Интегрированная среда разработки и отладки программ MC Studio. Инструменты ядра ELcore

Руководство программиста

Листов 51

Порядок использования данного документа

Настоящая документация охраняется действующим законодательством Российской Федерации об авторском праве и смежных правах, в частности, законом Российской Федерации «Об авторском праве и смежных правах». ГУП НПЦ «ЭЛВИС» является единственным правообладателем исключительных авторских прав на настоящую документацию.

Настоящую документацию, не иначе как по предварительному согласию ГУП НПЦ «ЭЛВИС», запрещается:

- воспроизводить, т.е. изготавливать один или более экземпляров настоящей документации, ее части, в любой форме, любым способом;
- сдавать в прокат;
- публично показывать, исполнять или сообщать для всеобщего сведения,
- переводить;
- переделывать или другим образом перерабатывать (дорабатывать).

ГУП НПЦ «ЭЛВИС» оставляет за собой право в любой момент вносить изменения (дополнения) в настоящую документацию без предварительного уведомления о таком изменении (дополнении).

ГУП НПЦ «ЭЛВИС» не несет ответственности за вред, причиненный при использовании настоящей документации.

Передача настоящей документации не означает передачи каких-либо авторских прав ГУП НПЦ «ЭЛВИС» на нее.

Возникновение каких-либо прав на материальный носитель, на котором передается настоящая документация, не влечет передачи каких-либо авторских прав на данную документацию.

Все указанные в настоящей документации товарные знаки принадлежат их владельцам.

Оглавление

1. Введение

В данной книге рассматриваются следующие инструменты DSP, применяемые для сборки, отладки и редактирования файлов проекта:
Ассемблер для ELCORE;
Компоновщик программ для ELCORE;
Библиотекарь;
Описание библиотеки ввода/вывода;
Анализатор/преобразователь форматов для объектных и исполняемых файлов.

2. Ассемблер для ELCORE

2.1. Введение

Ассемблер для ELCORE (**elcore-elvis-elf-as** или **elas**) предназначен для обработки исходных текстов на языке Ассемблера и создания объектных и исполняемых файлов, которые могут выполняться на DSP-ядре ELCORE.

Для реализации **Ассемблера** использованы источники GAS проекта binutils GNU.

2.2. Язык ассемблера

Язык Ассемблера содержит мнемонические коды для всех машинных команд DSP-ядра ELCORE. Кроме того, язык содержит мнемонические директивы, задающие дополнительные действия, которые должны быть выполнены ассемблером. В частности, они позволяют определять макрокоманды, которые осуществляют краткую запись группы команд с выполнением текстовой подстановки того или иного рода.

2.3. Запуск Ассемблера

Запуск ассемблера выполняется из командной строки. При этом нужно задать ключи работы программы и перечислить имена входных файлов. Ассемблер имеет следующие ключи:

- -a...;
- --defsym;
- --help;
- -I;
- -o;
- --gstabs;
- -D;
- -f;
- -J;
- -L;
- --statistics;
- --strip-local-absolute;
- --version;
- -W;
- --warn;
- --fatal-warnings;

- --itbl;
- -Z;
- --listing-lhs-width;
- --listing-lhs-width2;
- --listing-rhs-width;
- --listing-cont-lines;
- --mc11;
- --mcx2.

Пример запуска ассемблера с получением листинга: ***elas*** *-al xxx.s*

2.3.1. -a...

Ключ -a... указывает опции управления листингом:

- c - исключить области, которые относятся к отвергнутым при условном ассемблировании;
- d - пропустить опции отладки;
- l - добавить сгенерированный код;
- m - включить макрорасширения.

Параметры, следующие после **-a**, могут быть скомбинированы в один ключ. Например, **-alm**.

2.3.2. --defsym

Ключ --defsym SYM=VAL устанавливает значение символа **SYM** равным **VAL**. **VAL** должно быть константой.

2.3.3. --help

Ключ --help выводит информацию о ключах Ассемблера.

2.3.4. -I

Ключ -I dir добавляет директорию **dir** к списку поиска для поиска файлов, указанных в директиве .include. Вы можете использовать ключ **-I** столько раз, сколько нужно. Поиск файлов сначала осуществляется в рабочей директории, а затем в директориях, указанных ключом **-I** в порядке их задания (слева направо).

2.3.5. -o

Ключ -o OBJFILE указывает результирующий объектный файл (**OBJFILE**). По умолчанию elas пытается собрать файл *a.out*.

2.3.6. --gstabs

Ключ --gstabs указывает elas добавить к результирующему файлу отладочную информацию.

2.3.7. -D

Ключ -D указывает elas выводить отладочные сообщения по работе ассемблера.

2.3.8. -f

Ключ -f позволяет пропустить обработку комментариев и пробелов. Это приводит к тому, что elas не выполняет предварительной обработки символов-разделителей и комментариев в тексте при ассемблировании.

2.3.9. -J

Ключ -J отключает предупреждения при переполнении.

2.3.10. -L

Ключ -L (**--keep-locals**) указывает elas сохранять локальные символы (например, начинающиеся с '**L**'). Метки, начинающиеся с **L** (только верхний регистр), называются локальными метками. Обычно эти метки невидимы при отладке, потому что они предназначены для использования программами типа компиляторов, которые создают ассемблерный код. Обычно и elas, и ld опускают такие метки, так что у вас не получится нормальной отладки с ними. Эта опция говорит elas оставлять '**L...**'-символы в объектном файле. Вы также должны указать линковщику ld, чтобы он сохранял символы с именами, начинающимися на '**L**'.

2.3.11. --statistics

Ключ --statistics позволяет вывести различную статистику выполнения. Используется для вывода двух характеристик ресурсов, использованных elas: максимальный размер занятого пространства во время ассемблирования (в байтах) и общее время ассемблирования (время, которое процессор исполнял elas, в секундах).

2.3.12. --strip-local-absolute

Ключ --strip-local-absolute указывает elas удалить локальные абсолютные символы.

2.3.13. --version

Ключ --version позволяет вывести версию используемого ассемблера.

2.3.14. -W

Ключ -W (**--no-warn**) запрещает предупреждения (warnings).

2.3.15. --warn

Ключ --warn разрешает вывод предупреждений (warnings).

2.3.16. --fatal-warnings

Ключ --fatal-warnings указывает elas рассматривать предупреждения (warnings) как ошибки.

2.3.17. --itbl

Ключ --itbl INSTTBL позволяет расширить набор инструкций инструкциями, соответствующими спецификациям в файле **INSTTBL**.

2.3.18. -Z

Ключ -Z позволяет сгенерировать объектный файл даже при наличии ошибок.

2.3.19. --listing-lhs-width

Ключ --listing-lhs-width устанавливает ширину колонки в словах для листинга.

2.3.20. --listing-lhs-width2

Ключ --listing-lhs-width2 устанавливает ширину в словах линий продолжения.

2.3.21. --listing-rhs-width

Ключ --listing-rhs-width устанавливает максимальную длину строки исходных файлов.

2.3.22. --listing-cont-lines

Ключ --listing-cont-lines устанавливает максимальное число строк для вывода в листинге.

2.3.23. --mc11

Ключ --mc11 разрешает использовать только операции, регистры и способы адресации mc11.

2.3.24. --mcx2

Ключ --mcx2 разрешает использовать все операции, регистры и способы адресации mc02.

2.3.25. --mcx3

Ключ --mcx3 разрешает использовать все операции, регистры и способы адресации mcx3.

2.3.26. --mcx4

Ключ **–mcx4** разрешает использовать все операции, регистры и способы адресации mcx4.

2.3.27. --mcx26

Ключ **–mcx4** разрешает использовать все операции, регистры и способы адресации ЦПОС-2.

