пересылки данных*, в которых непосредственное поле команды используется как операнд или для адресации.

IV. *Прочие* команды, такие как модификация и проверка бита, нет операции и простой.

Команды нумеруются от 1 до 23. Некоторые команды имеют несколько синтаксических форм, например, команда четвертого типа имеет четыре различные формы. При программировании номер команды не учитывается,

но он соответствует коду операции, распознаваемому аппаратурой ADSP-2106x.

Многие из команд могут быть условными. Этим командам предшествует IF плюс мнемоника условия. В условной команде выполнение всей команды основывается на заданном условии.

A.2. Карта регистров

PC Счетчик команд (PC)
PCSTK Вершина стека PC
PCSTKP Указатель стека PC
FADDR Адрес выборки
DADDR Адрес декодирования
LADDR Адрес завершения цикла
CURLCNTR Счетчик текущего цикла
LCNTR Счетчик цикла
R15_R0 Ячейки памяти регистрового файла
I15_I0 Индексные регистры DAG1 и DAG2
M15_M0 Регистры модификации DAG1 и DAG2
L15_L0 Регистры длины DAG1 и DAG2
B15_B0 Регистры базового адреса DAG1 и DAG2

Системные регистры

MODE1 Управление режимом 1
MODE2 Управление режимом 2
IRPTL Фиксация прерываний
IMASK Маскирование прерываний
IMASKP Указатель маски прерываний
ASTAT Арифметическое состояние

STKY «Залипшее» состояние

USTAT1 Регистр определяемого пользователем состояния

USTAT2 Регистр определяемого пользователем состояния

A.3. Команды вычисления и пересылки

1. Параллельная пересылка данных между памятью данных и памятью программы с использованием регистрового файла, необязательная операция вычисления

Синтаксис:

compute, DM(Ia,Mb)=dreg1 , PM(Ic,Md)=dreg2 ;

compute, dreg1= DM(Ia,Mb), dreg2=PM(Ic,Md);

Функция:

Параллельный доступ к памяти данных и памяти программы из регистрового файла. Определенные регистры I используются для адресации памяти данных и памяти программы. Значение I постмодифицируется и обновляется значением определенного регистра M. Предмодификация адреса со смещением не поддерживается.

Примечание: Ia и Ic должны быть из DM и PM соответственно.

Примеры:

R7=BSET R6 BY R0, DM(I0,M3)=R5, PM(I11,M15)=R4;

R8=DM(I4,M1), PM(I12,M12)=R0;

2. Операция вычисления, необязательное условие

Синтаксис:

IF условие вычисление ;

Функция:

Условная команда вычисления. Команда выполняется, если при проверке определенное условие верно.

Примеры:

IF MS MRF=0;

F6=(F2+F3)/2;

3. Пересылка данных между памятью данных или памятью программы и универсальным регистром, необязательное условие, необязательная операция вычисления

Синтаксис:

a. *IF условие вычисление,* DM(Ia,Mb) = ureg;

IF условие вычисление, PM(Ic,Md)= ureg;

b. *IF условие вычисление,* DM(Mb,Ia) = ureg;

IF условие вычисление, PM(Md,Ic) = ureg;

c. *IF условие вычисление,* ureg = DM(Ia,Mb);

IF условие вычисление, ureg = PM(Ic,Md);

d. *IF условие вычисление,* ureg = DM(Mb,Ia);

IF условие вычисление, ureg = PM(Md,Ic);

Функция:

Обращения между памятью данных или памятью программы и универсальным регистром. Определенные регистры I используются для адресации памяти данных и памяти программы. Значение I либо предмодифицируется (M, I), либо постмодифицируется (I, M) значением определенного регистра M. При постмодификации значение регистра I обновляется модифицированным значением.

Если определена операция вычисления, то она выполняется параллельно с доступом к данным. Если условие определено, то оно влияет на выполнение всей команды.

Примеры:

R6=R3-R11, DM(I0,M1)=ASTAT;

IF NOT SV F8=CLIP F2 BY F14, PX=PM(I12,M12);

A.4. Управление последовательностью выполнения программы

1.Косвенный (или относительно PC) переход и необязательная операция вычисления с передачей данных между памятью данных и регистровым файлом

Синтаксис:

IF условие JUMP (Md,Ic), ELSE *вычисление*, DM(Ia,Mb)=dreg

IF условие JUMP (PC,<reladdr24>), ELSE *вычисление,* dreg =DM(Ia,Mb)

Функция:

Условный переход по адресу, определяемому предмодифицированным значением регистра I, или по адресу относительно PC и необязательная операция вычисления параллельно с пересылкой данных между памятью данных и регистровым файлом. В этой команде условие IF и ключевое слово ELSE должны обязательно использоваться. Если заданное условие верно, то выполняется переход. Если заданное условие ложно, то операция вычисления и передача из памяти (в память) данных выполняются параллельно. В этой команде необязательна только операция вычисления.

Для перехода адрес относительно PC – это 6-разрядное число в виде двоичного дополнения. Если регистр I определен (Ic), он модифицируется значением определенного регистра M (Md) для генерации адреса перехода. Операция модификации не влияет на содержимое регистра I. Заметим, что модификаторы команды задержанного перехода (DB), прерывания цикла (LA) и очистки прерывания не разрешены в этой команде.

Для доступа к памяти данных адрес обеспечивается регистром I (Ia). Значение регистра I постмодифицируется значением определенного регистра M и обновляется модифицированным значением. Адресация с предмодификацией для такого доступа к памяти данных запрещена.

Примеры:

IF TF JUMP(M8,I8), ELSE R6=DM(I6,M1);

IF NE JUMP(PC,0X20), ELSE F12=FLOAT R10 BY R3, R6=DM(I15,M0);

2. Возврат из подпрограммы или прерывания, необязательное условие, необязательная операция вычисления

Синтаксис:

IF условие RTS *(DB), вычисление ELSE вычисление*;

IF условие RTS *(LR), вычисление ELSE вычисление*;

IF условие RTI *(DB), вычисление ELSE вычисление*;

Функция:

Возврат из подпрограммы (RTS) или возврат из программы обработки прерывания (RTI). Модификатор команды DB указывает, что переход задержанный, иначе он незадержанный.

