

Компилятор C/C++ Clang для DSP ELcore-30M

Руководство пользователя

ПОРЯДОК ИСПОЛЬЗОВАНИЯ 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». ОАО НПЦ «ЭЛВИС» является единственным правообладателем исключительных авторских прав на настоящую документацию.

Настоящую документацию, не иначе как по предварительному согласию ОАО 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).

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

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

Передача настоящей документации не означает передачи каких-либо авторских прав ОАО 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.

ОАО НПЦ «ЭЛВИС» ©, 2016

АННОТАЦИЯ

В документе описан компилятор Clang для процессора Elcore-30M. В разделе 2 «Введение» описаны компилятор и инструкция по сборке и запуску минимальной программы “Hello, world”. В разделе 3 «Ключи компилятора» описан набор поддерживаемых опций для управления этапами сборки программы, форматом и объемом диагностической информации, уровнем оптимизации кода. В разделе 4 «Входные данные» описаны поддерживаемые диалекты языков C/C++, набор расширений языка, технология использования ассемблерных вставок и специфичные для Elcore-30M builtin-функции. В приложении 1 приведены примеры программ с builtin-функциями.

Соглашение о вызовах компилятора описано в отдельном документе.

ОГЛАВЛЕНИЕ

1.	ВВЕДЕНИЕ	5
1.1.	Компилятор c/c++	5
1.2.	Быстрый старт.....	6
	Пример запуска.....	6
	Специальные ключи компилятора	8
2.	КЛЮЧИ КОМПИЛЯТОРА	10
2.1.	Общие опции (опции, управляющие видом вывода)	10
2.2.	Опции, управляющие выбором стандарта	11
2.3.	Опции, определяющие предупреждения и ошибки	12
2.4.	Опции для отладки и отладочной информации.....	13
2.5.	Опции препроцессора	13
2.6.	Опции директорий.....	15
2.7.	Опции генерации кода	16
2.8.	Опции оптимизации	16
2.9.	Архитектурно – зависимые опции	17
2.10.	Опции ассемблера	18
2.11.	Опции линковщика.....	18
2.12.	Переменные окружения компилятора	19
3.	ВХОДНЫЕ ДАННЫЕ	20
3.1.	Поддерживаемые расширения GNU C	20
3.2.	Расширения Clang C/C++	21
3.3.	Список builtin-функций для архитектуры Elcore-30M.....	24
4.	АССЕМБЛЕРНЫЕ ВСТАВКИ	27
5.	ОБРАЩЕНИЕ К ВНЕШНЕЙ ПАМЯТИ	31
1.	ПРИЛОЖЕНИЕ 1. Примеры программ с builtin-функциями	32
1.1.	ПРИМЕР 1. Бинарные операции над массивами short.....	32
1.2.	ПРИМЕР 2. Бинарные операции над массивами int	33
1.3.	ПРИМЕР 3. Поиск минимума, максимума и их индексов в массиве	34
1.4.	ПРИМЕР 4. Бинарные операции над массивами float	35
1.5.	ПРИМЕР 5. Бинарные операции над массивами комплексных чисел	36

1. ВВЕДЕНИЕ

1.1. Компилятор c/c++

Компилятор C/C++ (далее компилятор) для DSP-ядра Elcore-30M преобразовывает код, написанный на языках C/C++ согласно стандартам ANSI C(C89, C90), C99, C++11, C++14 в код на языке ассемблер для DSP-ядра Elcore-30M.

Компилятор поставляется в составе набора инструментов eltools (компилятор, ассемблер, стек библиотек, компоновщик) для ОС Windows и ОС Linux.

Компилятор создан на базе LLVM/Clang-инфраструктуры и поддерживает стандартный набор оптимизаций clang. Для эффективного использования возможностей ядер Elcore-30M используется специфичный архитектурно-зависимый набор оптимизаций. Для создания объектного файла компилятор может вызывать ассемблер из пакета инструментов eltools. Для получения исполняемой программы компилятор также может вызвать компоновщик.

Компилятор поддерживает следующий стек стандартных библиотек:

Llvm compiler-rt (библиотека оптимизированных функций поддержки компиляции (преобразование типов, арифметика, плавающая арифметика и т. п.));

newlib-2.2 (стандартная библиотека языка C). Обращения к операционной системе (syscalls) эмулируются соответствующими обращениями к симулятору или оборудованию, по причине отсутствия операционной системы для Elcore-30M.

1.2. Быстрый старт

Основные ключи компилятора

Основные ключи компилятора:

-target elcore в качестве целевой платформы задаётся DSP Elcore-30M;

-mcpu=elcore30m (по умолчанию) выбор процессора Elcore-30M;

-c результатом компиляции должен быть объектный файл (в противном случае делается попытка собрать программу);

-S результатом программы должен быть файл на языке ассемблера. По умолчанию, имя файла с ассемблерным кодом формируется из имени исходного файла заменой суффикса '.c', '.i', и.т.д. на '.s'. В сочетании с опцией **-emit-llvm** выдаёт llvm-IR

-o file результат работы компилятора записывается в файл file

-Идиректория добавить директорию в список поиска для директивы **#include** файл

-Лдиректория добавить директорию в список поиска компоновщиком библиотек в процессе сборки

-Лбиблиотека использовать данную библиотеку во время компоновки

-Wa,опция передает 'опцию' в качестве опции ассемблеру. Если 'опция' содержит запятые, текст расщепляется на опции.

-g компиляция с добавлением информации для отладчика

-Ox компиляция с оптимизацией, где x – уровень оптимизации, 0 – минимальные оптимизации, 3 – максимальная оптимизация.

Пример запуска

Для воспроизведения шагов в этом разделе необходим полный пакет инструментов eltools и программная модель sim3 для Elcore-30M.

Пример простейшей программы на языке C (файл prog.c):

```
#include <stdio.h>
int main() {
printf("Hello, world!\n");
return 0;
}
```

Запуск компилятора осуществляется командой:

```
~/eltools/bin/clang prog.c -target elcore -mcpu=elcore30m -lsim
```

Здесь:

-prog.c – имя компилируемой программы

-target elcore - в качестве целевой платформы задаётся DSP Elcore

-mcpu=elcore30m - выбор процессора

-lsim – библиотека, подменяющая системные вызовы (printf использует системный вызов write) на обращения к симулятору.

В результате этого запуска будет получен исполняемый файл с традиционным именем по умолчанию a.out. Для запуска симулятора используем скрипт (файл shell.fs):

```
trace -flog.log vt.shell
cm nv02t
loadelf a.out
dsp0.dcsr=0x4000
dsp0.pc= ____start
run
exit
```

Запуск симулятора из linux терминала осуществляется командой:

```
~/sim3/trunk/bin/freeshell.x86 shell.fs
```

В результате работы симулятора будет получен файл log, в котором содержится выдача программы:

```
Hello, world!
```

Достаточно полезную информацию в случае затруднений при сборке программ можно увидеть, если к запуску компилятора добавить опцию -v. Например:

```
~/eltools/bin/clang prog.c -target elcore -mcpu=elcore30m -lsim -v
```

В этом случае компилятор выдаст сообщение вида:

```
clang version 3.6.0 (tags/RELEASE_360/final)
Target: elcore
Thread model: posix
"/home/user/eltools/bin/clang-3.6" -cc1 -triple elcore -S -disable-free -disable-
llvm-verifier -main-file-name prog.c -mrelocation-model static -mthread-model
posix -mdisable-fp-elim -no-integrated-as -mconstructor-aliases -target-cpu el-
core30m -v -dwarf-column-info -resource-dir
/home/user/eltools/bin/./lib/clang/3.6.0 -internal-externc-isystem
/home/user/eltools/bin/./elcore30m-elvis-elf/include -fno-dwarf-directory-asm -
fdebug-compilation-dir /home/user/test1 -ferror-limit 19 -fmessage-length 125 -
mstackrealign -fobjc-runtime=gcc -fdiagnostics-show-option -fcolor-diagnostics -o
/tmp/prog-51b51b.s -x c prog.c
clang -cc1 version 3.6.0 based upon LLVM 3.6.0 default target x86_64-unknown-
linux-gnu
#include "..." search starts here:
#include <...> search starts here:
/home/user/eltools/bin/./elcore30m-elvis-elf/include
/usr/local/include
/home/user/eltools/bin/./lib/clang/3.6.0/include
```

```

/usr/include
End of search list.
"/home/user/eltools/bin/elcore30m-elvis-elf-as" -mcx7 -Wdojb-force-long -o
/tmp/prog-1cbf64.o /tmp/prog-51b51b.s
"/home/user/eltools/bin/elcore30m-elvis-elf-ld" -o a.out /tmp/prog-1cbf64.o
/home/user/eltools/bin/./elcore30m -elvis-elf/lib/crt0.o -lsim -
L/home/user/eltools/bin/./elcore30m -elvis-elf/lib -lc -lcrt -
T/home/user/eltools/bin/./elcore30m -elvis-elf/lib/generic.ld

```

В представленном сообщении можно увидеть версию компилятора и шаги создания объектного файла с опциями на каждом этапе (вызовы препроцессора, кодогенератора, ассемблера, линковщика). В этом случае используется опция -S для генерации ассемблера и результат работы ядра компилятора сохраняется в промежуточный временный файл /tmp/prog-51b51b.s. Далее можно увидеть, что этот файл используется ассемблером для генерации ещё одного промежуточного файла - объектного /tmp/prog-1cbf64.o и запуск компоновщика. Кроме заданной библиотеки libsim компоновщик использует по умолчанию библиотеки libc, libunwind, libcert, а также скрипт dsp.ld. Все эти файлы он ищет в каталоге по умолчанию eltools/elcore30m-elvis-elf/lib.