2.4. Работа Ассемблера

Ассемблер последовательно обрабатывает все строки файла. При этом сначала выполняются все директивы макроподстановки, а затем полученный результат ассемблируется. После обработки всего файла выполняется окончательная обработка выражений и те из них, которые не могут быть вычислены на этом этапе, остаются для компоновщика.

2.5. Определения

Память процессора ELCORE содержит 3 компоненты: область программ, и, условно, область данных (X) и область констант (Y). Ассемблер позволяет создать код только для первых двух областей памяти и использовать имена для ссылок на область констант. Так как область констант видима только через регистр *AT*, в который и надо загружать соответствующие ссылки, и не доступна иным способом, то достаточно введение имен для констант либо прямо через .set, либо через .struct.

3. Написание программ на языке Ассемблера

3.1. Введение

В этой главе рассматриваются принципы написания программ на языке Ассемблера:

- директивы Ассемблера;
- структурное программирование;
- управление памятью;
- синтаксис Ассемблера.

3.2. Формат исходного файла

Программы ассемблера состоят из последовательности исходных операторов. Маленькие и большие буквы считаются эквивалентными при записи мнемоник команд, директив, кодов условий и имен регистров, но отличаются во всех остальных случаях, т.е. при записи меток, символов и литерных строк.

3.3. Сообщения об ошибках и предупреждения

elas может выдавать **предупреждения** (warnings) и **сообщения об ошибках** в стандартный файл ошибок (обычно, терминал). Этого не должно происходить, когда компилятор запускает elas автоматически. **Предупреждения** делаются, предполагая, что elas может ассемблировать дефектную программу, а **сообщения об ошибках** выдаются при серьезных проблемах, которые прекращают **ассемблирование**.

Предупреждения имеют следующий формат:
имя_файла:*NNN*:**Текст Предупреждения**
(где *NNN* - номер строки).

Если было задано имя логического файла ([**.line**]), то он используется для вычисления выводимого номера, иначе выводится текущая строка обрабатываемого исходного файла.

Сообщения об ошибках имеют формат:
Имя_файла:*NNN*:**FATAL**:**Текст Сообщения Об Ошибке**
Имя файла и номер строки определяются так же, как и для предупреждения.

Для того чтобы elas обрабатывал **предупреждения** так же, как **сообщения об ошибках**, используется ключ --fatal-warnings.

3.4. Символы

Для записи **символа** можно использовать любую комбинацию букв латинского алфавита, цифры и знак подчеркивания. Первым символом имени должна быть буква. Большие и малые буквы в символьном имени считаются различными. Все символы являются значащими, т.е. ограничений на длину имени не накладывается.

Можно использовать локальные цифровые метки. Одна и та же метка может встречаться много раз. При этом ссылка на следующую/предыдущую по тексту метку, например, "*1*" делается следующим образом: *1f* и *1b*.

3.5. Литерные константы

Для использования в программе на Ассемблере символов *ASCII* следует использовать константы в виде *'символ*. Данная комбинация заменяется на код символа и может быть использована в выражениях.

При необходимости сформировать в памяти строку ее следует записывать в кавычках. При этом используется нотация С для вставки спецсимволов:

- **** - вставка самого \;
- **\b** - возврат на позицию (код 010);
- **\f** - перевод страницы (014);
- **\n** - перевод строки (012);
- **\r** - перевод каретки (015);
- **\t** - табуляция (011);
- **** цифра цифра цифра - задание кода в восьмеричном формате;
- **\x** цифра цифра - задание числа двумя шестнадцатеричными цифрами;
- **\"** - вставка кавычки.

3.6. Формат исходного оператора

3.6.1. Формат исходного оператора

Каждый **исходный оператор** может содержать до 9 полей: поле метки, поле первой операции, операнды, поле второй операции, операнды второй операции, первая команда пересылки, параметры команды пересылки, команда чтения констант и параметры команды чтения констант. Поля отделяются друг от друга каким-то числом пробелов или символов табуляции. При этом строка может быть ограничена символом комментария **";"** и в этом случае ее конец игнорируется. Кроме того, можно использовать С-стиль написания комментариев.

Поля не должны включать в себя пробелы и символы табуляции.

3.6.2. Поле метки

Поле метки является первым в команде. Если метка начинается не с первой позиции, то она должна заканчиваться двоеточием.

3.6.3. Поле первой операции

Поле операции отделяется от поля метки или начала строки как минимум одним пробелом или символом табуляции. В поле операции может использоваться:

- мнемоническое имя команды;
- директива ассемблирования;
- вызов макроса.

При использовании мнемонического имени команды или имени макроса возможно указание модификатора. Модификатор отделяется от самой команды/макроса точкой и используется для указания условий выполнения или размера операндов. Например, *absl.ne r0,R4*

При программировании параллельных команд (*формат 8*) в поле операции первой указывается команда из группы *OP2*.

3.6.4. Поле операндов

Поле операндов представляет собой набор аргументов для операции. Аргументы отделяются друг от друга запятой. В качестве аргументов можно использовать:

- **регистры общего назначения** *R0-R31*. Вместе с ними для исключения неоднозначности можно использовать модификатор размера "*.L*", например, *r4.L*, *R0.L*, *R8.L* и т.п. Используемый размер операнда обычно определяется операцией. Фактически указание размера операнда необходимо только для межрегистровых пересылок;
- **адресные и управляющие регистры** (т.к. все они 16-битные, то никаких модификаторов использовать нельзя). Например, *A5, LA*;
- **адреса с А-регистрами**, например, *(A1), (A2)+, (A3)-, (A0)+I0, (A2)-I2, (A3+I3)*;
- **адрес А+смещение** в форме (*An+выражение*) или (*An-выражение*);
- выражение;
- **#выражение**.

Ассемблер не делает различий между записями **#выражение** и **выражение**. Исключением являются те случаи, когда необходимо четко указать, что в данном месте должно быть абсолютное или относительное выражение в командах перехода и вызова функции. Символ **#** указывает на использование абсолютного выражения. Например:

```
 118  01b4 002AA98A     dofor  #q1
```
- операция использует абсолютный адрес (*DOFOR*)

```
 119  01b5 001D018B     dofor  q1
```
- операция использует относительный адрес (*DOFORR*)

3.6.5. Поле второй операции

Поле второй операции используется для задания операции из группы *OP1* операций *8-ого формата* DSP-ядра. При этом в поле первой операции должна быть записана одна из операций *OP2 8-го формата*. Допустимо опускать поле второй операции вместе с параметрами. В этом случае, например, при наличии двух пересылок, будет использован *8-ой формат* с операцией *NOP* вместо *OP1*.

3.6.6. Поле операндов второй операции

Так как вторая операция должна удовлетворять требованиям *8-го формата*, то в **поле операндов второй операции** возможно указывать только регистры *R0..R31*.

3.6.7. Команда пересылки

При задании **пересылки** можно не указывать самого кода команды. Так как DSP-процессор всегда использует *32*-битные операции для работы с памятью и только при работе с регистрами имеется выбор *16/32* бита, то в большинстве случаев в специальном задании команды нет необходимости. В качестве кода команды можно использовать либо *M*, либо *ML*. Соответственно, *ML* используется для явного указания *32*-битного размера данных. Это может быть необходимо только для пересылок между *R*-регистрами. Для указания размера можно также явно сослаться на размер регистра, например, записи *ML R0,R4* и *R0,R4.L* приводят к идентичному результату.

Пример использования параллельной пересылки:

```
99 0128 110E077A   >>> btstl r2,r4 #12,a7
99           0000000C
99 012a 110C877A   >>> btstl r2,r4 #12,r6.1
99           0000000C
```

Также при пересылках возможно использование различных параметров.