Команда возврата из подпрограммы заставляет процессор выполнить переход по адресу, сохраненному в вершине стека счетчика команд. Различие между RTS и RTI заключается в том, что команда RTI не только извлекает адрес возврата из стека счетчика команд, но также: 1) извлекает содержимое стека состояния, если туда было помещено содержимое регистров состояния ASTAT и MODE1 (если было прерывание IRQ, прерывание по таймеру или векторное прерывание VIRPT); 2) обнуляет соответствующий бит в регистрах фиксирования прерывания (IRPTL) и указателя маски прерывания (IMASKP).

Возврат выполняется, если условие определено и верно. Если операция вычисления определена без ELSE, она выполняется параллельно с возвратом.

Если операция вычисления определена с ELSE, она выполняется только в том случае, если условие ложно. Заметьте, что условие должно быть определено, если определена фраза *ELSE вычисление*.

Если незадержанный вызов используется как одна из трех последних команд в цикле, то с командой RTS должен применяться модификатор повторного входа в цикл (LR). Команда RTS (LR) гарантирует правильный повторный вход в цикл. Например, в цикле, организованном по счетчику,

условие завершения цикла проверяется посредством уменьшения значения счетчика текущего цикла (CURLCNTR) в течение выполнения двух последних команд цикла. Команда RTS(LR) гарантирует правильный повторный вход в цикл, предотвращая декремент счетчика циклов (дважды для одной и той же итерации цикла). При возвращении из подпрограммы, в которую была сведена программа обработки прерывания командой JUMP (CI), должен использоваться модификатор (LR) команды RTS (в случае если прерывание поступает в течение двух последних команд цикла).

Примеры:

RTI, R6=R5 XOR R1;

IF NOT GT RTS(DB);

IF SZ RTS, ELSE R0=LSHIFT R1 BY R15;

3. Загрузка счетчика цикла, команда DO UNTIL LCE

Синтаксис:

LCNTR = <data16> , DO <addr24> UNTIL LCE ;

LCNTR = ureg , DO (PC,<reladdr24>) UNTIL LCE ;

Функция:

Инициализирует организованный по счетчику программный цикл. Счетчик цикла LCNTR загружается 16-разрядным непосредственным значением данных или значением из универсального регистра. Начальный адрес цикла помещается в стек PC. Адрес окончания цикла и условие окончания LCE помещаются в стек адреса цикла. Конечный адрес может быть меткой для абсолютного 24-разрядного адреса памяти программы или 24-разрядным адресом относительно PC в виде двоичного дополнения. LCNTR помещается в стек счетчика цикла и становится значением CURLCNTR. Цикл выполняется до тех пор, пока содержимое CURLCNTR не достигнет нуля.

Примеры:

LCNTR=100, DO fmax UNTIL LCE; (fmax – программная метка)

LCNTR=R12, DO (PC,16) UNTIL LCE;

4. Команда организации цикла DO UNTIL

Синтаксис:

DO <addr24> UNTIL условие окончания ;

DO (PC,<reladdr24>) UNTIL условие окончания ;

Функция:

Инициализирует организованный по счетчику программный цикл. Начальный адрес цикла помещается в стек PC. Адрес окончания цикла и условие окончания помещаются в стек цикла. Конечный адрес может быть меткой для абсолютного 24-разрядного адреса памяти программы или 24-разрядным адресом относительно PC в виде двоичного дополнения. Цикл повторяется до тех пор, пока не выполняется заданное условие окончания цикла.

Примеры:

DO end UNTIL FLAG_IN; (end – программная метка)

DO (PC,7) UNTIL AC;

A.5. Команды непосредственной пересылки данных

1. Пересылка данных между памятью данных или памятью программы и универсальным регистром, прямая адресация, непосредственное значение адреса

Синтаксис:

a. DM(<addr32>) = ureg ;

PM(<addr24>) = ureg ;

b. ureg = DM(<addr32>) ;

ureg = PM(<addr24>);

Функция:

Обращения между памятью данных или памятью программы и универсальным регистром при прямой адресации. Полный адрес памяти данных или памяти программы определяется в программе. Адреса памяти данных 32-разрядные (от 0 до 2^{32}-1). Адреса памяти программы 24-разрядные (от 0 до 2^{24}-1).

Примеры:

DM(temp)=MODE1; (temp – программная метка)

DMWAIT=PM(0x489060);

2. Пересылка данных между памятью данных или памятью программы и универсальным регистром, прямая адресация, непосредственный модификатор

Синтаксис:

a. DM(<data32>,Ia) = ureg ;

PM(<data24>,Ic) = ureg;

b. ureg = DM(<data32>,Ia)

ureg =PM(<data24>,Ic) ;

Функция:

Обращения между памятью данных или памятью программы и универсальным регистром при косвенной адресации с использованием регистров I. Регистр I предмодифицируется непосредственным значением, заданным в команде. Содержимое регистра I не обновляется. Модификаторы адреса памяти данных 32-разрядные (от 0 до 2^{32}-1). Модификаторы адреса памяти программы 24-разрядные (от 0 до 2^{24}-1).

Замечания:

1. Универсальный регистр (ureg) может находиться не в том же самом DAG (т. е. DAG1 или DAG2), что и Ia/Mb или Ic/Md.

Примеры:

DM(24,15)=TCOUNT;

USTAT1=PM(offs,I13); (“offs” – определенная константа)

3. Прямая запись данных в память данных или память программы

Синтаксис:

DM(Ia,Mb) = <data32>;

PM(Ic,Md) = <data32>;

Функция:

Запись 32-разрядных данных в память данных или память программы при косвенной адресации. Данные размещаются в старших 32 разрядах 40-разрядного слова памяти. Младшие 8 разрядов обнуляются. Регистр I постмодифицируется и обновляется определенным значением регистра M.

Примеры:

DM(I4,M0)=19304;

PM(I14,M11)=count; (count – константа, определяемая пользователем)

4. Прямая запись данных в универсальный регистр

Синтаксис:

ureg = <data32> ;

Функция:

Запись 32-разрядных данных в универсальный регистр. Если регистр 40-разрядный, данные размещаются в старших 32 разрядах, а в 8 младших записываются нули.