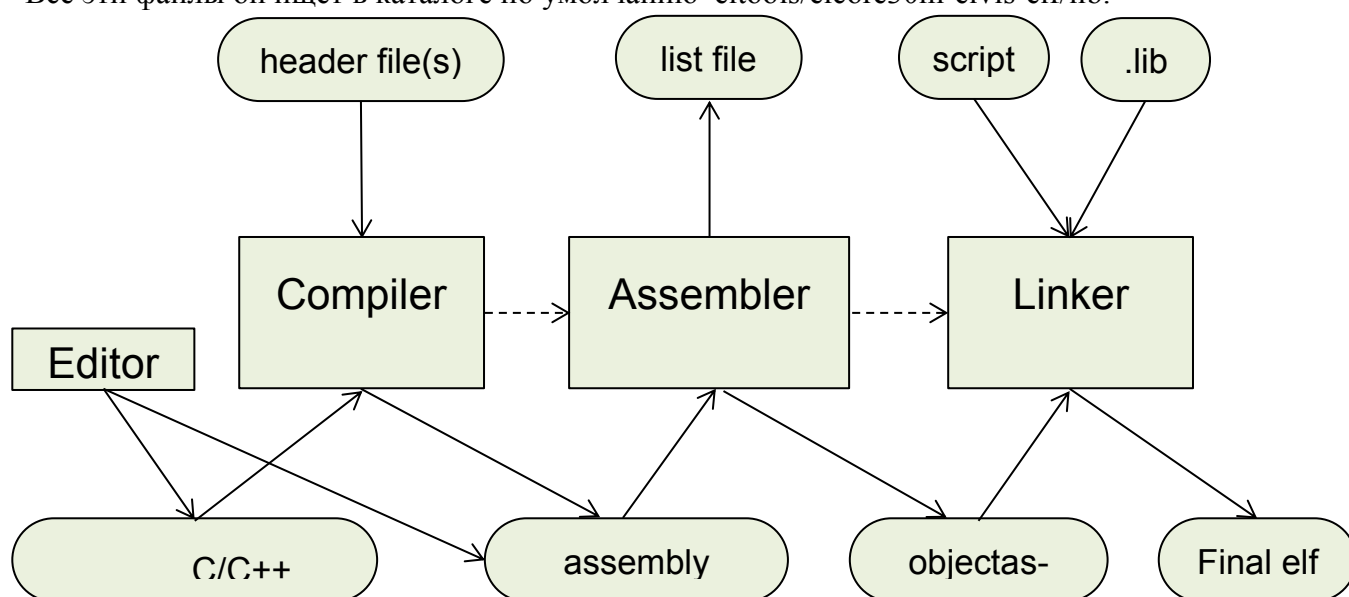


Рисунок 1.1. Этапы сборки программы для Elcore-30M

Специальные ключи компилятора

Для гибкого управления процессом архитектурно-зависимой оптимизации кода под процессоры DSP Elcore добавлены специальные ключи, позволяющие включить или выключить отдельные оптимизационные проходы. Специальные ключи задаются через опцию -mllvm.

Основные ключи следующие:

-elcore-enable-delay-filler – опция разрешает использование слотов задержки инструкций перехода. Без неё в слот задержки помещается пустая операция (nop)

-elcore-use-indexed-loadstore – опция, разрешающая использование постинкрементной и постдекрементной адресаций.

-regalloc=pbqp – опция, позволяющая выбрать используемый аллокатор регистров. Выбор аллокатора PBQB позволяет использовать адресацию со смещением (регистр адреса + регистр индекса), которая не используется в других моделях распределения регистров.

-elcore-enable-copy-cse – опция, разрешающая коалеасинг(объединение) нескольких виртуальных регистров с целью вынесение инварианта за пределы базового блока(тела цикла или ветви условной операции). Позволяет улучшить качество кодогенерации для архитектур, использующих отдельные регистровые файлы для адресов и для вычислений.

-elcore-enable-selectaddr-rr – опция, разрешающая использовать адресацию регистр адреса + регистр индекса.

-elcore-enable-vliw – опция, разрешающая объединение независимых инструкций в пакеты VLIW

Примеры запусков компилятора с использованием специальных опций:

```
~/eltools/bin/clang prog.c -target elcore -mcpu=elcore30m -lsim -v -mllvm -elcore-enable-vliw
```

2. КЛЮЧИ КОМПИЛЯТОРА

Работа компилятора состоит из нескольких этапов: препроцессор, компилятор (преобразование исходной программы C/C++ в промежуточное представление, преобразование промежуточного представления в программу на языке ассемблера), ассемблер (преобразование программы на языке ассемблера в объектный файл в формате ELF).

Компилятор clang предоставляет широкий набор ключей, которые управляют сценарием компиляции, форматом входных и выходных файлов, параметрами оптимизации и т.д. Все ключи для удобства разбиваются на группы:

- общие опции (опции, управляющие видом вывода)
- опции, управляющие выбором стандарта
- опции предупреждений
- опции отладки
- опции оптимизации
- опции препроцессора
- опции генерации кода
- опции директорий
- архитектурно-зависимые опции

2.1. Общие опции (опции, управляющие видом вывода)

Задают стадии сборки программы: препроцессор, компилятор, ассемблер, линковщик.

-c -S -E -o FILE -pipe -v -x язык -x none

Таблица 2.1. - Опции компилятора, управляющие стадиями компиляции

Ключ	Команда
-c	Компилировать или ассемблировать исходные файлы, но не линковать. Стадия линковки не выполняется. Конечный вывод происходит в форме объектного файла для каждого исходного файла. По умолчанию имя объектного файла формируется из имени исходного файла заменой расширения '.c', '.i', '.s' и т.д. на '.o'.
-S	Остановиться сразу после стадии препроцессирования и компиляции, не ассемблировать. Вывод производится в форме файла с ассемблерным кодом для каждого не ассемблерного входного файла. По умолчанию, имя файла с ассемблерным кодом формируется из имени исходного файла заменой суффикса '.c', '.i', и т.д. на '.s'
-E	Остановиться после препроцессора, не запускать компилятор. Вывод делается в форме препроцессированного исходного кода, который посылается на стандартный вывод
-o file	Поместить вывод в файл 'file'. Эта опция применяется вне зависимости от вида выходного файла (объектный файл, ассемблерный файл или Препроцессированный C код)
-pipe	Использовать системные каналы pipe (метод коммуникации процес-

Ключ	Команда
	сов) вместо временных файлов для коммуникации между различными стадиями компиляции
<i>-save-temps</i>	Сохраняет промежуточные "временные" файлы, которые помещаются в текущий каталог, а их имена основываются на имени исходного файла. Например, при компилировании 'foo.c' с опциями '-c -save-temps' будут сохранены файлы 'foo.i' (промежуточный, код C, результат работы препроцессора), 'foo.s' (промежуточный, ассемблер, результат работы компилятора), также как и 'foo.o' (объектный код после вызова ассемблера).
<i>-v</i>	Печатать (в стандартный вывод) команды, выполняемые для запуска стадий компиляции. Также печатать номер версии управляющей программы компилятора, препроцессора и самого компилятора
<i>-x язык</i>	Прямо специфицирует язык последующих входных файлов (даже если компилятор может выбрать язык на основании расширения имени файла). Эта опция действует на все входные файлы вплоть до следующего появления опции '-x'.
<i>-x none</i>	Выключает любое указание языка так, что последующие файлы обрабатываются в соответствии с расширениями имен.
<i>-v</i>	Распечатка версии программы
<i>-help</i>	Распечатка основных ключей clang
<i>-###</i>	Вывод этапов компиляции, без запуска программ
<i>-ccc-print-phases</i>	Вывод этапов компиляции в формате "{step}, name", без запуска программ
<i>-ccc-print-bindings</i>	Печать имен временных файлов для передачи данных между этапами компиляции
<i>-ccc-print-options</i>	Распечатка списка переданных параметров

2.2. Опции, управляющие выбором стандарта

Таблица 2.2. - Опции компилятора, управляющие выбором стандарта языка

Ключ	Команда
<i>-ansi</i>	Для C эквивалентно <code>-std=c90</code> , для C++ <code>-std=c++98</code>
<i>-std=std_name</i>	Выбор стандарта. Допустимые значения 'c90', 'c89', 'iso9899:1990' 'c99', 'c9x', 'iso9899:1999', 'iso9899:199x' 'c11', 'c1x', 'iso9899:2011' 'gnu90', 'gnu89' 'gnu99', 'gnu9x' 'gnu11', 'gnu1x' 'c++98', 'c++03' 'gnu++98', 'gnu++03' 'c++11', 'c++0x'
<i>-fno-asm</i>	Отключает возможность использования ассемблерных вставок
<i>-fno-builtin</i>	Отключает возможность использования встроенных функций

Ключ	Команда
<i>-fsigned-char</i> <i>-funsigned-char</i>	Выбор типа char (знаковый или беззнаковый)
<i>-fsigned-bitfield</i> <i>-funsigned-bitfield</i>	Выбор типа записей битовых полей (знаковые или беззнаковые)
<i>-fgnu-keywords</i>	Разрешает использовать расширенный набор символов GNU независимо от стандартного языка.