3.6.8. Параметры команды пересылки

В поле **параметров команды пересылки** возможны следующие комбинации:

- копирование обычных и управляющих регистров;
- чтение/запись регистра через An-адресацию;
- условное копирование константы в регистр;
- изменение A-регистра.

3.6.9. Команда чтения константы

Команда чтения константы осуществляет, используя значения *AT* и соответствующие виды адресации, например, *(AT)*, *(AT+IT)*, чтение константы в регистр *R0*. Команда может быть либо представлена в виде *M*, *ML*, либо опущена. С **командой чтения константы** следует использовать параметры команды чтения константы.

3.6.10. Параметры команды чтения константы

Команда чтения констант всегда имеет два **параметра**. Первый из них представляется в виде *AT*-адреса: *(AT)*, *(AT)+IT*, *(AT+IT)*. Второй всегда *R0*.

В связи с тем, что Ассемблер не может проверить выбранный режим статусного регистра, а код в команде для *(AT+IT)* и *(AT)+DT* идентичен, то исключительно для контроля и наглядности можно использовать директиву .mcaddr:

- ***.mcaddr** both* – допускаются оба варианта написания (по умолчанию);
- ***.mcaddr** dt* – режим с использованием *DT*-адреса, *(AT+IT)* запрещен;
- ***.mcaddr** it* – режим с использованием *IT*-адреса, *(AT)+DT* запрещен.

Замечание: **Дисассемблер** всегда выводит вариант *(AT+IT)*.

3.6.11. Поле комментария

Комментарии игнорируются ассемблером и могут использоваться для повышения читабельности кода. **Комментарии** начинаются с символа **;**. Все, что следует за этим символом на данной строке, считается **комментарием**. Также можно использовать С-стиль написания **комментариев**.

3.7. Результат ассемблирования

В результате работы ассемблера выдается необязательный листинг и объектный/исполняемый код программы. DSP-ядро использует для работы режим *LSB*, т.е. младший байт - первый. В соответствии с этим действует и Ассемблер. Однако, для более естественного представления данных в листинге используется обратный порядок байт. В результате при отображении данных с размером менее *32* бит происходит перестановка данных. Например:

Естественное представление данных:

```
 23 0021 40000000      .float  0.5, 0.25, 0.125
 23      20000000
 23      10000000
```

Изменение порядка данных:

```
 14 0014 20004000      .single 0.5,0.25,-0.5,-0.25
 14      E000C000
```

В данном случае:

Второй байт 0.5
| Первый байт 0.5
20 00 40 00
| Третий байт принадлежит 0.25
Четвертый байт принадлежит 0.25

3.8. Выражения

3.8.1. Введение

Выражение представляет собой величину, которая может быть использована вместо операнда в **команде** или директиве. При этом вычисленная величина должна быть ограничена в соответствии с требованием формата **команды** или директивы. Промежуточные результаты в общем случае могут выходить за пределы этих ограничений.

3.8.2. Использование выражений

Выражения, которые могут быть вычислены на этапе работы Ассемблера, явным образом вставляются в код. Если выражение за счет наличия адресов не может быть вычислено немедленно, то его вычисление будет перенесено на этап сборки. Недопустимо использовать в одном выражении адреса из разных областей памяти.

3.8.3. Запись числовых констант в выражениях и операндах

Числовые константы, которые встречаются непосредственно в виде операндов или являются частью выражений следует записывать следующим образом:

- для целочисленных констант можно использовать различные системы счисления, при этом возможны следующие способы указания системы счисления:
  - *0* с одним из символов *oOqQ* указывает на восьмеричную систему счисления (например, *0q27723577*);
  - *0* с символом *hHxX* указывает на шестнадцатеричную систему счисления (например, *0xFFFF*);
  - *0* с символом *B* указывает на двоичную систему счисления (например, *0B100011011101*);

- по умолчанию целые числа, которые начинаются с нуля, тоже относятся к восьмеричным;
- остальные числа по умолчанию считаются десятичными.

- вещественная константа имеет следующий формат: *+/- целая_часть [.дробная_часть] [E/e] [+/-] экспонента*, причем либо разделитель *дробной части* "**.**", либо *экспонента* должны присутствовать.

3.8.4. Запись специальных DSP-констант в операндах

DSP-ядро поддерживает ряд специальных форматов с фиксированной точкой. Кроме того, предполагается использование форматов с программной плавающей точкой. Для того чтобы обеспечить возможность использования их в ассемблерном коде, введена специальная операция "**[ ]**". Аргументы для этой операции должны быть определены. Не допускается использование неопределенных имен. Имеются следующие форматы записи операции:

- **[целое выражение]** - эквивалентно записи *(выражение)*;
- **[старшее полуслово, младшее полуслово]** - позволяет задать 32-битное слово из двух половинок, в качестве полуслова можно использовать либо целочисленное выражение, либо вещественную константу в диапазоне *(-1.,1.);*
- **[константа с плавающей точкой]** - биты *мантиссы* (*32*) для константы в форме программной плавающей точки;
- **[^константа с плавающей точкой]** - биты *экспоненты* (*16*) для константы в форме программной плавающей точки;
- **[@старший байт, младший байт]** - задание *16*-битной константы из двух *8*-битных половин.

Форматы данных DSP-ядра:

№	Описание формата	Непосредственный операнд	Константа в памяти DSP
1	Целый 16-разрядный	*# -32767*	*.dw –32767*
2	Целый 32-разрядный	*# -32768*32767*	*.dl –0xFFFFFFFF*
3	Целый комплексный (16 разр., 16 разр.)	*# [-32767,-0x8000]*	*.dw –32767, -0x8000*
4	Дробный 16-разрядный	*# -0.875*	*.fr –1.0*
5	Дробный 32-разрядный	*# -0.875*	*.frl –0.875*
6	Дробный комплексный (16 разр., 16 разр.)	*# [-0.5,-0.375]*	*.single –0.5, -0.375*
7	Программная плавающая точка	*-31.25e-1*	*.double –31.25e-1*
8	Плавающая точка (32 бита)	*#2.5*	*.real -3.7e6*

Пояснения:

1. Значение числа в дробном 16-разрядном формате равно:

$$B_{14}*2^{-1} + B_{13}*2^{-2} + \ldots + B_0*2^{-14}.$$

Код – дополнительный, бит B_0 – младший. Выражение для *32*–разрядного формата аналогично.

2. Значение числа с плавающей точкой равно:

$$2^{\pm E} * (\pm F),$$ где:

E – *экспонента* в *16*-разрядном дополнительном коде, | *E* | *<16384*,
F – *мантисса* в *32*-разрядном дополнительном коде.

Примеры использования:

```
1 0000 00000000    addl #0.5,R2,R0
1           40000000
2 0002 07FCE184    add  #-0.25,R0
6 0008 00000000    .double 0.5
6          40000000
```

Макро для присвоения значения константы с плавающей программной точкой 3-м регистрам.

```
9                  .macro  movfi,v,a,b
10                 move #[^\v],\a
11                 move #[\v],\b.L
12                 .endm
```

Использование макро:

```
13                movfi   0.25,r1,r2
13 0010 00028700   > move #[^0.25],r1
13      0000FFFF
13 0012 00058700   > move #[0.25],r2.L
13      40000000
14 0014 20004000   .fr 0.5,0.25,-0.5,-0.25
14  14 E000C000
23 0021 40000000   .frl  0.5, 0.25, 0.125
23      20000000
23      10000000
```

Для отличия *32*-битных значений с плавающей точкой от значений с фиксированной точкой в командах следует использовать псевдокоманды .ffloat и .ffix. По умолчанию работает режим с фиксированной точкой.