Примеры:

IMASK=0xFFFC0060;

M15=mod1; (mod1 – константа, определяемая пользователем)

A.6. Прочие команды

1. Операции с битами системного регистра

Синтаксис:

BIT SET sreg <data> ;

BIT CLR sreg <data> ;

BIT TGL sreg <data> ;

BIT TST sreg <data> ;

BIT XOR sreg <data> ;

Функция:

Команда операции с битами в системном регистре. При помощи этой команды можно устанавливать, очищать, переключать или проверять определенные биты, сравнивать по схеме «исключающее ИЛИ» (XOR) значение системного регистра с определенным значением данных. В первых четырех операциях значение данных является маской. Операция установки устанавливает в заданном системном регистре все биты, которые установлены в определенном значении данных. Операция очистки обнуляет все биты, которые установлены в значении данных. Операция переключения переключает все биты, которые установлены в значении данных. Операция проверки устанавливает флаг проверки бита (BTF в ASTAT), если все биты, которые установлены в значении данных, также установлены в системном регистре. Операция XOR устанавливает флаг проверки бита (BTF в ASTAT), если значение системного регистра такое же, как значение данных. Операции с битами данных в регистровом файле выполняются командами устройства сдвига.

Примеры:

BIT SET MODE2 0x00000070;

BIT TST ASTAT 0x00002000;

2. Непосредственная модификация регистра I с битреверсией или без битреверсии

Синтаксис:

a. MODIFY (Ia,<data32>);

MODIFY (Ic,<data24>) ;

b. BITREV (Ia,<data32>);

BITREV (Ic,<data24>) ;

Функция:

Модификация и обновление определенного регистра I 32-разрядным (DAG1) или 24-разрядным (DAG2) значением данных. Команда BITREV добавляет 32-разрядное значение к содержимому регистра индекса DAG1 (или 24-разрядное значение к содержимому регистра индекса DAG2), переставляет биты результата в обратном порядке и записывает полученный результат в тот же самый индексный регистр.

Примеры:

MODIFY (I4,304);

BITREV (I7,space); (space – определенная константа)

3. Нет операции (NOP)

Синтаксис:

NOP;

Функция:

Нулевая операция; увеличивается только адрес выборки.

4. IDLE

Синтаксис:

IDLE;

Функция:

Выполняется NOP, и процессор переводится в состояние низкой потребляемой мощности. Процессор остается в этом состоянии, пока не придет сигнал прерывания.

После возврата из программы обработки прерывания выполнение программы начинается с команды, следующей за IDLE.

Приложение Б. Справочник вычислительных операций

Вычислительные операции выполняются в умножителе, АЛУ и устройстве сдвига. В этом приложении подробно описывается каждая вычислительная операция, синтаксис ассемблера и код операции. Вычислительными операциями называются:

1) однофункциональная операция, выполняемая одним вычислительным устройством;

2) многофункциональная операция, которая определяет параллельное выполнение операций умножителем и ALU или выполнение двух операций ALU;

3) передача данных между регистрами MR, которая является специальным типом вычислительной операции, использующейся для обращения к аккумулятору с фиксированной точкой в умножителе.

Б.1. Операции ALU

Rn = Rx + Ry

Синтаксис:

Rn = Rx + Ry

Функция:

Сложение полей с фиксированной точкой (ПФТ) в регистрах Rx и Ry. Результат помещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. В режиме насыщения (бит режима насыщения ALU установлен в MODE1) при положительном переполнении в Rn возвращается максимальное положительное число (0x7FFF FFFF), а при отрицательном переполнении возвращается максимальное отрицательное число (0x8000 0000).

Rn = Rx – Ry

Синтаксис:

Rn = Rx – Ry

Функция:

Вычитается ПФТ регистра Ry из ПФТ регистра Rx. Результат помещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. В режиме насыщения (бит режима насыщения ALU установлен в MODE1) при положительном переполнении в Rn возвращается максимальное положительное число (0x7FFF FFFF), а при отрицательном переполнении возвращается максимальное отрицательное число (0x8000 0000).

Rn = (Rx + Ry)/2

Синтаксис:

Rn = (Rx + Ry)/2

Функция:

Сложение ПФТ регистров Rx и Ry и деление результата на 2. Результат помещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Округление результата к ближайшему или усечение (стандарт IEEE), в зависимости от установки бита режима округления в регистре MODE1.

COMP(Rx, Ry)

Синтаксис:

COMP(Rx, Ry)

Функция:

Сравнение ПФТ регистра Rx с ПФТ регистра Ry. Если два операнда равны, то устанавливается флаг AZ; если операнд в регистре Rx меньше, чем операнд в регистре Ry, то устанавливается флаг AN. В битах 24–31-го

регистра ASTAT хранятся результаты восьми предыдущих операций сравнения ALU. При выполнении команды сравнения с фиксированной или с плавающей точкой эти биты сдвигаются вправо (бит 24 теряется). Старший бит в ASTAT устанавливается, если X-операнд больше, чем Y-операнд (значение бита равно результату операции «И» между флагами AZ и AN); иначе − сброшен.

Флаги состояния:

AZ установлен, если операнды в регистре Rx и регистре Ry равны, иначе − сброшен;

AN установлен, если операнд в регистре Rx меньше, чем в операнд в регистре Ry, иначе − сброшен.

Rn = – Rx

Синтаксис:

Rn = – Rx

Функция:

Изменение знака операнда с фиксированной точкой в регистре Rx. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Изменение знака минимального отрицательного числа (0x8000 0000) приводит к переполнению. В режиме насыщения (бит режима насыщения ALU установлен в MODE1) при переполнении в Rn возвращается максимальное положительное число (0x7FFF FFFF).

Rn = ABS Rx

Синтаксис:

Rn = ABS Rx

Функция:

Определение абсолютного значения операнда с фиксированной точкой в Rx. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Определение абсолютного значения минимального

отрицательного числа (0x8000 0000) приводит к переполнению. В режиме насыщения (бит режима насыщения ALU установлен в MODE1) при переполнении в Rn возвращается максимальное положительное число (0x7FFF FFFF).