2.3. Опции, определяющие предупреждения и ошибки

Таблица 2.3. - Опции компилятора, управляющие печатью предупреждений

Ключ	Команда
<i>-w</i>	Отменяет все предупреждения
<i>-W</i>	Печатает дополнительные предупреждения
<i>-Weverything</i>	Печатает все предупреждения
<i>-Werror</i>	Превращает все предупреждения в ошибки
<i>-Werror=foo</i>	Превращает предупреждение 'foo' в ошибку
<i>-Wno-error=foo</i>	Отменяет трактовку предупреждения 'foo' как ошибки
<i>-Wfatal-errors</i>	Останавливает компиляцию после первой встретившейся фатальной ошибки
<i>-Wfoo</i>	Включает предупреждение 'foo'
<i>-Wno-foo</i>	Отключает предупреждение 'foo'
<i>-pedantic</i>	Включает предупреждения для расширений языка
<i>-Wsystem-headers</i>	Включает предупреждения для системных заголовочных файлов
<i>-ferror-limit=n</i>	Задаёт максимально количество ошибок, после которого прекращается синтаксический анализ. По умолчанию error-limit=20, сбрасывается командой error-limit=0

Таблица 2.4. - Опции компилятора, управляющие форматом вывода диагностической информации

Ключ	Команда
<i>-f[no]-show-column</i>	Включает печать номер столбца для ошибки или предупреждения
<i>-f[no]-show-source-location</i>	Добавляет строку из кода в информацию об ошибке или предупреждении
<i>-f[no-]color-diagnostics</i>	Цветовая подсветка ошибок и предупреждений при выводе на терминал
<i>-fdiagnostics-format=clang/msvc/vi</i>	Задаёт формат сообщения clang: prog.c:3:10: warning: ... msvc: prog.c(3,10) :warning: ... vi: prog.c +3,10: warning: ...
<i>-f[no-]diagnostics-print-source-range-info</i>	Вывод диагностического сообщения в формате prog.c:21:3:{21:8-21:10}: warning: ...

Ключ	Команда
<code>-f[no-]diagnostics-show-option</code>	Печать имени предупреждения в формате [-Woption]
<code>-fdiagnostics-show-category=none/id/name</code>	Включает печать категории ошибок или предупреждения, к которым относится найденная ошибка или предупреждение. По умолчанию <code>diagnostics-show-category=none</code>

2.4. Опции для отладки и отладочной информации

Таблица 2.5.- Опции компилятора, управляющие формированием отладочной информации

Ключ	Команда
<code>-g</code>	Включает генерацию отладочной информации в формате DWARF3
<code>-g[0,1,2,3]</code>	Задаёт степень подробности отладочной информации <code>-g0</code> – отладочная информация отключена, уровень, выставленный по умолчанию – ‘ <code>-g2</code> ’
<code>-f[no-]eliminate-unused-debug-symbols</code>	Удалять неиспользуемые символы из отладочной информации
<code>-f[no-]limit-debug-info</code>	Ограничивает отладочную информацию
<code>-print-file-name=библиотека</code>	Печатает полное абсолютное имя библиотечного файла 'библиотека', которое использовалось бы при линковке. С этой опцией ничего не компилируется и не линкуется, только печатается имя файла
<code>-print-prog-name=программа</code>	Печатает полное абсолютное имя 'программы'
<code>-print-search-dirs</code>	Печатает директории, используемые для поиска нужных библиотек и программ
<code>-emit-ast</code>	Выдает AST файлы (абстрактное синтаксическое дерево) исходного кода (не собирается)
<code>-emit-llvm</code>	Использует LLVM представление ассемблерных и объектных файлов

2.5. Опции препроцессора

Таблица 2.6.- Опции управления препроцессором

Ключ	Команда
<code>-C</code>	Указывает препроцессору не отбрасывать комментарии. Используется с опцией '-E'.
<code>-P</code>	Указывает препроцессору не генерировать директивы '#line'. Используется с опцией '-E'.
<code>-M</code>	Указывает препроцессору выводить правила для make, описывающие зависимости каждого объектного файла. Для каждого исходного файла препроцессор выводит одно make-правило,

Ключ	Команда
	цель которого - имя объектного файла. Правило может быть одиночной строкой или может быть продолжено с помощью '\'-новая строка, если оно длинное. Список правил печатается в стандартный вывод.
<i>-MM</i>	Аналогична опции '-M', но вывод упоминает только заголовочные файлы пользователя, включенные с помощью #include "файл". Системные заголовочные файлы, включенные с помощью #include <файл>, опускаются.
<i>-MG</i>	Обрабатывает отсутствующие заголовочные файлы как генерируемые файлы и считает, что они находятся в том же самом каталоге, что и исходный файл. Ключ должен использоваться совместно с '-M', или '-MM'.
<i>-H</i>	Вывод полных имен включаемых заголовочных файлов на стандартный вывод.
<i>-Dмакрос=определение</i>	Определяет макрос.
<i>-Uмакрос</i>	Отменяет определение макроса.
<i>-dM</i>	Указывает препроцессору вывести только список макроопределений, которые имеют действие в конце препроцессирования. Используется с опцией '-E'.
<i>-dD</i>	Указывает препроцессору передать все макроопределения в вывод в их последовательности в другом выводе.
<i>-dN</i>	Подобна '-dD' за исключением того, что макроаргументы и содержание опускаются. В вывод включается только '#define имя'.
<i>-include файл</i>	Обрабатывает 'файл' как ввод перед обработкой обычного входного файла. Фактически, содержимое 'файла' компилируется сначала. Любая опция '-D' или '-U' из командной строки обрабатывается до '-include файл' вне зависимости от порядка, в котором они записаны. Все опции '-include' и '-imacros' обрабатываются в том порядке, в котором они записаны.
<i>-imacros файл</i>	Обрабатывает 'файл' как ввод, аналогично опции <i>-include</i> . Макрокоманды, определенные в 'файле' становятся доступны для применения в главном вводе. Любая опция '-D' или '-U' из командной строки обрабатывается до '-imacros файл', вне зависимости от порядка, в котором они записаны. Все опции '-include' и '-imacros' обрабатываются в том порядке, в котором они записаны.
<i>-idirafter директория</i>	Добавляет каталог 'директорий' ко второму маршруту включения. В каталогах второго маршрута включения ищет, когда заголовочный файл не обнаружен ни в одном из каталогов в главном маршруте включения (маршрут, к которому добавляет опция '-

Ключ	Команда
	Г).
<i>-iprefix префикс</i>	Определяет 'префикс' как префикс для опции '-iwithprefix'.
<i>-iwithprefix директория</i>	Добавляет каталог ко второму маршруту включения. Имя каталога получается объединением 'префикса' и 'директории', где 'префикс' определялся предварительно опцией '-iprefix'. Если префикс ещё не определён, по умолчанию используется каталог, содержащий установленные проходы компилятора.
<i>-iwithprefixbefore директория</i>	Добавляет каталог к главному маршруту включения. Имя каталога получается объединением 'префикса' и 'директории', как в случае с '-iwithprefix'.
<i>-isystem директория</i>	Добавляет каталог 'директории' к началу второго маршрута включения, помечая его как системный каталог, так что он имеет ту же самую специальную обработку, что и стандартные системные каталоги.
<i>-undef</i>	Отменяет определение заданного системного макроса.

2.6. Опции директорий

Таблица 2.7.- Опции управления директориями

Ключ	Команда
<i>-I директория</i>	Добавляет 'директорию' в начало списка каталогов, используемых для поиска заголовочных файлов(include-файлов).
<i>-I-</i>	Все директории, которые были упомянуты до <i>-I-</i> будут использоваться для #include "файл", но не для #include <файл> (обычно, системные include-файлы), директории, упомянутые после <i>-I-</i> могут быть использованы в любом качестве.
<i>-L директория</i>	Добавляет 'директорию' в список поиска библиотек в процессе сборки.
<i>-working-directory value</i>	Задаёт директорию 'value' в качестве рабочей.
<i>-fmodule-cache-path <directory></i>	Задаёт путь к кэшу модулей.