3.8.5. Операторы

При написании целочисленных выражений допустимо использовать следующие **операторы**:

- **(выражение)** - объединение элементов выражения;
- **||**, **&&** - логические операции *ИЛИ* и *И*;

- **==**, **<>**, **<**, **<=**, **>=** , **>** - операции сравнения;
- **+**, **-** - операции сложения и вычитания;
- **&**, **^**, **~**, **|**, **!** - побитовые операции *И*, *ИСКЛЮЧАЮЩЕЕ ИЛИ*, *НЕ*, *ИЛИ* и логическое *НЕ*;
- *****, */*, **%**, **<<**, **>>** - операции умножения, деления, получения остатка, сдвига вправо и влево;
- **унарный -** и **+**.

Приоритет выполнения операций соответствует вышеприведенному порядку. Например, оператор "**!**" имеет более высокий приоритет, чем операция сравнения.

Ассемблер не включает выражений с участием значений с плавающей точкой, за исключением специальных DSP-констант.

3.8.6. Символ "."

При записи выражений можно использовать символ "**.**". Этот символ обозначает текущую позицию.

3.9. Управление размещением данных в памяти

3.9.1. Введение

По умолчанию elas размещает секции программы и данных, начиная с нулевых адресов соответствующих областей памяти. Тем не менее, пользователь может изменять адреса размещения секций и управлять выделением памяти.

3.9.2. Секции

Ассемблер использует следующие **секции** для размещения программы и данных:

- **text** - размещение кода;
- **data** - размещение данных;
- **stab** - секция отладочной информации, генерируется автоматически по запросу;
- **stabstr** - таблица имен для отладочной информации, генерируется автоматически.

В **директивах указания секций** можно явно заказать подсекцию номером в диапазоне *0-8192*. Все, что попадет при работе Ассемблера в одну подсекцию, окажется в одной области памяти:

```
.text 1
код подсекции 1
.text 2
код подсекции 2
```

```
.text 1
продолжение кода подсекции 1
```

В данном случае подсекция *2* окажется вне кода подсекции *1*.

В отличие от ряда других Ассемблеров, допускается возможность резервирования места в памяти без размещения данных или кода. Это осуществляется посредством директив .comm/.lcomm.

3.9.3. Выделение места

Для **выделения места** в памяти, начиная с текущей позиции, следует использовать директивы .space и .skip.

3.10. Макроопределения и условное ассемблирование

3.10.1. Макроопределения

Для упрощения программирования Ассемблер позволяет использовать **макроопределения**.
Используется следующий синтаксис задания **макроопределения**:

Заголовок макроопределения (.macro)
Тело макроопределения
.endm

Для заголовка **макроопределения** используется директива .macro. При этом задание имени **макроопределения** и параметров может быть выполнено одним из следующих способов:

- Имя_макро .macro параметры
- .**macro** Имя_макро параметры
- .**macro** Имя_макро(параметры)

Параметры могут отделяться друг от друга запятой или пробелами и включать задание значения по умолчанию:

```
4           .macro abc x,y=2 z=7
5           .dl  \x
6           .dl  \z-\y
7           .endm
```

При вызове **макроопределения** можно не указывать все параметры, при этом пропущенные параметры будут либо "пустыми", либо иметь соответствующее значение по умолчанию.

Как видно из вышеприведенного примера, для выполнения подстановки параметров следует использовать обращение следующего вида: *\имя_параметра*. Возможна также ссылка на параметр в форме *&имя_параметра*.

При вызове **макроопределения** возможно указание модификатора, для ссылки на модификатор в теле **макроопределения** следует использовать \0, при этом точка не входит в значение параметра:

```
.macro test
b.\0  next
.endm
test.eq
```

Если в теле **макроопределения** необходима безусловная вставка некоторого текста, которые содержит обрабатываемые символы, то их можно защитить от обработки при помощи заключения в конструкцию следующего вида: *\(текст)*.

Для преждевременного выхода из **макроопределения** следует использовать директиву .exitm. Для удаления **макроопределения** - директиву .purgem.

Перед стартом процесса расширения макро осуществляется преобразование параметров. В частности, отбрасываются окружающие кавычки.

3.10.2. Условное ассемблирование

Условное ассемблирование позволяет генерировать код в зависимости от каких-либо условий. В частности, этот механизм может быть использован для вложенных макроопределений. Пример:

```
9                 zzz  .macro  f
10                   .dl 12-\f
11                   .ifge  \f
12                   zzz "(\f-1)"
13                   .endif
14                   .endm
15                   zzz 1
15 0002 0000000B     > .dl 12-1
15                 > .ifge 1
15                 > zzz "(1-1)"
15 0003 0000000C     >> .dl 12-(1-1)
15                 >> .ifge (1-1)
15                 >> zzz "((1-1)-1)"
15 0004 0000000D     >>> .dl 12-((1-1)-1)
15                 >>> .ifge ((1-1)-1)
15                 >>> zzz "(((1-1)-1)-1)"
15                 >>> .endif
15                 >> .endif
15                 > .endif
```

Данный пример был получен при компиляции с ключами -alm. Эта комбинация ключей позволяет полностью проверить процедуру макрорасширения. Соответственно, символами ">" в листинге указан уровень вложенности макрорасширения.

Имеются следующие директивы **условного ассемблирования**:

- .if условие — проверка на неравенство нулю
- **.ifeq** выражение — проверка на равенство нулю
- **.ifge** выражение — проверка на больше или равно нулю
- **.ifgt** выражение — проверка на больше нуля
- **.ifle** выражение — проверка на меньше или равно нулю
- **.iflt** выражение — проверка на меньше нуля
- **.ifne** выражение — проверка на неравенство нулю
- .ifdef имя — проверить определенность имени

- **.ifndef** имя — проверить неопределенность имени
- .ifnotdef имя — проверить неопределенность имени
- **.ifc** строка1,строка2 — проверить строки на несовпадение
- **.ifnc** строка1,строка2 — проверить строки на несовпадение
- **.ifeqs** строка1,строка2 — проверить С-строки на несовпадение
- **.ifnes** строка1,строка2 — проверить С-строки на несовпадение
- .else — часть "иначе"
- .elseif условие — альтернативное условие
- .endif — конец условия

Под С-строкой здесь понимается строка в кавычках.

3.10.3. Циклы

Циклы предназначены для повторения определенного набора операторов несколько раз и - возможно - с разными параметрами. Существуют следующие директивы для оформления **циклов**:

- .irp;
- .irpc;
- .rept.

Конец **цикла** для всех трех директив задается директивой .endr.

3.11. Директивы ассемблера

Директивы ассемблера управляют режимами его работы, обеспечивают выделение памяти и выравнивание на определенной границе, генерацию отладочной информации и т.п. Существуют следующие **директивы ассемблера DSP**:

- .abort;
- .align;
- .ascii;
- .asciz;
- .balign;
- .byte;
- .comm;

- .data;
- .dl;
- .double;
- .dw;
- .else;
- .elseif;
- .end;
- .endif;
- .endm;
- .endr;
- .equ;
- .equiv;
- .err;
- .exitm;
- .extern;
- .fail;
- .ffix;
- .ffloat;
- .fill;
- .float;
- .fr;
- .frl;
- .global;
- .hword;
- .if;
- .ifdef;
- .ifnotdef;
- .include;
- .int;
- .irp;
- .irpc;
- .lcomm;
- .long;
- .macro;
- .mcaddr;
- .octa;
- .print;
- .psize;
- .purgem;
- .real;
- .reg;
- .rept;
- .scalar;

- .set;
- .short;
- .simd;
- .single;
- .skip;
- .space;
- .struct;
- .text;
- .title;
- .word.

Кроме перечисленных **директив**, в elas включены директивы, зарезервированные для отладки высокоуровневых языков.