Rn = Rx AND Ry

Синтаксис:

Rn = Rx AND Ry

Функция:

Логическая операция «И» над операндами с фиксированной точкой из Rx и Ry. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули.

Флаги состояния:

AZ установлен, если результат с фиксированной точкой равен 0, иначе − сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Rn = Rx OR Ry

Синтаксис:

Rn = Rx OR Ry

Функция:

Логическая операция «ИЛИ» над операндами с фиксированной точкой из Ry. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули.

Флаги состояния:

AZ установлен, если результат с фиксированной точкой равен 0, иначе − сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Rn = Rx XOR Ry

Синтаксис:

Rn = Rx XOR Ry

Функция:

Логическая операция «исключающее ИЛИ» над операндами с фиксированной точкой из Rx и Ry. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. *Флаги состояния:*

AZ установлен, если результат с фиксированной точкой равен 0, иначе− сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Rn = NOT Rx

Синтаксис:

Rn = NOT Rx

Функция:

Поразрядное инвертирование операнда с фиксированной точкой из Rx. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули.

Флаги состояния:

AZ установлен, если результат с фиксированной точкой равен 0, иначе − сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Rn = MIN(Rx, Ry)

Синтаксис:

Rn = MIN(Rx, Ry)

Функция:

Возвращение меньшего из двух операндов с фиксированной точкой в Rx и Ry. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули.

Флаги состояния:

AZ установлен, если результат с фиксированной точкой равен 0, иначе − сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Rn = MAX(Rx, Ry)

Синтаксис:

Rn = MAX(Rx, Ry)

Функция:

Возвращение большего из двух операндов с фиксированной точкой из Rx и Ry. Результат помещается в ПФТ Rn. В оставшиеся биты ПФТ регистра Rn записываются нули.

Флаги состояния:

AZ установлен, если результат с фиксированной точкой равен 0, иначе − сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Rn = CLIP Rx BY Ry

Синтаксис:

Rn = CLIP Rx BY Ry

Функция:

Возвращение операнда с фиксированной точкой из Rx, если абсолютное значение этого операнда меньше абсолютного значения операнда с фиксированной точкой в Ry. В противном случае возвращается |Ry|, если значение Rx положительное, и –|Ry|, если Rx значение отрицательное. Результат помещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули.

Флаги состояния:

AZ установлен, если результат с фиксированной точкой равен 0, иначе − сброшен;

AN установлен, если старший бит результата равен 1, иначе − сброшен.

Для операторов с плавающей точкой справедливы все приведенные выше операции, только вместо R пишется F

Fn = Fx + Fy

Синтаксис:

Fn = Fx + Fy

Функция:

Сложение операндов с плавающей точкой из регистров Fx и Fy. Нормализованный результат помещается в регистр Fn. Округление к ближайшему или усечение до 32 бит или до 40 бит (в зависимости от установки битов режима округления и границы округления в MODE1). При переполнении после округления возвращается ±∞ (округление к ближайшему) или ±NORM.MAX (округление к нулю). После округления ненормализованного результата возвращается ±0. Ненормализованные входные операнды обнуляются. При вводе NAN возвращается NAN (1 во всех разрядах).

Fn = ABS (Fx + Fy)

Синтаксис:

Fn = ABS (Fx + Fy)

Функция:

Сложение операндов с плавающей точкой в регистрах Fx и Fy и помещение абсолютного значения нормализованного результата в регистр Fn. Округление к ближайшему или усечение до 32 бит или до 40 бит (в зависимости от установки битов режима округления и границы округления в MODE1). При переполнении после округления возвращается ±∞ (округление к ближайшему) или ±NORM.MAX (округление к нулю). После округления ненормализованного результата возвращается +0.

Ненормализованные операнды обнуляются. При вводе NAN возвращается NAN (1 во всех разрядах).

Fn = RND Fx

Синтаксис:

Fn = RND Fx

Функция:

Округление операнда с плавающей точкой из регистра Fx до 32 бит. Округление к ближайшему или усечение (в зависимости от установки битов режима округления и границы округления в MODE1). При переполнении после округления возвращается ±∞ (округление к ближайшему) или ±NORM.MAX (округление к нулю). После округления ненормализованного результата возвращается ±0. Ненормализованные входные операнды обнуляются. При вводе NAN возвращается NAN (1 во всех разрядах).

Rn = FIX Fx Rn = TRUNC Fx

Синтаксис:

Rn = FIX Fx Rn = TRUNC Fx Rn = FIX Fx BY Rn = TRUNC Fx BY Ry

Функция:

Преобразование операнда с плавающей точкой из Fx в 32-разрядное целое число с фиксированной точкой в виде двоичного дополнения. Если в регистре MODE1 бит TRUNC=1, то при выполнении операции FIX мантисса усекается к –∞. Если бит TRUNC=0, при выполнении операции FIX мантисса округляется к ближайшему целому. При выполнении операции TRUNC округление всегда к 0. Заметим, что бит TRUNC не влияет на операцию, соответствующую команде TRUNC. Если определен коэффициент масштабирования (Ry), то перед преобразованием значение из Ry (целое с фиксированной точкой в виде двоичного дополнения) добавляется к порядку операнда с плавающей точкой из Fx. Результат преобразования выравнивается по правому разряду (формат 32.0) в ПФТ регистра Rn.

Оставшиеся биты поля с плавающей точкой в Rn обнуляются. В режиме насыщения (бит режима насыщения ALU установлен в MODE1) при положительном переполнении и при результате, равном +∞, возвращается максимальное положительное число (0x7FFF FFFF), а при отрицательном переполнении и при результате, равном –∞, возвращается максимальное отрицательное число (0x8000 0000). При операции FIX происходит округление к ближайшему (IEEE) или усечение (в зависимости от состояния бита режима округления в MODE1). При вводе NAN возвращается NAN (1 во всех разрядах). Если режим насыщения не установлен, то при входном операнде, равном ∞, или при переполнении результата возвращаются единицы во всех разрядах ПФТ. При потере значащих разрядов положительным числом возвращается 0. При потере значащих разрядов отрицательным числом после его округления к ближайшему возвращается 0. При потере значащих разрядов отрицательным числом после его усечения возвращается –1 (0xFF FFFF FF00).