2.7. Опции генерации кода

Таблица 2.8. - Опции управления кодогенератором

Ключ	Команда
<i>-femit-all-decls</i>	Генерирует все (в том числе неиспользуемые) объявления функций.
<i>-f[no-]inline-functions</i>	Разрешить [запретить] генерацию встроенных функций.
<i>-fshort-enums</i>	Выделяет для типа перечисления только такое количество байтов, которое нужно для объявленного диапазона возможных значений. Тип перечисления будет эквивалентен наименьшему целому типу, который имеет достаточно места.
<i>-fshort-wchar</i>	Преобразует тип данных <code>wchar_t</code> в тип <code>short</code> .
<i>-fpack-struct=<value></i> <i>-f[no-]pack-struct</i>	Устанавливает максимальное значение для выравнивания элементов структуры.
<i>-fstrict-enums</i>	Строгое определение диапазона перечислимых значений.
<i>-msoft-float</i>	Порождает код, содержащий библиотечные вызовы для функций с плавающей точкой.
<i>-Xassembler опция</i> <i>-Wa, опция, опция</i>	Передаёт 'опцию' в качестве опции ассемблеру. Если нужно передать опцию, которая имеет параметр, то необходимо использовать ' -Xassembler' дважды: один раз для опции и один раз для параметра.
<i>-Xclang опция</i>	Передаёт 'опцию' в качестве опции компилятору Clang. Если нужно передать опцию, которая имеет параметр, то необходимо использовать '-Xclang' дважды: один раз для опции и один раз для параметра.
<i>-Xpreprocessor опция</i>	Передаёт 'опцию' в качестве опции препроцессору.
<i>-Qunused-arguments</i>	Не выдавать предупреждения о неиспользуемых аргументах драйвера.

2.8. Опции оптимизации

-O0, -O1, -O2, -O3 – задает уровень оптимизации .

-Os – оптимизация кода по размеру.

Далее представлена таблица 9 с опциями, которые включают или отключают соответствующий проход оптимизации. В рабочем режиме управляются выбранным уровнем оптимизации (-O0, -O1, -O2, -O3 или -Os). Пользоваться ключами следует при необходимости получить более производительный код.

Таблица 2.9.- Архитектурно-независимые оптимизации

Наименование	Вид оптимизации
<i>f[no-]merge-all-constants</i>	Оптимизация объединения одинаковых констант в одну.

<code>f[no-]omit-frame-pointer</code>	Оптимизация по исключению при возможности указателя фрейма из эпилога и пролога функции.
<code>f[no-]optimize-sibling-calls</code>	Оптимизация хвостовых вызовов.
<code>-funroll-loops</code>	Разворачивание циклов.
<code>-mllvm -disable-machine-cse</code>	Поиск с удалением тривиально избыточных инструкций(CSE – common subexpression elimination) .
<code>-mllvm -optimize-regalloc</code>	Включить оптимальное распределение регистров.
<code>-mllvm -enable-misched</code>	Оптимизация по планированию потока инструкций.
<code>-mllvm -enable-tail-merge</code>	Оптимизация по слиянию хвостовых базовых блоков.
<code>-mllvm -disable-postra-machine-licm</code>	Оптимизация по выносу инвариантного кода из тела цикла после фазы распределения регистров(LICM –loop invariant code motion).
<code>-mllvm -disable-machine-sink</code>	Перемещение машинных инструкций в последующие линейные участки так, чтобы они не исполнялись по путям управления, на которых их результаты не используются.
<code>-mllvm -disable-lsr</code>	Понижение силы операций доступа к элементам массива в цикле, осуществляемым при помощи индуктивной переменной (LSR – loop strength reduction).
<code>-mllvm -disable-copyprop</code>	Оптимизация удаления избыточного копирования переменных.

2.9. Архитектурно – зависимые опции

Опции определяют целевую платформу и набор включаемых архитектурно-зависимых оптимизаций.

`-target elcore` – задает архитектуру elcore в качестве целевой.

`-mcpu=[elcore30m]` – выбор ядра и системы инструкции DSP в качестве целевого.

`-mllvm [опция]` – передает опцию кодогенератору.

Таблица 2.10. - Архитектурно-зависимые оптимизации кода

Ключ	Команда
<code>-elcore-enable-delay-filler</code>	Разрешает использование слотов задержки инструкций перехода. Без неё в слот задержки помещается пустая операция (nop).
<code>-elcore-use-indexed-loadstore</code>	Разрешает использование постинкрементной и постдекрементной адресации.
<code>-elcore-enable-copy-cse</code>	Разрешает объединение нескольких регистров с целью вынесения инварианта за пределы базового блока(тела цикла или ветви условной операции). Позволяет улучшить качество кодогенерации для архитектур, использующих отдельные регистровые файлы для адресов и для вычислений.
<code>-elcore-enable-</code>	Разрешает использование адресации регистр адреса + регистр ин-

<i>selectaddr-rr</i>	декса.
<i>-elcore-enable-vliw</i>	Разрешает объединение независимых инструкций в пакеты VLIW.
<i>-elcore-enable-hwloop</i>	разрешает использование аппаратных циклов.

2.10. Опции ассемблера

Для передачи опций ассемблеру есть два механизма:

1) Перед каждой опцией ассемблера указывать ключ –**Xassembler**.

2) Все опции, которые необходимо передать ассемблеру, указать через запятую с ключом -**Wa,<опция1, опция2, опция N>**.

2.11. Опции линковщика

Для передачи опций линковщику есть два механизма:

1) Перед каждой опцией ассемблера указывать ключ –**Xlinker**.

2) Все опции, которые необходимо передать линковщику, указать через запятую с ключом -**Wl,<опция1, опция2, опция N>**.

При этом список библиотек, путь к библиотекам нужно указывать без дополнительных опций. Для управления линковщиком используются опции.

Таблица 2.11. – Опции линковщика

Ключ	Команда
<i>-c</i>	Компилировать или ассемблировать исходные файлы, но не линковать. Стадия линковки не выполняется. Конечный вывод происходит в форме объектного файла для каждого исходного файла. По умолчанию имя объектного файла формируется из имени исходного файла заменой расширения '.c', '.i', '.s' и т.д. на '.o'.
<i>-L<library_path></i>	Использовать путь <i><library_path></i> как параметр линковщика при поиске библиотек.
<i>-l<library></i>	При линковке найти и использовать библиотеку <i>lib<library>.a</i> . Форма имени <i>«lib<library>.a»</i> является шаблоном для имени библиотеки и указания ключа. Библиотеки подключаются в порядке, указанном в командной строке. Поиск библиотек осуществляется по системному пути компилятора <i>../elcore30m-elvis-elf/lib</i> для Elcore-30M и по путям, указанным через опцию <i>–L</i> .
<i>-nodefaultlibs</i> <i>-nostdlib</i>	При линковке не использовать библиотеки, подключаемые по умолчанию. Для Elcore-30M подключаются <i>–lc –lcrt</i> . При использовании одной из опций в случае необходимости необходимо обеспечить замену функций из отключаемых библиотек. <i>–lc</i> – стандартная библиотека языка C. <i>–lcrt</i> – библиотека поддержки компилятора (преимущественно вычислительные операции). <i>–lunwind</i> – библиотека поддержки исключений C++.
<i>-nostartfiles</i>	Не использовать стандартный <i>crt0.s</i>
<i>-T <script_name></i>	Использовать скрипт линковки <i><script_name></i> вместо скрипта по

Ключ	Команда
	умолчанию. По умолчанию используются: для Elcore-30M - ..\elcore30m-elvis-elf\lib\generic.ld;

2.12. Переменные окружения компилятора

Списком директорий, используемых для поиска заголовочных файлов и библиотек, можно управлять, используя переменные окружения.

Таблица 2.12. - Принимаемые компилятором переменные окружения

Команда	Пояснение
C_INCLUDE_PATH	Переменная окружения для пути к заголовочным файлам C.
CPLUS_INCLUDE_PATH	Переменная окружения для пути к заголовочным файлам C++.
LIBRARY_PATH	Переменная окружения для пути к библиотечным файлам.

3. ВХОДНЫЕ ДАННЫЕ

Компилятор поддерживает языки C/C++ и их диалекты:

- поддержка диалектов ANSI C и ANSI C++ согласно стандартам языков C (ISO/IEC 9899:1990 (C89), ISO/IEC 9899:1999(C99)), C++ (ISO/IEC 14882:1998, ISO/IEC 14882:2003) , C++11 (ISO/IEC 14882:2011), C++14 соответственно;
- поддержка диалектов GNU C и GNU C++.