3.11.1. .abort

Директива .abort немедленно прекращает работу Ассемблера. Вы можете использовать **.abort**, например, при обнаружении некорректности каких-либо параметров.

3.11.2. .align

Директива .align x1,x2,x3 осуществляет выравнивание позиции данных. Выражение **x1** определяет границу выравнивания (в байтах). Выражение **x2** (если задано) указывает значение байта для заполнения, а выражение **x3** (если задано) - задает ограничение на количество байт для вставки.

3.11.3. .ascii

Директива .ascii Str позволяет разместить в памяти строку символов (**Str**) без вставки нуля в конце.

3.11.4. .asciz

Директива .asciz Str размещает в памяти строку символов **Str** с вставкой гарантированного нуля в конце.

3.11.5. .balign

Директивы .balignw и .balignl осуществляют выравнивание границы памяти и используют, соответственно, *16* и *32*-битные заполнители. Синтаксис директив полностью аналогичен синтаксису директивы .align.

3.11.6. .byte

Директива .byte X позволяет расположить в памяти байт со значением **X**. Следует помнить, что для ELCORE единицей адресации является *32*-битное слово.

3.11.7. .comm

Директива .comm Name,Size создает в области данных общий для различных модулей глобальный символ **Name** размера **Size** (в байтах). Фактическое выделение места происходит на этапе сборки.

3.11.8. .data

Директива .data N указывает, что elas должен ассемблировать все последующие операторы в конец подсекции *data* с номером **N**. Если номер не указан, по умолчанию используется подсекция *0*.

3.11.9. .dl

Директива .dl X размещает в памяти целое *32*-разрядное число **X**.
Подробнее об этом см. Форматы данных DSP-ядра.

3.11.10. .double

Директива .double X размещает в памяти *64*-битное число **X** с плавающей точкой (программное представление).
Подробнее об этом см. Форматы данных DSP-ядра.

3.11.11. .dw

Директива .dw X размещает в памяти целое *16*-битное число **X**.
Подробнее об этом см. Форматы данных DSP-ядра.

3.11.12. .else

Директива .else указывает elas начало блока кода, ассемблируемого условно в том случае, если условие, указанное в предыдущей директиве .if было ложным.
Подробнее об этом см. Условное ассемблирование.

3.11.13. .elseif

Директива .elseif C указывает elas на начало блока, ассемблируемого только условно в случае, если условие, указанное в предыдущей директиве .if ложно, а альтернативное условие **C** - истинно.
Подробнее об этом см. Условное ассемблирование.

3.11.14. .end

Директива .end указывает elas на конец программы. Все содержимое файла после **.end** игнорируется. В конце файла с кодом обязательно должна присутствовать директива **.end**.

3.11.15. .endif

Директива .endif означает конец блока кода, ассемблируемого условно. Подробнее об этом см. Условное ассемблирование.

3.11.16. .endm

Директива .endm означает конец блока кода, задающего макроопределение и начинающегося последней директивой .macro.

3.11.17. .endr

Директива .endr означает конец цикла, оформленного директивами .irp, .irpc или .rept.

3.11.18. .equ

Директива .equ Symbol,X устанавливает значение символа **Symbol** равным **X**. Возможно многократное переопределение значений одного и того же символа.

3.11.19. .equiv

Директива .equiv Symbol,X проверяет, был ли определен ранее символ **Symbol** и, если символ не определен, устанавливает его значение равным **X**.

3.11.20. .err

Директива .err mssg выводит диагностическое сообщение **mssg** и отменяет генерацию объектного файла.

3.11.21. .exitm

Директива .exitm осуществляет преждевременный выход из тела макроопределения.

3.11.22. .extern

Директива .extern Symbol объявляет **Symbol** как внешний глобальный символ.

3.11.23. .fail

Директива .fail X проверяет, меньше ли значение **X**, чем *500*. Данная директива используется, например, для проверки уровня вложенности макроопределения.

3.11.24. .ffix

Директива .ffix предполагает генерацию констант в формате с фиксированной точкой. Директива предназначена для управления генерацией кода команд при использовании аргументов с плавающей точкой.

3.11.25. .ffloat

Директива .ffloat предполагает генерацию констант в формате *32*-битного числа с плавающей точкой. Директива предназначена для управления генерацией кода команд при использовании аргументов с плавающей точкой.

3.11.26. .fill

Директива .fill Number,Size,X осуществляет заполнение области памяти байтами количества **Size**, повторенными **Number** раз и имеющими значение **X**.

3.11.27. .float

Директива .float X размещает в памяти *32*-битное число **X** в формате с фиксированной точкой.

3.11.28. .fr

Директива .fr X размещает в памяти дробное *16*-разрядное число **X**.
Подробнее об этом см. Форматы данных DSP-ядра.

3.11.29. .frl

Директива .frl X размещает в памяти дробное *32*-разрядное число **X**.
Подробнее об этом см. Форматы данных DSP-ядра.

3.11.30. .global

Директива .global Symbol (**.globl Symbol**) объявляет символ **Symbol** как глобальный и делает символ видимым за пределами модуля.

3.11.31. .hword

Директива .hword X располагает в памяти целое *16*-битное число **X**.

3.11.32. .if

Директива .if C объявляет начало блока, ассемблируемого только в том случае, если условие **C** истинно. Подробнее об этом см. Условное ассемблирование.

3.11.33. .ifdef

Директива .ifdef Symbol объявляет начало блока, ассемблируемого только в том случае, если символ **Symbol** был определен. Подробнее об этом см. Условное ассемблирование.

3.11.34. .ifnotdef

Директива .ifnotdef Symbol (**.ifndef Symbol**) объявляет начало блока, ассемблируемого только в том случае, если символ **Symbol** не был определен. Подробнее об этом см. Условное ассемблирование.

3.11.35. .include

Директива .include File включает в ассемблерный файл содержимое файла **File**. Список директорий для поиска файла может быть задан ключом -I.

3.11.36. .int

Директива .int X размещает в памяти целое *32*-битное число **X**.

3.11.37. .irp

Директива .irp Parameter, X1, X2,..,XN выполняет блок операторов **N** раз, последовательно придавая символу **Parameter** значения **X1..XN**. Блок операторов начинается с **.irp** и заканчивается директивой .endr. Для обращения к значению символа **Parameter** в теле цикла следует использовать **\Parameter**. Например, ассемблирование

```
.irp param,1,2,3
move r\param, 10
.endr
```

ассемблируется как

```
move r1,10
move r2,10
move r3,10
```

Если после символа не указано никаких значений, блок операторов ассемблируется один раз с символом, установленным в нулевую строку.

3.11.38. .irpc

Директива .irpc Parameter, X выполняет блок операторов **N** раз, последовательно придавая символу **Parameter** значения каждого знака **X**. Блок операторов начинается с **.irpc** и заканчивается директивой .endr. Для обращения к значению символа **Parameter** в теле цикла следует использовать **\Parameter**. Например, ассемблирование

```
.irpc param,123
move r\param, 10
.endr
```

ассемблируется как

```
move r1,10
move r2,10
move r3,10
```

Если после символа не указано никаких значений, блок операторов ассемблируется один раз с символом, установленным в нулевую строку.

3.11.39. .lcomm

Директива .lcomm Name,Size создает в области данных символ **Name** размера **Size** (в байтах). Фактическое выделение места происходит на этапе сборки. В отличие от директивы .comm, символ **Symbol** не становится глобальным.

3.11.40. .long

Директива .long X размещает в памяти целое *32*-битное число **X**.

3.11.41. .macro

Директива .macro Name,p1,..,pN задает макроопределение с именем **Name** и аргументами **p1**,..,**pN**. Блок макроопределения должен заканчиваться директивой .endm.