Флаги состояния:

AZ кстановлен, если результат с фиксированной точкой равен нулю, иначе − сброшен.

Fn = FLOAT Rx BY Ry Fn = FLOAT Rx

Синтаксис:

Fn = FLOAT Rx BY Ry Fn = FLOAT Rx

Функция:

Преобразование операнда с фиксированной точкой из Rx в результат с плавающей точкой. Если задан коэффициент масштабирования (Ry), то число из Ry (целое с фиксированной точкой в виде двоичного дополнения) прибавляется к порядку результата с плавающей точкой. Окончательный результат помещается в регистр Fn. Выполняется округление к ближайшему или усечение (в зависимости от заданного режима округления); граница округления всегда 40 бит, независимо от установки бита границы округления

в MODE1. Масштабное смещение порядка может вызывать переполнение и потерю значащих разрядов числа с плавающей точкой. При переполнении возвращается ±∞□ (округление к ближайшему) или NORM.MAX (округление к нулю); при потере значащих разрядов возвращается ±0.

Б.2. Операции умножителя

RnIMR = Rx * Ry

Синтаксис:

Rn = Rx * Ry *mod2*

MRF = Rx * Ry *mod2*

MRB = Rx * Ry *mod2*

Функция:

Перемножение полей с фиксированной точкой (ПФТ) из регистров Rx и Ry. Если округление определено (только для дробных данных), то результат округляется. Результат помещается или в ПФТ в регистр Rn, или в один из регистров накопления MR. Если определен Rn, то в него передается только часть результата, которая имеет одинаковый формат с входным операндом (биты 31-0 для целых, биты 63-32 для дробных чисел). В оставшиеся биты ПФТ регистра Rn записываются нули. Если определены MRF или MRB, то в них помещается весь 80-разрядный результат.

RnIMR = MR + Rx * Ry

Синтаксис:

Rn = MRF + Rx * Ry *mod2*

Rn = MRB + Rx * Ry *mod2*

MRF = MRF + Rx * Ry *mod2*

MRB = MRB + Rx * Ry *mod2*

Функция:

Умножение ПФТ из регистров Rx и Ry и сложение полученного значения со значением определенного регистра MR. Если округление определено (только для дробных данных), то результат округляется. Результат помещается или в ПФТ в регистр Rn, или в один из регистров накопления MR (в тот же самый регистр, из которого брался входной операнд). Если определен Rn, то в него передается только часть результата, которая имеет одинаковый формат с входным операндом (биты 31-0 для целых, биты 63-32 для дробных чисел). В оставшиеся биты ПФТ регистра Rn записываются нули. Если определены MRF или MRB, то в них помещается весь 80-разрядный результат.

Rn I MR = MR – Rx * Ry

Синтаксис:

Rn = MRF – Rx * Ry *mod2*

Rn = MRB – Rx * Ry *mod2*

MRF = MRF – Rx * Ry *mod2*

MRB = MRB – Rx * Ry *mod2*

Функция:

Умножение ПФТ из регистров Rx и Ry и вычитание результата из значения определенного регистра MR. Если округление определено (только для дробных данных), то результат округляется. Результат помещается или в ПФТ в регистр Rn, или в один из регистров накопления MR (в тот же самый регистр, из которого брался входной операнд). Если определен Rn, то в него передается только часть результата, которая имеет одинаковый формат с входным операндом (биты 31-0 для целых, биты 63-32 для дробных чисел). В оставшиеся биты ПФТ регистра Rn записываются нули. Если определены MRF или MRB, то в них помещается весь 80-разрядный результат.

RnIMR = SAT MR

Синтаксис:

Rn = SAT MRF *mod1*

Rn = SAT MRB *mod1*

MRF = SAT MRF *mod1*

MRB = SAT MRB *mod1*

Функция:

Если значение определенного регистра MR больше, чем максимальное значение, определяемое форматом данных, то умножитель установит результат в максимальное значение. В противном случае значение MR не изменяется. Результат помещается или в ПФТ в регистр Rn, или в один из регистров накопления MR (в тот же самый регистр, из которого брался входной операнд). Если определен Rn, то в него передается только часть результата, которая имеет одинаковый формат с входным операндом (биты 31-0 для целых, биты 63-32 для дробных чисел). В оставшиеся биты ПФТ регистра Rn записываются нули. Если определены MRF или MRB, то в них помещается весь 80-разрядный результат.

Флаги состояния:

MN установлен, если результат отрицательный, иначе − сброшен;

MV сброшен;

MU установлен, если все старшие 48 битов дробного результата равны 0 (знаковый или беззнаковый результат) или 1 (знаковый результат), а младшие 32 бита не все равны 0. Для целого результата нет потери значащих разрядов**.**

RnIMR = RND MR

Синтаксис:

Rn = RND MRF *mod1*

Rn = RND MRB *mod1*

MRF = RND MRF *mod1*

MRB = RND MRB *mod1*

Функция:

Округление значения определенного MR к ближайшему до 32 бит (до границы MR1-MR0). Результат помещается или в ПФТ в регистр Rn, или в один из регистров накопления MR (в тот же самый регистр, из которого брался входной операнд). Если определен Rn, то в него передается только часть результата, которая имеет одинаковый формат с входным операндом (биты 31-0 для целых, биты 63-32 для дробных чисел). В оставшиеся биты ПФТ регистра Rn записываются нули. Если определены MRF или MRB, то в них помещается весь 80-разрядный результат.

Флаги состояния:

MN установлен, если результат отрицательный, иначе – сброшен;

MV установлен, если не все старшие биты равны 0 (знаковый или беззнаковый результат) или 1 (знаковый результат).

Для операторов с плавающей точкой определены аналогичные операторы.