Компилируемая программа должна быть написана на языке C/C++ согласно стандарту. При необходимости стандарт выбирается через флаг `-std=` (см. табл. из раздела «Обращение к компилятору»). Помимо стандарта и диалектов GNU поддерживаются расширения языка C/C++ Clang и расширения языка C/C++, специфичные для архитектуры Elcore-30M.

3.1. Поддерживаемые расширения GNU C

Использование объявлений переменных внутри блоков

Например,

```
#define maxint(a,b) \
    ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

использование условного выражения с отсутствующим ветвлением (x? :y)

```
{int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Поддержка комплексных типов, встроенные ключевые слова `__Complex`, `__real__`, `__imag__`.

Поддержка операций комплексной арифметики: `+`, `-`, `*`, `/`, (комплексное сопряжение).

Например,

```
__complex__ float x = 1.0f+2.5fi;
__complex__ int x = 4-5i;
int reX = __real__ x;
int imX = __imag__ x;
__complex__ int conjX = ~x;
```

Массивы и структуры нулевой длины

```
struct line {
    int length;
    char contents[0];
};
```

Макросы с переменным числом аргументов

```
#define eprintf(format, args...) \
    printf (format , ## args)
```

Помеченные элементы в инициализаторах

```
int a[6] = { [4] 29, [2] = 15 };
```

эквивалентно

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

для инициализации диапазона значений

```
int width[] = {[0..9]=1, [10..99]=2, [100]=3};
```

Все пропущенные поля заполняются 0.

Диапазоны case

```
case 2..9
```

Приведение к типу объединения

```
union foo u;
```

```
...
```

```
u = (union foo) x == u.i = x
```

```
u = (union foo) y == u.d = y
```

Объявления атрибутов функции

Объявляется с использованием ключевого слова `__attribute__`. Возможные параметры `noreturn`, `const`, `format`, `section`, `constructor`, `destructor`, `unused`, `weak`, `alias`.

```
int square (int) __attribute__ ((const));
```

Объявление атрибутов переменных

Объявляется с использованием ключевого слова `__attribute__`. Возможные параметры: `aligned`, `mode`, `nocommon`, `packed`, `section("section_name")`, `transparent_union`, `unused`, `weak`.

Объявление атрибутов типов

Объявляется с использованием ключевого слова `__attribute__`. Возможные параметры: `aligned`, `packed`, `transparent_union`.

Векторные расширения

```
typedef int v4si __attribute__ ((vector_size (16)));
```

Встроенные арифметические операции над векторными типами: поэлементное сложение, умножение, вычитание, сравнение.

```
typedef int v4si __attribute__ ((vector_size (16)));
v4si a = {3,4,5,6};
v4si a = {1,2,3,4};
v4si c = a+b;
```

3.2. Расширения Clang C/C++

Встроенные функции для проверки допустимых возможностей

```
__has_builtin
#if __has_builtin(__builtin_trap)
__builtin_trap();
#else
abort();
#endif

__has_feature, __has_extension
```

Встроенные функции для проверки включенных заголовочных файлов и предупреждений

```
__has_include, __has_warning
#if __has_include("myinclude.h") && __has_include(<stdint.h>)
#include "myinclude.h"
#endif
```

```
#if __has_warning("-Wformat")
...
#endif
```

Встроенные макросы

```
__BASE_FILE__
```

Имя файла

```
__COUNTER__
```

Увеличивается на 1 при каждом вызове макроса

```
__INCLUDE_LEVEL__
```

Уровень включенного файла в иерархии включенных файлов

Векторные типы данных.

Поддерживаются gcc векторные расширения

Векторные константы

```
typedef int v4si __attribute__((__vector_size__(16)));
typedef float float4 __attribute__((__ext_vector_type(4)));
typedef float float2 __attribute__((__ext_vector_type(2)));

v4si vsi = (v4si){1, 2, 3, 4};
float4 vf = (float4){1.0f, 2.0f, 3.0f, 4.0f};
vector int vi1 = (vector int){1}; // vi1 will be (1, 1, 1, 1).
vector int vi2 = (vector int){1}; // vi2 will be (1, 0, 0, 0).
vector int vi3 = (vector int){1, 2}; // error
vector int vi4 = (vector int){1, 2}; // vi4 will be (1, 2, 0, 0).
vector int vi5 = (vector int){1, 2, 3, 4};
float4 vf = (float4)((float2){1.0f, 2.0f}, (float2){3.0f, 4.0f});
```

Встроенные векторные операции

[], +, -, ++, --, *, /, %, &, |, ^, ~, <<, >>, !, &&, ||, ==, !=, >, <, <=, >=, =, :?, sizeof

Все унарные и бинарные операции являются поэлементными

Перегрузка функций в языке C

```
#include <math.h>
float __attribute__((overloadable)) tabs(float x) { return absf(x); }
int __attribute__((overloadable)) tabs(int x) { return abs(x); }
```

Объявление комплексных переменных через список

```
#include <math.h>
#include <complex.h>
complex float x = { 1.0f, INFINITY }; // Init to (1, Inf)
```

Встроенные функции

Помимо списка встроенных функций gcc, предоставляются встроенные функции:

- 1) `__builtin_shufflevector(vec1, vec2, idx1, idx2)` – выполняет операцию shuffle;
- 2) `__builtin_convertvector(src_vec, dst_vec_type)` – выполняет преобразование вектора к указанному векторному типу;
- 3) `__builtin_addressof(expr)` – аналога унарного оператора вычисления адреса &
- 4) встроенные для арифметики с переносом.

```
unsigned char      __builtin_addcb (unsigned char x, unsigned char y, un-
signed char carryin, unsigned char *carryout);
unsigned short     __builtin_addcs (unsigned short x, unsigned short y, un-
signed short carryin, unsigned short *carryout);
unsigned           __builtin_addc  (unsigned x, unsigned y, unsigned carryin,
unsigned *carryout);
unsigned long      __builtin_addcl (unsigned long x, unsigned long y, un-
signed long carryin, unsigned long *carryout);
unsigned long long __builtin_addcll(unsigned long long x, unsigned long long
y, unsigned long long carryin, unsigned long long *carryout);
unsigned char      __builtin_subcb (unsigned char x, unsigned char y, un-
signed char carryin, unsigned char *carryout);
unsigned short     __builtin_subcs (unsigned short x, unsigned short y, un-
signed short carryin, unsigned short *carryout);
unsigned           __builtin_subc  (unsigned x, unsigned y, unsigned carryin,
unsigned *carryout);
unsigned long      __builtin_subcl (unsigned long x, unsigned long y, un-
signed long carryin, unsigned long *carryout);
unsigned long long __builtin_subcll(unsigned long long x, unsigned long long
y, unsigned long long carryin, unsigned long long *carryout);
```

- 5) встроенные функции для арифметики с переполнением

```
bool __builtin_uadd_overflow (unsigned x, unsigned y, unsigned *sum);
bool __builtin_uaddl_overflow (unsigned long x, unsigned long y, unsigned
long *sum);
bool __builtin_uaddll_overflow(unsigned long long x, unsigned long long y,
unsigned long long *sum);
bool __builtin_usub_overflow (unsigned x, unsigned y, unsigned *diff);
bool __builtin_usubl_overflow (unsigned long x, unsigned long y, unsigned
long *diff);
bool __builtin_usubll_overflow(unsigned long long x, unsigned long long y,
unsigned long long *diff);
```

```
bool __builtin_umul_overflow (unsigned x, unsigned y, unsigned *prod);
bool __builtin_umull_overflow (unsigned long x, unsigned long y, unsigned
long *prod);
bool __builtin_umulll_overflow(unsigned long long x, unsigned long long y,
unsigned long long *prod);
bool __builtin_sadd_overflow (int x, int y, int *sum);
bool __builtin_saddl_overflow (long x, long y, long *sum);
bool __builtin_saddll_overflow(long long x, long long y, long long *sum);
bool __builtin_ssub_overflow (int x, int y, int *diff);
bool __builtin_ssubl_overflow (long x, long y, long *diff);
bool __builtin_ssubll_overflow(long long x, long long y, long long *diff);
bool __builtin_smul_overflow (int x, int y, int *prod);
bool __builtin_smull_overflow (long x, long y, long *prod);
bool __builtin_smulll_overflow(long long x, long long y, long long *prod);
```

3.3. Список builtin-функций для архитектуры Elcore-30M

Компилятор поддерживает набор встроенных функций, которые соответствуют системе инструкций Elcore-30M. Использование builtin-функций по функциональности аналогично использованию ассемблерных вставок, но в некоторых вариантах использования более компактно и понятно для программиста. Builtin функции необходимо использовать при оптимизации кода, когда существующие стандартные механизмы оптимизации: использование опции -O3, автовекторизации и VLIW-пакетов не дают необходимой скорости вычисления.