3.11.42. .mcaddr

Директива .mcaddr both/it/dt устанавливает режим адресации.
Подробнее см. Параметры команды чтения константы.

3.11.43. .octa

Директива .octa X размещает в памяти целое *16*-битное число **X**.

3.11.44. .print

Директива .print A выводит аргумент **A** во время ассемблирования.

3.11.45. .psize

Директива .psize Rows,Columns устанавливает размер страницы листинга. Количество строк принимает значение **Rows**, а количество столбцов - **Columns**. По умолчанию генерируется *60* строк по *200* позиций.

3.11.46. .purgem

Директива .purgem позволяет удалить макроопределение.

3.11.47. .real

Директива .real X размещает в памяти *32*-битное число **X** в формате с плавающей точкой.

3.11.48. .reg

Директива **.reg Symbol,Register** позволяет использовать для символ **Symbol** в качестве значения регистр **Register**. Символ, определенный таким образом, нельзя использовать в *A*-адресации.

3.11.49. .rept

Директива .rept Number осуществляет циклическое выполнение операторов, заключенных между **.rept** и .endr, **Number** раз.

3.11.50. .scalar

Директива .scalar указывает, что при генерации области данных следует предполагать скалярный режим работы DSP.

Примечание: директивы .simd и **.scalar** актуальны только для DSP *Elcore* с четырьмя модулями **SIMD**. Для DSP *Elcore_24* (2 модуля **SIMD**) директивы не поддерживаются.

3.11.51. .set

Директива .set полностью эквивалентна директиве .equ.

3.11.52. .short

Директива .short X размещает в памяти целое *16*-битное число **X**.

3.11.53. .simd

Директива .simd указывает, что при генерации области данных следует предполагать параллельный режим работы DSP-ядер. Действует только на

операции выделения места под константы (только длинной *32/64* бита). При этом константа выравнивается по границе нулевой секции и дублируется для всех 4-х DSP-ядер. Режим отключается по директиве .scalar. Пример:

```
 2 0000 00000001    .dl  1
 3                  .simd
```

Три следующих слова обеспечивают выравнивание.

```
 4 0001 00000000    .double 0.25
 4      00000000
 4      00000000
 4      0000FFFF
 4      0000FFFF
 4      0000FFFF
 4      0000FFFF
 4      40000000
 4      40000000
 4      40000000
 4      40000000
 5 000c 00000002    .dl  2,3
 5      00000002
 5      00000002
 5      00000002
 5      00000003
 5      00000003
 5      00000003
 5      00000003
```

Каждой секции достается по 4 байта.

```
 8 001c 04030201    .db  1,2,3,4
 8      04030201
 8      04030201
 8      04030201
```

Примечание: директивы .simd и **.scalar** актуальны только для DSP *Elcore* с четырьмя модулями **SIMD**. Для DSP *Elcore_24* (2 модуля **SIMD**) директивы не поддерживаются.

3.11.54. .single

Директива .single X размещает в памяти *16*-битное число **X** с фиксированной точкой.

3.11.55. .skip

Директива .skip Number резервирует **Number** байт памяти.

3.11.56. .space

Директива .space Number,Fill резервирует **Number** байт памяти и заполняет их значением **Fill**.

3.11.57. .struct

Директива .struct предназначена для задания абсолютных имен в виде смещений. Например,

```
  .struct 0
field1:
.struct field1 + 4
field2:
.struct field2 + 4
field3:
```

В данном примере *field1-filed3* получают значения *0*, *4*, *8*.

3.11.58. .text

Директива .text N указывает, что elas должен ассемблировать все последующие операции в конец подсекции *text* с номером **N**. Если номер не указан, по умолчанию используется подсекция *0*.

3.11.59. .title

Директива .title Str задает заголовок листинга.

3.11.60. .word

Директива .word X размещает в памяти целое *32*-битное число **X**.

3.11.61. Зарезервированные директивы для отладки

Для отладки высокоуровневых языков зарезервированы следующие директивы: **.def**, **.desc**, **.dim**, **.endef**, **.func**, **.endfunc**, **.file**, **.ident**, **.line**, **.ln**, **.scl**, **.size**, **.stab**, **.stabn**, **.stabs**, **.tag**, **.type** и **.val**.

3.12. Макросы структурного программирования

3.12.1. Введение

Для упрощения программирования следует использовать **макросы структурного программирования**. Файл со стандартными макросами приведен в конце книги. **Макросы структурного программирования** позволяют проще записывать разнообразные конструкции, избавляют от необходимости следить за

метками, но, естественно, уменьшают производительность. В этой главе описаны следующие **макросы структурного программирования**:

- макросы организации ветвлений (**if**/**else**/**endif**);
- макрос организации цикла for;
- макрос организации цикла while;
- макрос организации цикла loop;
- макросы организации цикла repeat/until;
- макрос continue;
- макрос break.

3.12.2. Организация ветвлений if/else/endif

Для организации **ветвлений** применяется конструкция **if**/**else**/**endif**. **If** имеет две формы. В первой форме условие задается модификатором, во второй - записывается в виде трех параметров, где второй параметр задает операцию сравнения. Макрос **else** предназначен для задания блока кода, выполняющегося, если условие в **if** ложно. Макрос **endif** означает конец блока условий. Примеры использования:

```
if.eq
 add r2,r4
endif

if.ne
  add r3,r5
else
  add r4,r6
endif

if.l r2,"=",r6 ; В данном случае указан еще и размер операндов
.l
  add r3,r9
endif
```

3.12.3. Цикл for

Макрос **for** предназначен для организации *циклов с параметром*. Можно задавать до 5 аргументов. Первые три - обязательны. Первый аргумент задает регистр для параметра цикла, второй - начальное значение параметра, третий - граничное значение. Четвертый, если присутствует, задает направление изменения параметра. По умолчанию параметр увеличивается. Пятый задает шаг. Тело цикла должно заканчиваться макросом **endf**. Пример использования цикла **for**:

```
for r1,5,8,"+",2 ; в данном примере цикл будет исполняться от 5 до 8 с
увеличением параметра r1 на 2 каждую
  add r1,r3    ; итерацию
endf
```

For может иметь модификатор **.l** для указания размера используемого регистра.

3.12.4. Цикл while

Макрос **while** предназначен для организации *циклов с предусловием*. Код в теле цикла будет выполняться пока верно условие, указанное в **while**. Условие задается либо в виде модификатора, либо в виде трех параметров, где второй параметр задает операцию сравнения. Тело цикла должно заканчиваться макросом **endw**. Пример:

```
while  r1,"<>",r3
 add r2,r3
endw
```

В данном примере содержимое регистра *r2* будет прибавляться к содержимому регистра *r3* до тех пор, пока содержимое *r3* не станет равным содержимому регистра *r1*.

Следует помнить, что если условие, указанное в **while** ложно, то ни одной итерации не будет выполнено. Если необходимо, чтобы хотя бы одна итерация выполнилась, используйте *цикл с постусловием* repeat/until.

3.12.5. Цикл loop

Макрос **loop** позволяет осуществить *аппаратный DO-цикл*, то есть выполнить некоторые действия определенное число раз. Так как вся обработка осуществляется макропроцессором, а он не знает точную длину команд, то возникает проблема корректного задания адреса последней операции в **endl**. Соответственно, следует либо задать эту длину (1/2 слова) в самой команде **endl**, либо **endl** вставит операцию **NOP** в качестве последней команды цикла. Пример:

```
loop   7
 add   r2,r7
endl   1
```

В данном примере содержимое регистра *r2* будет прибавляться к содержимому регистра *r7* 7 раз.