Б.3. Операции устройства сдвига

Rn = LSHIFT Rx BY Ry | <data8>

Синтаксис:

Rn = LSHIFT Rx BY Ry Rn = LSHIFT Rx BY <data8>

Функция:

Логический сдвиг операнда с фиксированной точкой в регистре Rx при помощи 32_разрядного значения в регистре Ry или 8-разрядного непосредственного значения данных в команде. Сдвинутый результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Значение сдвига – это число в виде двоичного дополнения. Положительное значение определяет сдвиг влево,

отрицательное – вправо. 8-разрядные непосредственные данные могут принимать значения от –128 до 127 включительно, что позволяет осуществлять полномасштабный сдвиг 32-разрядного поля.

Rn = Rn OR LSHIFT Rx BY Ry | <data8>

Синтаксис:

Rn = Rn OR LSHIFT Rx BY Ry Rn = Rn OR LSHIFT Rx BY <data8>

Функция:

Логический сдвиг операнда с фиксированной точкой в регистре Rx при помощи 32-разрядного значения в регистре Ry или 8-разрядного непосредственного значения данных в команде. Со сдвинутым результатом и ПФТ регистра Rn производится операция «ИЛИ», а результат записывается обратно в регистр Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Значение сдвига – это число в виде двоичного дополнения. Положительное значение определяет сдвиг влево, отрицательное – вправо. 8-разрядные непосредственные данные могут принимать значения от –128 до 127 включительно, что позволяет осуществлять полномасштабный сдвиг 32-разрядного поля.

Rn = ASHIFT Rx BY Ry | <data8>

Синтаксис:

Rn = ASHIFT Rx BY Ry Rn = ASHIFT Rx BY <data8>

Функция:

Арифметический сдвиг операнда с фиксированной точкой в регистре Rx при помощи 32-разрядного значения в регистре Ry или 8-разрядного непосредственного значения данных в команде. Результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Значение сдвига – это число в виде двоичного дополнения. Положительное значение определяет сдвиг влево, отрицательное – вправо. 8-разрядные непосредственные данные могут принимать значения от –128 до 127

включительно, что позволяет осуществлять полномасштабный сдвиг 32-разрядного поля.

Rn = ROT Rx BY Ry | <data8>

Синтаксис:

Rn = ROT Rx BY Ry Rn = ROT Rx BY <data8>

Функция:

Циклический сдвиг операнда с фиксированной точкой в регистре Rx при помощи 32_разрядного значения в регистре Ry или 8-разрядного непосредственного значения данных в команде. Результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Значение сдвига – это число в виде двоичного дополнения. Положительное значение определяет сдвиг влево, отрицательное – вправо. 8-разрядные непосредственные данные могут принимать значения от –128 до 127 включительно, что позволяет осуществлять полномасштабный сдвиг 32-разрядного поля**.**

Rn = BCLR Rx BY Ry | <data8>

Синтаксис:

Rn = BCLR Rx BY Ry Rn = BCLR Rx BY <data8>

Функция:

Обнуление бита операнда с фиксированной точкой в регистре Rx. Результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Положение бита определяется 32-разрядным значением в регистре Ry или 8-разрядным непосредственным значением данных в команде. 8-разрядные непосредственные данные могут принимать значения от 31 до 0 включительно, что позволяет обнулять любой бит внутри 32-разрядного поля. Если значение положения бита больше 31 или меньше 0, ни один бит не будет обнуляться.

Rn = BSET Rx BY Ry | <data8>

Синтаксис:

Rn = BSET Rx BY Ry Rn = BSET Rx BY <data8>

Функция:

Установка бита операнда с фиксированной точкой в регистре Rx. Результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Положение бита определяется 32-разрядным значением в регистре Ry или 8-разрядным непосредственным значением данных в команде. 8-разрядные непосредственные данные могут принимать значения от 31 до 0, включительно, что позволяет обнулять любой бит внутри 32-разрядного поля. Если значение положения бита больше 31 или меньше 0, ни один бит не будет обнуляться.

Rn = BTGL Rx BY Ry | <data8>

Синтаксис:

Rn = BTGL Rx BY Ry Rn = BTGL Rx BY <data8>

Функция:

Изменение состояние бита операнда с фиксированной точкой в регистре Rx. Результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Положение бита определяется 32_разрядным значением в регистре Ry или 8_разрядным непосредственным значением данных в команде. 8_разрядные непосредственные данные могут принимать значения от 31 до 0 включительно, что позволяет обнулять любой бит внутри 32_разрядного поля. Если значение положения бита больше 31 или меньше 0, ни один бит не будет обнуляться.

Флаги состояния:

SZ установлен, если результат устройства сдвига равен нулю, иначе − сброшен;

SV установлен, если входной операнд сдвинут влево больше чем на 0, иначе − сброшен.

BTST Rx BY Ry | <data8>

Синтаксис:

BTST Rx BY Ry BTST Rx BY <data8>

Функция:

Проверка бита операнда с фиксированной точкой в регистре Rx. Флаг SZ устанавливается, если бит равен 0, и сбрасывается, если бит равен 1. Результат размещается в ПФТ регистра Rn. В оставшиеся биты ПФТ регистра Rn записываются нули. Положение бита определяется 32-разрядным значением в регистре Ry или 8-разрядным непосредственным значением данных в команде. 8-разрядные непосредственные данные могут принимать значения от 31 до 0 включительно, что позволяет обнулять любой бит внутри 32-разрядного поля. Если значение положения бита больше 31 или меньше 0, ни один бит не будет обнуляться.

Флаги состояния:

SZ установлен, если проверяемый бит равен нулю или значение положения бита больше 31; сброшен, если проверяемый бит равен 1;

SV установлен, если входной операнд сдвинут влево больше, чем на 0, иначе − сброшен.

Rn = FDEP Rx BY Ry | <bit6>:<len6>

Синтаксис:

Rn = FDEP Rx BY Ry Rn = FDEP Rx BY <bit6>:<len6>

Функция:

Внесение поля из регистра Rx в регистр Rn. Входное поле выравнивается по правому разряду ПФТ регистра Rx. Длина поля определяется полем len6 в регистре Ry или непосредственным значением len6 в команде. Значение bit6 (в регистре Ry или в команде) определяет начальное положение бита для внесения поля в ПФТ Rn. Биты слева и справа от внесенного поля обнуляются. В расширенное ПФТ Rn (биты 7–0 40-

разрядного слова) записываются нули. Bit6 и len6 могут принимать значения между 0 и 63 включительно, что позволяет вносить поле длиной от 0 до 32 разрядов и располагать его в любом месте 32-разрядного слова.