При описании функций используется нотация

```
// short vectors
typedef __attribute__((__vector_size__(2 * sizeof(short)))) short v2i16;
typedef __attribute__((__vector_size__(4 * sizeof(short)))) short v4i16;
typedef __attribute__((__vector_size__(8 * sizeof(short)))) short v8i16;

// int vectors
typedef __attribute__((__vector_size__(2 * sizeof(int)))) int v2i32;
typedef __attribute__((__vector_size__(4 * sizeof(int)))) int v4i32;

// int64 vectors
typedef __attribute__((__vector_size__(2 * sizeof(long long)))) long long
v2i64;

typedef __attribute__((__vector_size__(2 * sizeof(float)))) float v2f32;
typedef __attribute__((__vector_size__(4 * sizeof(float)))) float v4f32;
```

Ограничение: При использовании векторных builtin-функций накладывается ограничение на выравнивание входных аргументов. Статическое приведение к типу v2i16, v4i16, v8i16, v2i32, v4i32, v2i64, v2f32, v4f32 должно выполняться только от соответствующим образом выровненного указателя. Компилятор не проверяет необходимое выравнивание.

Например, дальнейшее использование указателя pV4I16SomeData в builtin-функции может привести к ошибке

```
short SomeData[128] = {1};
v4i16 *pV4I16SomeData = (v4i16*) SomeData;
```

Правильно указывать атрибут выравнивания


```
short SomeData[128] __attribute__((aligned(8))) = {1};
v4i16 *pV4I16SomeData = (v4i16*) SomeData;
```

Таблица 3.1. - Список builtin-функций

Ин- струк- ция DSP	Нотация builtin-функции	Описание
A8	v8i16 __builtin_v8i16_add (v8i16 a, v8i16 b)	Восемь сложений (short)
S8	v8i16 __builtin_v8i16_sub (v8i16 a, v8i16 b)	Восемь вычитаний (short)
MS8	v8i16 __builtin_v8i16_subabs (v8i16 a, v8i16 b)	Модули восьми разностей (short)
A81	short __builtin_v8i16_sum (v8i16 a)	два вектора из восьми чисел в формате short
A42	v2i16 __builtin_v8i16_sum 42 (v8i16 a)	Два сложения по четыре (short)
A24	v4i16 __builtin_v8i16_sum 24 (v8i16 a)	Четыре сложения по два (short)
A4	v4i16 __builtin_v4i16_add (v4i16 a, v4i16 b)	Четыре сложения (short)
S4	v4i16 __builtin_v4i16_sub (v4i16 a, v4i16 b)	Четыре вычитания (short)
M4	v4i32 __builtin_v4i16_mul (v4i16 a, v4i16 b)	Четыре умножения (short)
MS4	v4i16 __builtin_v4i16_subabs (v4i16 a, v4i16 b)	Модули четырех разностей (short)
MS2	v2i16 __builtin_v2i16_subabs (v2i16 a, v2i16 b)	Модули двух разностей (short)
M2	v2i32 __builtin_v2i16_mul (v2i16 a, v2i16 b)	Два умножения (short)
ALL4	v4i32 __builtin_v4i32_add (v4i32 a, v4i32 b)	Четыре сложения (long)
SLL4	v4i32 __builtin_v4i32_sub (v4i32 a, v4i32 b)	Четыре вычитания (long)
ALL41	int __builtin_v4i32_sum (v4i32 a)	Сложение по четыре (long)
ALL2	v2i32 __builtin_v2i32_add (v2i32 a, v2i32 b)	Два сложения (long)
SLL2	v2i32 __builtin_v2i32_sub (v2i32 a, v2i32 b)	Два вычитания (long)
ML2	v2i64 __builtin_v2i32_mul (v2i32 a, v2i32 b)	Два умножения (long)
UML2	v2i64 __builtin_v2i32_umul (v2i32 a, v2i32 b)	Два умножения (long)
AL4	v4i32 __builtin_v4i16_32_add (v4i16 a, v4i32 b)	Четыре сложения операндов разного формата (short, long)
AL2	v2i32 __builtin_v2i16_32_add (v2i16 a, v2i32 b)	Два сложения операндов разного формата (short, long)
MAX8	v4i16 __builtin_v8i16_max (short, v8i16 a, v4i16 b)	Поиск максимума и его номера
MIN8	v4i16 __builtin_v8i16_min (short, v8i16 a, v4i16 b)	Поиск минимума и его номера
MAX4	v4i16 __builtin_v4i16_max (short, v4i16 a, v4i16 b)	Поиск максимума и его номера
MIN4	v4i16 __builtin_v4i16_min (short, v4i16 a, v4i16 b)	Поиск минимума и его номера
MAXL4	v2i32 __builtin_v4i32_max (short, v4i32 a, v2i32 b)	Поиск максимума и его номера
MINL4	v2i32 __builtin_v4i32_min (short, v4i32 a, v2i32 b)	Поиск минимума и его номера
MAXL2	v2i32 __builtin_v2i32_max (short, v2i32 a, v2i32 b)	Поиск максимума и его номера
MINL2	v2i32 __builtin_v2i32_min (short, v2i32 a, v2i32 b)	Поиск минимума и его номера
FA4	v4f32 __builtin_v4f32_add (v4f32 a, v4f32 b)	Четыре сложения (float)
FS4	v4f32 __builtin_v4f32_sub (v4f32 a, v4f32 b)	Четыре вычитания (float)
FM4	v4f32 __builtin_v4f32_mul (v4f32 a, v4f32 b)	Четыре умножения (float)

Ин-струк-ция DSP	Нотация builtin-функции	Описание
FM4C	v4f32 __builtin_cv4f32_mul (float c, v4f32 a)	Четыре умножения на общую константу, (float)
FM2	v2f32 __builtin_v2f32_mul (v2f32 a, v2f32 b)	Два умножения (float)
FMS2	v2f32 __builtin_v2f32_mulreverse (v2f32 a, v2f32 b)	Два умножения с перестановкой (float)
FSA	v2f32 __builtin_v2f32_addsub (v2f32 a, v2f32 b)	Вычитание и сложение (float)
FIN4	v4f32 __builtin_v4f32_recip (v4f32 b)	Четыре нулевых приближения к обратной величине
FINR4	v4f32 __builtin_v4f32_recipsqrt (v4f32 b)	Четыре нулевых приближения к обратной величине от квадратного корня
FASX	v4f32 __builtin_cmplx_faddsub (v2f32 a, v2f32 b)	Сложение и вычитание комплексных операндов (float)
FASXS	v4f32 __builtin_cmplx_faddsubconj (v2f32 a, v2f32 b)	Сложение и вычитание комплексных операндов с перестановкой (float)
FAX	v2f32 __builtin_cmplx_fadd (v2f32 a, v2f32 b)	Сложение комплексных операндов (float)
FSX	v2f32 __builtin_cmplx_fsub (v2f32 a, v2f32 b)	Вычитание комплексных операндов (float)
ASX2	v8i16 __builtin_v4i16_asx2 (v4i16 a, v4i16 b)	Два сложения и вычитания комплексные (X16)
ASXS2	v8i16 __builtin_v4i16_asxs2 (v4i16 a, v4i16 b)	Два сложения и вычитания комплексных с перестановкой (X16)
ASXS	v4i16 __builtin_v2i16_asxs (v2i16 a, v2i16 b)	Сложение и вычитание комплексные с перестановкой (X16)
RA4	v4i16 __builtin_v4i16_ra4 (v4i16 a, v4i16 b)	Четыре скользящие суммы (short)
RA8	v8i16 __builtin_v8i16_ra8 (v8i16 a, v8i16 b)	Восемь скользящих сумм (short)
SGA4	v4i16 __builtin_v4i16_sga4 (short, v4i16 a, v4i16 b)	Четыре знаковых сумм (short)
SGA8	v8i16 __builtin_v8i16_sga8 (short, v8i16 a, v8i16 b)	Восемь знаковых сумм (short)
BF4	v8i16 __builtin_v4i16_bf4 (v4i16 a, v4i16 b)	Базовая операция FFT-4 (X16)
BIF4	v8i16 __builtin_v4i16_bif4 (v4i16 a, v4i16 b)	Базовая операция IFFT-4 (X16)
AXJ4	v8i16 __builtin_v8i16_axj4 (v8i16 a, v8i16 b)	Четыре комплексных сложения с предварительным умножением одного из операндов на мнимую единицу (X16)
SXJ4	v8i16 __builtin_v8i16_sxj4 (v8i16 a, v8i16 b)	Четыре комплексных сложения с предварительным умножением одного из операндов на мнимую единицу (short)

Примеры программ, иллюстрирующих применение builtin-функций, приведены в приложении 1.

4. АССЕМБЛЕРНЫЕ ВСТАВКИ

Ассемблерные вставки используются для встраивания в Си-программы ассемблерного кода, явно заданного программистом. Содержимое ассемблерной вставки никак компилятором не анализируется, но имеется возможность описать то, как это содержимое взаимодействует с переменными Си-программы и как изменятся регистры после выполнения этого ассемблерного кода.

Использование ассемблерных вставок может реализовать наиболее критичные в отношении производительности части алгоритма. Это позволяет программисту не ограничиваться конструкциями компилятора.