3.12.6. Цикл repeat/until

Конструкция **repeat**/**until** предназначена для организации *циклов с постусловием*. Тело цикла выполняется пока условие, указанное в **until**, ложно. В **until** используется стандартная схема для задания условий if. Например:

```
repeat
  add #1,r1
if  r2,"<>",r3
  continue
else
  break
endif
until.eq
```

В данном примере тело цикла будет выполняться до тех пор, пока содержимое *r2* не станет равным содержимому регистра *r3*.

Циклы с постусловием следует использовать тогда, когда необходимо, чтобы была выполнена хотя бы одна итерация.

3.12.7. continue

Макрос **continue** вызывает переход к следующей итерации цикла (for, loop, while, repeat). В циклах while/repeat происходит переход на проверку условия. В цикле for - на увеличение параметра цикла.

3.12.8. break

Макрос **break** служит для немедленного выхода из цикла.

4. Компоновка программ для ELCORE

4.1. Введение

Компоновщик программ **elcore-elvis-elf-ld** (**elld**) компонует выполняемый файл из набора объектных файлов и, если это необходимо, библиотек.

Вызов **компоновщика** из командной строки: **elcore-elvis-elf-ld** *ключи файлы*

Для более точного задания правил компоновки возможно использование управляющего файла компоновщика.

4.2. Ключи компоновщика

Для работы с компоновщиком используется следующий набор ключей:

- **-o file** - указать, что файлом результата должен быть **file** (по умолчанию это *a.out*);
- **-M** - вывести карту памяти на стандартный вывод;
- **-r** - не удалять информацию о перемещаемых символах для досборки;
- **-s** - выбросить всю информацию о символах;
- **-S** - выбросить всю информацию для отладчика;
- **-Map file** - задать файл для вывода карты памяти;
- **-Ttext addr** - установка адреса **addr** для секции кода (по умолчанию это *0*);
- **-Tdata addr** - установка адреса **addr** для секции данных (по умолчанию это *0*);
- **-Tbss addr** - установка адреса **addr** для неинициализированных данных введенных директивами .comm/.lcomm.

После использования elas и elld получаются объектные файлы в формате **elf**. Для того чтобы их можно было слинковать с объектными файлами RISC-ядра, они должны быть преобразованы специальной программой **elcopy**.

4.3. Управляющий файл компоновщика

4.3.1. Введение

Для более точного задания правил компоновки следует использовать **управляющий файл**. Для работы с этим файлом используются следующие директивы:

- SECTIONS;
- ENTRY;
- INCLUDE;
- FILE;
- GROUP;

- OUTPUT;
- SEARCH_DIR;
- STARTUP;
- OUTPUT_FORMAT;
- TARGET;
- ASSERT;
- EXTERN;
- NOCROSSREFS;
- OUTPUT_ARCH;
- PROVIDE;
- MEMORY.

Для задания **выражений** можно использовать любые С-выражения, т.е. **=**, **+=**, **-=**, ***=**, **/=**, **<<=**, **>>=**, **&=**, **|=**. При этом "**.**" обозначает текущую позицию. После выражения обязательно ставить "**;**".

Указание стартовой точки.

По умолчанию стартовой точкой программы становится один из символов следующего списка:

- символ, заданный ключом -e компоновщика;
- символ, указанный в директиве компоновщика ENTRY;
- символ ***start***, если он определен;
- первый байт ***.text***-секции, если он определен;
- адрес *0*.

Комментарии в управляющем файле компоновщика можно оформлять в С-стиле, т.е. при помощи /* … */.

По умолчанию, среда MCS использует управляющий файл *<имя модуля>.xl*, например *dsp.xl*. Чтобы использовать другой файл, следует в командной строке компоновщика заменить *%Unit.xl* на имя необходимого файла. Для получения более подробной информации о настройке инструментария в среде MCS см. книгу **MC Studio**.

4.3.2. SECTIONS

Управляющий файл компоновщика должен, как минимум, содержать директиву **SECTIONS**. Директива **SECTIONS** задает расположение секций в памяти. В директиве задается группа присваиваний переменным и управление выводом секций. Например:

```
SECTIONS
{
    . = 0x100;
    .text : { *(.text) }
```

```
    . = 0x800;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

В данном примере выполняются присваивания текущей позиции *"."* и вывод всех (*"*"*) секций *.text* с позиции *0x100* памяти, а остальных секций - с позиции *0x800*.
Полный синтаксис директивы следующий:

```
SECTION [ADDRESS] [(TYPE)] : [AT(LMA)]
{
 OUTPUT-SECTION-COMMAND
 OUTPUT-SECTION-COMMAND
} [>REGION] [AT>LMA_REGION] [:PHDR :PHDR ...] [=FILLEXP]
```

Здесь:

- **ADDRESS** - задает адрес для данной секции. Если адрес не указан, то используется либо текущее значение счетчика, либо значения соответствующего **REGION**. Это виртуальный адрес;
- **TYPE** - задает тип секции;
- **NOLAD** - указывает не загружать секцию в память;
- **DSECT**, **COPY**, **INFO**, **OVERLAY** - указывает не выделять для секции места;
- **LMA** - задает адрес загрузки (по умолчанию совпадает с **ADDRESS**);
- **REGION** - задает область для размещения (см. директиву MEMORY);
- **LMA_REGION** - задает область для загрузки;
- **PHDR** - размещение в рамках сегментов;
- **FILLEXP** -задает значение для заполнения не специфицированных областей.

4.3.3. ENTRY

Директива ENTRY(Sym) указывает, что в качестве стартовой точки должен использоваться символ **Sym**.

4.3.4. INCLUDE

Директива INCLUDE позволяет включить в данной точке еще один управляющий файл компоновщика.

4.3.5. FILE

Директива FILE(file1 file2 ...) (или **FILE(file1,file2,...)**) позволяет явно подключить (аналогично тому, как это делается из командной строки), дополнительные файлы для сборки.

4.3.6. GROUP

Директива GROUP аналогична директиве FILE, но применяется только для библиотек. Указывает на необходимость многократного просмотра данных библиотек до полного исчерпания возможностей разрешения имен.

4.3.7. OUTPUT

Директива OUTPUT(File) задает файл для вывода. По умолчанию вывод осуществляется в файл *a.out*.

4.3.8. SEARCH_DIR

Директива SEARCH_DIR задает список директорий для поиска библиотек.

4.3.9. STARTUP

Директива STARTUP(File) аналогична директиве INCLUDE, но дополнительно требует размещения файла **File** первым.

4.3.10. OUTPUT_FORMAT

Директива OUTPUT_FORMAT(Name) задает формат выходного файла.

4.3.11. TARGET

Директива TARGET задает формат входного файла.

4.3.12. ASSERT

Директива ASSERT(Expr, mssg) позволяет проверить корректность выражения **Expr** (например, ограниченность размера секции и т.п.) и выдать диагностическое сообщение **mssg**.

4.3.13. EXTERN

Директива EXTERN позволяет принудительно объявить набор имен неопределенным.

4.3.14. NOCROSSREFS

Директива NOCROSSREFS позволяет задать список **секций** (как правило, оверлейных), которые не имеют права ссылаться друг на друга.

4.3.15. OUTPUT_ARCH

Директива OUTPUT_ARCH позволяет получить выходной файл в "чужой" архитектуре.

4.3.16. PROVIDE

Директива PROVIDE(Sym = Exp) позволяет задать значение символу **Sym** равным выражению **Exp**. При этом значение задается только в том случае, если символ более нигде не определен.

4.3.17. MEMORY

Директива MEMORY позволяет задать область памяти для размещения каких-либо секций и имеет следующий синтаксис:

```
MEMORY
{
        NAME [(ATTR)] : ORIGIN = ORIGIN, LENGTH = LEN
        ...
}
```

5. Библиотекарь

Библиотекарь (**elcore-elvis-elf-ar**) позволяет создавать библиотеки объектных модулей. Библиотекарь **выполняет** следующие функции:

- создание библиотеки модулей;
- добавление объектного файла в библиотеку;
- удаление и замена объектного файла в библиотеке.

Для ускорения выделения требуемых для сборки модулей из библиотеки, библиотекарь создает в ней таблицу символов.

Запуск библиотекаря из командной строки: **elcore-elvis-elf-ar** *ключи имя_библиотеки имена_файлов*

Библиотекарь имеет следующие ключи:

- **-d** - удалить файл из библиотеки;
- **-q** - быстрое (без выполнения всех действий над таблицей символов) добавление файла в библиотеку;
- **-r** - заменить или добавить файл в библиотеку;
- **-t** - выдать список файлов в библиотеке;
- **-x** - извлечь файл из библиотеки.

Например, *elcore-elvis-elf-ar -r libx.a nfile.o* добавляет файл *ngile.o* к библиотеке *libx.a*.

6. Анализатор/преобразователь форматов для объектных и выполняемых файлов eldump

6.1. eldump

Программа **elcore-elvis-elf-objdump** (**eldump**) предназначена для проверки, анализа и обработки объектных и выполняемых файлов. Включает в себя набор средств по отображению отдельных составляющих файлов, межформатному преобразованию, например, с генерацией *S-records*, дизассемблированию.

Дизассемблер предназначен для обратного преобразования объектного/выполняемого кода в код на языке Ассемблера с целью проверки и анализа его.

Замечание: Дизассемблер *dump*-памяти выводит в формате ядра DSP, т.е. LSB.

Запуск программы **eldump** из командной строки: **elcore-elvis-elf-objdump** *ключи файлы*

Программа **eldump** имеет следующие ключи:

- **-a** - просмотр заголовка архива;
- **-f** - просмотр заголовка файла;
- **-p** - просмотр заголовка объектного файла;
- **-h** - просмотр заголовков секций;
- **-x** - просмотр содержимого всех заголовков;
- **-d** - дизассемблирование выполняемого кода;
- **-D** - дизассемблирование всех секций;
- **-S** - дизассемблирование;
- **-s** - вывод секций;
- **-t** - выдача таблицы символов;
- **-r** - выдача информации о таблице перемещений;
- **-V** - информация о версии программы;
- **-i** - список поддерживаемых архитектур;
- **-H** - вывести описание всех ключей.

7. Примеры макросов

```
; L_ENn - end of construction
; L_CNn - point for continue
; L_BCn - point for continue after condition
; L_LABEL - label generator
; L_CUR   - L_C - point to else
; L_END    - L_E - point to end
; L_LOOP  - L_L - type of loop
; L_LEVEL - level of construction
 .set L_LABEL,1
 .set L_LEVEL,0
 .set L_END,0
 .set L_CUR,0
 .set L_LOOP,0

 .macro  save_level,number
  .set   L_C\number,L_CUR
  .set   L_D\number,L_END
  .set   L_L\number,L_LOOP
  .set   L_CUR,L_LABEL
  .set   L_END,L_LABEL
  .set   L_LABEL,L_LABEL+1
  .set   L_LEVEL,L_LEVEL+1
 .endm

 .macro  unsave_level,number
  .set   L_CUR,L_C\number
  .set   L_END,L_D\number
  .set   L_LOOP,L_L\number
  .set   L_LEVEL,L_LEVEL-1
 .endm

 .macro branch_noteq,label
  b.ne   \label
 .endm

 .macro branch_notne,label
  b.eq   \label
 .endm

 .macro if,rone,op,rtwo
  .ifne  \NARG
    cmp\0   \rone,\rtwo
    .ifc =,\op
```

```
   b.ne L_CN{L_LABEL}
 .endif
 .ifc  <>,\op
      b.ne    L_CN{L_LABEL}
   .endif
.else
  branch_not\0 L_CN{L_LABEL}
.endif
  save_level {L_LEVEL+1}
.endm

.macro  else
   b  L_EN{L_END}
   L_CN{L_CUR}:
   .set L_CUR,0
.endm

.macro  endif
 .ifne  L_CUR
   L_CN{L_CUR}:
 .endif
 L_EN{L_END}:
 unsave_level  {L_LEVEL}
.endm

.macro  repeat
     L_BC{L_LABEL}:
  save_level {L_LEVEL+1}
  .set  L_LOOP,L_LABEL-1
.endm

.macro  until
  L_CN{L_CUR}:
  .ifne \NARG
  cmp\0 \rone,\rtwo
  .ifc  =,\op
      b.neL_BC{L_CUR}
  .endif
  .ifc  <>,\op
       b.ne    L_BC{L_CUR}
     .endif
.else
   branch_not\0 L_BC{L_CUR}
.endif
    L_EN{L_END}:
 unsave_level  {L_LEVEL}
```

```
.endm

.macro break
  b  L_EN{L_LOOP}
.endm

.macro continue
  b  L_CN{L_LOOP}
.endm

.macro for,rn,from,to,op,step
  .ifge \NARG-5
      .setL_TMP,\step
  .else
      .setL_TMP,1
  .endif
  move  \from,\rn
  b  L_TMP{L_LABEL}
    L_CN{L_LABEL}:
  .ifge \NARG-4
      .ifc-,\op
        sub\0L_TMP,\rn
      L_TMP{L_LABEL}:
      cmp\0   \to,\rn
      b.ge L_EN{L_LABEL}
    .else
      add\0   L_TMP,\rn
      L_TMP{L_LABEL}:
      cmp\0   \to,\rn
      b.le L_EN{L_LABEL}
    .endif
  .else
    add\0  L_TMP,\rn
    L_TMP{L_LABEL}:
    cmp\0  \to,\rn
   b.le L_EN{L_LABEL}
  .endif
  save_level {L_LEVEL+1}
  .set  L_LOOP,L_LABEL-1
.endm

.macro endf
  b  L_CN{L_CUR}
  L_EN{L_CUR}:
  unsave_level {L_LEVEL}
.endm
```

```
.macro  loop,num
  do  \num,L_CN{L_LABEL}


  save_level       {L_LEVEL+1}
     .set     L_LOOP,L_LABEL-1
.endm

.macro  endl,size
  .ifge \NARG-1
   .set L_CN{L_CUR},.-\size
  .else
   L_CN{L_CUR}:
   nop
  .endif
  L_EN{L_CUR}:
  unsave_level {L_LEVEL}
.endm

.macro  while,rone,op,rtwo
  L_CN{L_LABEL}:
  .ifne \NARG
   cmp\0   \rone,\rtwo
   .ifc =,\op
      b.ne L_CN{L_LABEL}
   .endif
   .ifc  <>,\op
          b.ne     L_CN{L_LABEL}
       .endif
  .else
   branch_not\0 L_EN{L_LABEL}
  .endif
  save_level       {L_LEVEL+1}
     .set     L_LOOP,L_LABEL-1
.endm

.macro  endw
  b  L_CN{L_CUR}
  L_EN{L_CUR}:
  unsave_level {L_LEVEL}
.endm
```

Пример использования:

```
while   r1,"<>",r3
add  r2,r3
endw
```

```
loop 7
add  r2,r7
endl 1

for  r1,5,8,"+",2
  add r1,r3
endf

repeat
  add #1,r1
  if r2,"<>",r3
   continue
  else
   break
  endif
until.eq

if.eq
add r2,r4
endif
if.ne
  add r3,r5
else
  add r4,r6
endif
if.l r2,"=",r6
  add r3,r9
endif

.end
```

8. Предметный указатель