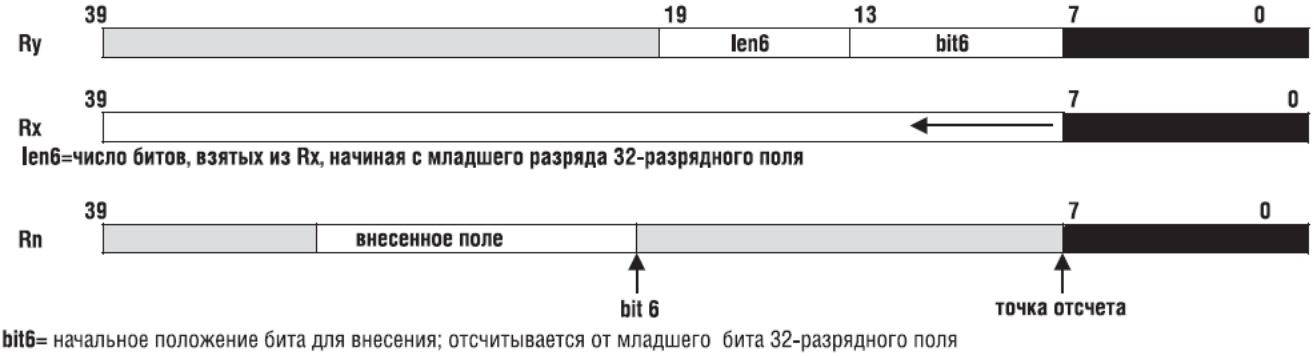
Пример:

Если len6=14 и bit6=13, тогда 14 битов Rx вносятся в биты Ry 34–21 (40- разрядного слова).

Флаги состояния:

SZ установлен, если выходной операнд равен 0, иначе − сброшен;

SV установлен, если какой-либо бит внесен слева от 32-разрядного выходного поля (т. е. если len6+bit6>32), иначе − сброшен.



bit6= начальное положение бита для внесения; отсчитывается от младшего бита 32-разрядного поля

Б.4. Многофункциональное вычисление

Многофункциональные вычисления бывают трех типов. Каждый тип имеет различный формат 23-разрядного вычислительного поля:

- Двойное сложение/вычитание.
- Параллельная операция умножителя и ALU.
- Параллельное умножение и сложение/вычитание.

При использовании ALU и умножителя для многофункциональных вычислений каждый из четырех входных операндов ограничен своим набором из четырех регистров регистрового файла (см. рис. Б.1). Например, вход X в ALU может быть только R8, R9, R10 или R11. Во всех других

операциях входной операнд может быть любым регистром регистрового файла.

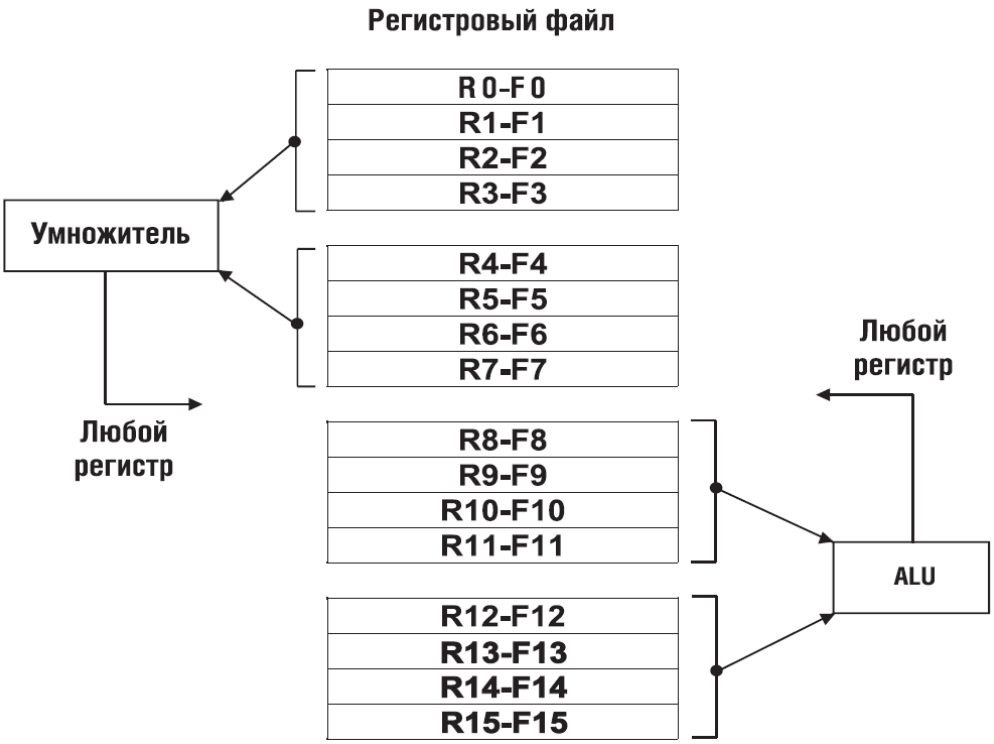


Рис. Б.1. Регистры входных данных, которые разрешено использовать для многофункциональных вычислений

Двойное сложение/вычитание

В операции двойного сложения/вычитания вычисляется сумма и разность двух входных операндов и возвращается два результата в различные регистры. Существуют два типа этой операции: с фиксированной и плавающей точкой.

С фиксированной точкой:

Синтаксис:

Ra = Rx + Ry, Rs = Rx – Ry

Функция:

Выполнение двойного сложения/вычитания ПФТ в регистрах Rx и Ry. Сумма помещается в ПФТ регистра Ra, разность – в регистр Rs. В расширенное ПФТ регистров Ra и Rs записываются нули. В режиме

насыщения (бит режима насыщения ALU установлен в MODE1) при положительном переполнении возвращается максимальное положительное число (0x7FFF FFFF), при отрицательном переполнении возвращается максимальное отрицательное число (0x8000 0000).