Для объявления ассемблерных вставок в языке C используется встроенная функция `asm()`. Общая структура ассемблерной вставки выглядит следующим образом:

```
asm volatile ( "code" : outputs : inputs : clobbers );
```

где `code` - ассемблерная вставка;

`outputs` – список выходных операндов;

`inputs` – список входных операндов;

`clobbers` – список изменяемых параметров;

`volatile` – ключевое слово `volatile`, необходимо использовать при описании ассемблерной вставки для явного указанию компилятору на запрет оптимизации области кода с ассемблерной вставкой.

Пример использования ассемблерной вставки показан ниже.

Пример 1

```
int main()
{
    int result=0, in1=10, in2=35;
    asm volatile (
        "maxml %1, %2, %0"           // ассемблерная вставка
        : "=r"(result)              // выходной операнд
        : "r"(in1), "r"(in2)        // входные операнды
    );
    return 0;
}
```

Существует и более короткая форма:

```
asm volatile ("code");           (см. пример 2)
```

Пример 2

```
asm volatile ("move dcsr, r11"); //пересылка из регистра управления
DCSR в регистр данных
```

Ключевое слово `volatile` служит для того, чтобы указать компилятору, что вставляемый ассемблерный код может давать побочные эффекты, поэтому попытки оптимизации могут привести к логическим ошибкам. Оптимизатор пытается переупорядочить и переписать код программы с целью минимизации времени исполнения даже в том случае, когда в программе имеются ассемблерные вставки. Если между инструкцией `asm` и ее операндами отсутствует ключевое слово `volatile`, то когда оптимизатор обнаруживает, что результат исполнения `asm`-инструкции нигде не используется, он может просто исключить ее из текста программы. Любая отдельно взятая ассемблерная вставка может быть перемещена со своего места, и крайне трудно угадать заранее, как ею распорядится оптимизатор. Поэтому необходимо всегда указывать слово `volatile` после `asm`.

Область “code” представляет собой строковую константу с ассемблерными инструкциями. Если необходимо написать ассемблерную вставку с несколькими инструкциями, нужно использовать перевод строки и символ табуляции “`\n\t`” в явном виде в тексте вставки. Иначе в сгенерированном ассемблерном файле все строки “склеятся” в одну (см. пример 3).

Пример 3

```
asm volatile (                                     // result3 = in5*2.7+in4
"move %0, r2.l\n\t"
"move %1, r4.l\n\t"
"fmpy 2.7, r2.l, r6.l\n\t"
"fadd r4.l, r6.l, %2\n\t"
:"=r"(result3)                                   // выходные операнды
:"r"(in4), "r"(in5)                             // входные операнды
);
```

В теле вставки могут находиться не только ассемблерные инструкции, но и любые директивы, распознаваемые ассемблером `elcore-elvis-elf-as`. Пример 4 использует ассемблерные инструкции и директивы.

Пример 4

```
asm volatile (
".rept 3\n\t"                                     //циклическое выполнение инструкций между .rept и .endr
3 раза
" move pc, r2\n\t"
" move r2, (a0)+\n\t"
".endr \n\t"
);
```

Область входных и выходных операндов используются для связи ассемблерных инструкций с переменными Си-программы. Если нет выходных операндов, но есть входные, то нужно писать два символа ‘:’ перед списком входных операндов.

Каждый описанный операнд может использоваться в ассемблерных инструкциях, обращение к нему осуществляется по номеру с префиксом `%`. Нумерация начинается с 0, и идет непрерывно, объединяя все элементы списков выходных и входных операндов. Номера присваиваются в порядке объявления операндов, если один из них ссылается на другой, то он пропускается. `%0`

относится к первому операнду (обычно выходному), %1 ко второму, и далее до %9. В примере 1 ассемблерная вставка использует операнды 0, 1 и 2, где %0 относится к выходному операнду "=r"(result), а %1 и %2 к входным операндам "r"(in1) и "r"(in2) соответственно.

Синтаксис у входного и выходного списков операндов одинаковый — перечисление описаний операндов, разделённых запятой. Описание операнда в общем случае имеет следующий вид:

"constraints" (value) или "Ограничение типа" (значение операнда)

Значение операнда (value) – это имя Си-переменной, значение которой используется в ассемблерной вставке. Для выходного операнда выражение должно быть lvalue.

Ограничение типа (например "=r") – строковая константа, которая описывает допустимый тип операнда и состоит из ограничителя и модификатора. Ограничитель описывается буквой и указывает компилятору, каким условиям должен соответствовать данный операнд. Возможные ограничения:

- "r" обозначает любой регистр данных. Размер регистра выбирается компилятором автоматически исходя из размера данных. (См. пример 4.3). При использовании ограничения "r" необходимо помнить, что компилятор не преобразовывает ассемблерную вставку и для архитектуры Elcore30m невозможно явно передать в ассемблерную вставку 8-битную переменную.
- "rm" обозначает, что операнд будет помещен в память (см. пример 5) или в регистр данных.
- "ri" обозначает непосредственный операнд или регистр данных.

```
int in1=2, in2=3, result;
asm volatile ("addl %1, %2, %0" : "=r"(result) : "ri"(in1), "r"(in2));
```

преобразуется в

```
addl 0x2, r0.l, r1.l
```

- "rm" (или "g"- "general"-общий) обозначает либо регистр данных, либо непосредственный операнд, либо память.

Модификатор - это символ, который при необходимости добавляется перед ограничителем. Операнд без модификатора трактуется как «read-only»(это для входных операндов), модификатор '=' делает операнд «write-only»(для выходных операндов), а модификатор '&' говорит, что операнд используется только как выход. Можно также использовать один и тот же регистр для выхода и входа. Для этого в ограничителе входного операнда пишется цифра 0. Если нужен ещё регистр, используемый для выхода и входа, то в ограничителе следующего входного операнда пишется цифра 1 и т.д (см. пример 7).

Область Clobbers - список изменяемых регистров, может быть опущен в ассемблерной вставке. Список изменяемых регистров содержит имена регистров, которые используются на запись в ассемблерной вставке, но не объявлены в списках операндов. Эти регистры, в случае необходимости, будут помещены в стек перед входом во вставку и выгружены обратно после выхода. Кроме того, может быть указано специальное слово «memory» в clobbers, которое помимо указания компилятору, что ассемблерная вставка изменяет содержимое памяти, означает, что те операции обращений в память, которые стоят выше по коду, в результирующем машинном коде будут выполняться до тех, которые стоят ниже ассемблерной вставки. Следует избегать использования clobbers, так как это сильно стесняет свободу оптимизатора. Например, временные переменные лучше объявлять вне ассемблерной вставки — это позволит компилятору самому выбирать регистры для их хранения.

В области clobbers возможно указать регистры данных r0-lr31.l и регистры адресного генератора AGU (a0-a7, i0-i7, m0-m7).

5. ОБРАЩЕНИЕ К ВНЕШНЕЙ ПАМЯТИ

Архитектура DSP-ядра ELcore-30M позволяет выполнять обращение не только к внутренней памяти XYRAM, но и к внешней по физическому адресу.

Компилятор в явном виде не поддерживает чтение и запись по внешним адресам. Код

```
int value = (*(volatile int*)addr)
```

где `addr` – адрес ячейки внешней памяти, выполнится некорректно.

При обращении к внешней памяти из DSP необходимо учитывать архитектурные ограничения

- 1) При обращении к внешней памяти необходимо виртуальный адрес переводить в физический
- 2) Адресация к внешней памяти также словная
- 3) Внешние адреса с `0x0000` до `0xffff` включительно не поддерживаются, т.к. перекрываются с внутренними адресами DSP-памяти

Для корректного обмена с внешней памятью необходимо использовать функции `readFromExtMem`, `writeToExtMem`, которые учитывают указанные архитектурные особенности.

```
#define PADDR(a) (a&0x7fffffff)

/*
 * Function: int readFromExtMem(int addr)
 * Description: loads value from external memory.
 * Input: addr - address of external memory, must be aligned at 4
 * Output: int - loaded value
 */
int readFromExtMem(int addr)
{
    int val = 0;
    addr = PADDR(addr);

    asm volatile("lsrl 2, %0, r6.l"::"r"(addr));
    asm volatile("move r6.l, a5.l"::"a5.l");
    asm volatile("move (a5.l), r6.l");
    asm volatile("move r6.l, %0"::"r"(val));

    return val;
}

/*
 * Function: void writeToExtMem(int addr, int value)
 * Description: loads value into external memory.
 * Input: addr - address of external memory, must be aligned at 4
 *        value - value to store
 */
void writeToExtMem(int addr, int value)
{
    addr = PADDR(addr);

    asm volatile("lsrl 2, %0, r6.l"::"r"(addr));
    asm volatile("move r6.l, a5.l"::"a5.l");
    asm volatile("move %0, (a5.l)"::"r"(value));
}
```