С плавающей точкой

Синтаксис:

Fa = Fx + Fy, Fs = Fx – Fy

Функция:

Выполнение двойного сложения/вычитания операндов с плавающей точкой в регистрах Fx и Fy. Нормализованные результаты помещаются в регистры Fa и Fs (сумма в Fa, разность в Fs). Округление к ближайшему или усечение до 32 бит или до 40 бит (в зависимости от установки битов режима округления и границы округления в MODE1). При переполнении после округления возвращается ±∞ (округление к ближайшему) или ±NORM.MAX (округление к нулю). После округления ненормализованного операнда возвращается ±0. Ненормализованные входные операнды обнуляются. При вводе NAN возвращается NAN (1 во всех разрядах).

Параллельная операция умножителя и ALU

В параллельной операции умножителя и ALU выполняется умножение или умножение/накопление и одна из следующих операций ALU: сложение, вычитание, усреднение, преобразование из формата с фиксированной точкой к формату с плавающей точкой и наоборот, операции с плавающей точкой ABS, MIN или MAX.

Операции умножителя и ALU определяются OPCODE. X-операнд и Y-операнд умножителя берутся из регистров данных RXM (FXM) и RYM (FYM). Операнд результата умножителя возвращается в регистр данных RM (FM). X-операнд и Y-операнд ALU берутся из регистров данных RXA (FXA) и RYA (FYA). Операнд результата ALU возвращается в регистр данных RA

(FA). Операнды результатов могут возвращаться в любой регистр регистрового файла. Каждый из четырех входных операндов ограничен определенным набором из четырех регистров данных.

Разрешенные источники

X умножителя: R3_R0 (F3_F0)

Y умножителя: R7_R4 (F7_F4)

X ALU: R11_R8 (F11_F8)

Y ALU: R15_R12 (F15_F12)

Синтаксис

Rm=R3_0 * R7_4 (SSFR), Ra=R11_8 + R15_12

Rm=R3_0 * R7_4 (SSFR), Ra=R11_8 _ R15_12

Rm=R3_0 * R7_4 (SSFR), Ra=(R11_8 + R15_12)/2

MRF=MRF + R3_0 * R7_4 (SSF), Ra=R11_8 + R15_12

MRF=MRF + R3_0 * R7_4 (SSF), Ra=R11_8 _ R15_12

MRF=MRF + R3_0 * R7_4 (SSF), Ra=(R11_8 + R15_12)/2

Rm=MRF + R3_0 * R7_4 (SSFR), Ra=R11_8 + R15_12

Rm=MRF + R3_0 * R7_4 (SSFR), Ra=R11_8 _ R15_12

Rm=MRF + R3_0 * R7_4 (SSFR), Ra=(R11_8 + R15_12)/2

MRF=MRF _ R3_0 * R7_4 (SSF), Ra=R11_8 + R15_12

MRF=MRF _ R3_0 * R7_4 (SSF), Ra=R11_8 _ R15_12

MRF=MRF _ R3_0 * R7_4 (SSF), Ra=(R11_8 + R15_12)/2

Rm=MRF _ R3_0 * R7_4 (SSFR), Ra=R11_8 + R15_12

Rm=MRF _ R3_0 * R7_4 (SSFR), Ra=R11_8 _ R15_12

Rm=MRF _ R3_0 * R7_4 (SSFR), Ra=(R11_8 + R15_12)/2

Fm=F3_0 * F7_4, Fa=F11_8 + F15_12

Fm=F3_0 * F7_4, Fa=F11_8 _ F15_12

Fm=F3_0 * F7_4, Fa=FLOAT R11_8 by R15_12

Fm=F3_0 * F7_4, Fa=FIX F11_8 by R15_12

Fm=F3_0 * F7_4, Fa=(F11_8 + F15_12)/2

Fm=F3_0 * F7_4, Fa=ABS F11_8

Fm=F3_0 * F7_4, Fa=MAX (F11_8, F15_12)

Fm=F3_0 * F7_4, Fa=MIN (F11_8, F15_12)

Параллельные операции умножения и двойного сложения/ вычитания

При параллельных операциях умножения и двойного сложения/вычитания выполняется умножение или умножение/накопление и вычисляется сумма и разность входных операндов ALU. Более подробно об операциях умножителя – в описании однофункциональных операций. Об операции двойного сложения/вычитания в разделе – двойное сложение/ вычитание.

С фиксированной точкой:

*Синтаксис**:***

Rm=R3_0 * R7_4 (SSFR), Ra=R11_8 + R15_12, Rs=R11_8 – R15_12

С плавающей точкой:

Синтаксис:

Fm=F3_0 * F7_4, Fa=F11_8 + F15_12, Fs=F11_8 – F15_12

X-операнд и Y-операнд умножителя берутся из регистров данных RXM (FXM) и RYM (FYM). Операнд результата умножителя возвращается в регистр данных RM (FM). X-операнд и Y-операнд ALU берутся из регистров данных RXA (FXA) и RYA (FYA). Операнд результата ALU возвращается в регистр данных RA (FA). Операнды результатов могут возвращаться в любой регистр регистрового файла. Каждый из четырех входных операндов ограничен определенным набором из четырех регистров данных.

Разрешенные источники

X умножителя: R3_R0 (F3_F0)

Y умножителя: R7_R4 (F7_F4)

X ALU: R11_R8 (F11_F8)

Y ALU: R15_R12 (F15_F12)

ЛИТЕРАТУРА

1. Ю.С.Юрченко. Программирование алгоритмов цифровой обработки сигналов на базе процессоров ADSP-21x6x SHARC. Учеб.Пособие. Спб.: Изд-во СПбГЭУ «ЛЭТИ», 2004. 80с.
2. Руководство пользователя по сигнальным процессорам семейства SHARC ADSP-2106x. Пер.с англ. А.В. Бархатов, А.А. Коновалов, М.Н. Петров. Санкт-Петербург. 2002. 684с.