1. ПРИЛОЖЕНИЕ 1. Примеры программ с builtin-функциями

1.1. ПРИМЕР 1. Бинарные операции над массивами short

```
// найти sum0 = a0+b0+a1+b1+...+a31+b31 (массивы short a[32] и short b[32])
// найти sum1 = (a0-b0)+(a1-b1)+...+(a31-b31)
// найти sum2 = |a0-b0|+|a1-b1|+...+|a31-b31|
// найти sum3 = c0*d0+c1*d1+...+c31*d31 (массивы short c[32] и short d[32])

#include "attribute.h"
short a[32] __attribute__((aligned(16))) = {1};
short b[32] __attribute__((aligned(16))) = {2};
short c[32] __attribute__((aligned(16))) = {3};
short d[32] __attribute__((aligned(16))) = {4};

int sum0=0;
int sum1=0;
int sum2=0;
int sum3=0;

int main()
{
    int i, s0, s1, s2, s3;
    v8i16 *a1 = (v8i16*)a;
    v8i16 *b1 = (v8i16*)b;
    v4i16 *c1 = (v4i16*)c;
    v4i16 *d1 = (v4i16*)d;

    for (i = 0; i < 4; i++)
    {
        v8i16 add = __builtin_v8i16_add(a1[i],b1[i]);
        s0 = __builtin_v8i16_sum(add);
        sum0 = sum0 + s0;

        v8i16 sub = __builtin_v8i16_sub(a1[i],b1[i]);
        v2i16 s1 = __builtin_v8i16_sum42(sub);
        sum1 = sum1 + s1[0] + s1[1];

        v8i16 subabs = __builtin_v8i16_subabs(a1[i],b1[i]);
        v4i16 s2 = __builtin_v8i16_sum24(subabs);
        sum2 = sum2 + s2[0] + s2[1] + s2[2] + s2[3];
    }
    for (i = 0; i < 8; i++)
    {
        v4i32 mul = __builtin_v4i16_mul(c1[i],d1[i]);
        s3 = __builtin_v4i32_sum(mul);
        sum3 = sum3 + s3;
    }
    return 0;
}
```


1.2. ПРИМЕР 2. Бинарные операции над массивами int

```
// найти sum0 = a0+b0+a1+b1+...+a31+b31 (массивы int a[32] и int b[32])
// найти sum1 = (c0-d0)+(c1-b1)+...+(c31-d31) (массивы int c[32] и int d[32])

#include "attribute.h"

int a[32] __attribute__((aligned(16))) = {1};
int b[32] __attribute__((aligned(16))) = {2};
int sum0=0;
int sum1=0;
int main()
{
    int i, s0, s1;
    v4i32 *a1 =(v4i32*)a;
    v4i32 *b1 =(v4i32*)b;
    v2i32 *c1 =(v2i32*)c;
    v2i32 *d1 =(v2i32*)d;
    for (i = 0; i < 8; i++)
    {
        v4i32 add = __builtin_v4i32_add(a1[i],b1[i]);
        s0 = __builtin_v4i32_sum(add);
        sum0 = sum0 + s0;
    }
    for (i = 0; i < 16; i++)
    {
        v2i32 sub = __builtin_v2i32_sub(c1[i],d1[i]);
        sum1 = sum1 + sub[0] + sub[1];
    }
    return 0;
}
```

1.3. ПРИМЕР 3. Поиск минимума, максимума и их индексов в массиве

```
// найти в массиве из 200 чисел (50 векторов из четырех чисел в формате
short) максимальное число и его номер, минимальное число и его номер
#include "attribute.h"

short x[200] __attribute__((aligned(8))) = {336,-968,-796,-271,444,-680,-693,-
635,-828,576,-793,-24,250,359,-908,998,727,-950,-134,-10,762,583,891,-764,-
257,-601,607,745,554,786,-16,723,-916,660,402,299,546,-284,398,843,-23,-737,-
847,-618,-537,908,-361,738,213,-93,46,-654,-967,-893,171,-77,-174,-219,-
621,431,649,796,587,479,-518,648,-655,399,983,-184,512,354,-403,987,-
447,996,-150,-437,732,-500,-86,22,-4,517,766,808,771,859,304,57,10,959,225,-
170,-45,935,-718,174,937,-728,-784,417,441,-24,753,776,258,202,434,-
57,201,473,704,963,-801,223,475,-941,-997,-467,-860,262,62,-929,-7,-549,-
675,-428,-770,941,-498,-878,578,-285,773,-385,653,-820,546,-148,107,-
184,437,682,196,-682,329,399,-447,-864,490,-725,-
204,964,782,725,274,130,490,-793,561,-822,297,112,-486,-436,909,765,254,28,-
371,253,-628,-657,-872,983,-500,-47,-520,-562,753,206,904,-754,469,-252,839,-
916,-740,228,-961,992,755,-866,-237,368,87,702,801,8};

v4i16 result0 = {0x8000,0,0,0};
v4i16 result1 = {0x7fff,0,0,0};
int main()
{
    short i, pos = 0;
    v4i16 *x1 = (v4i16*)x;
    for (i = 0; i < 50; i++)
    {
        result0 = __builtin_v4i16_max(pos, x1[i], result0);
        result1 = __builtin_v4i16_min(pos, x1[i], result1);
        pos = pos+4;
    }
    return pos;
}
// максимальное число 998 с номером 15, минимальное число -997 с номером 118
```

1.4. ПРИМЕР 4. Бинарные операции над массивами float

```
// найти sum0 = a0+b0+a1+b1+...+a31+b31 (массивы float a[32] и float b[32])
// найти sum1 = (c × a0 - b0) + (c × a1 - b1) +... + (c × a31 - b31) (массивы
float a[32] и float b[32], константа float c)
// найти sum2 = a0×b0 + a1×b1 +...+ a31×b31 (массивы float a[32] и float
b[32])

#include "attribute.h"

float __extractel_v4f32(v4f32, int);
float a[32] __attribute__((aligned(16))) = {105.33, 109.76, 183.33,
164.40, 81.99, 111.39, 75.72, 192.85, 71.22, 165.73, -18.31, 81.05, 186.86,
103.75, -0.36, -3.14, 171.63, 67.68, 174.74, 24.05, 97.00, 73.53, 94.65, -
14.31, 48.52, 1.47, 35.45, 66.68, 29.45, 21.11, 149.33, 116.68};

float b[32] __attribute__((aligned(16))) = {106.25, 61.06, 160.92, 130.14,
130.09, 118.60, 73.40, 28.90, 106.87, 191.85, 158.38, 93.36, 50.87, 149.11,
96.55, 45.45, 35.07, -15.96, 32.27, 57.29, 16.70, 39.19, 148.09, -16.30, -
13.14, 39.50, 103.47, 92.29, -12.33, 49.08, 115.84, 136.69};

float sum0 = 0;
float sum1 = 0;
float sum2 = 0;

int main()
{
    int i;
    float s, c = 2.8;
    v4f32 *a1 = (v4f32*)a;
    v4f32 *b1 = (v4f32*)b;

    for (i = 0; i < 8; i++)
    {
        v4f32 add = __builtin_v4f32_add(a1[i], b1[i]);
        sum = sum + __extractel_v4f32(add, 0) + __extractel_v4f32(add, 1)
+ __extractel_v4f32(add, 2) + __extractel_v4f32(add, 3);

        v4f32 n = __builtin_cv4f32_mul(c, a1[i]);
        v4f32 sub = __builtin_v4f32_sub(n, b1[i]);
        sum1 = sum1 + __extractel_v4f32(sub, 0) + __extractel_v4f32(sub,
1) + __extractel_v4f32(sub, 2) + __extractel_v4f32(sub, 3);

        v4f32 mul = __builtin_v4f32_mul(a1[i], b1[i]);
        sum2 = sum2 + __extractel_v4f32(mul, 0) + __extractel_v4f32(mul,
1) + __extractel_v4f32(mul, 2) + __extractel_v4f32(mul, 3);
    }
    return 0;
}
```

1.5. ПРИМЕР 5. Бинарные операции над массивами комплексных чисел

```
// найти sum0 = a0+b0+a1+b1+...+ a30+b30 + a31+b31 (массивы float a[32] и
float b[32])
// найти sum1 = a0-b0+a1-b1+...+ a30-b30 + a31-b31 (массивы float a[32] и
float b[32])
// найти sum2 = a1-b1+a0-b0+a1+b1+a0+b0+...+ a31+b31 + a30+b30 (массивы float
a[32] и float b[32])
// найти sm = (a1-b0)-(a0+b1)-(a1+b0)-(a0-b1)-...- (a31+b30) - (a30-b31)
(массивы float a[32] и float b[32])

#include "attribute.h"

float __extractel_v4f32(v4f32, int);
float a[32] = {1.0f};
float b[32] = {2.0f};
float sum0 = 0;
float sum1 = 0;
float sum2 = 0;
float sm = 0;

int main()
{
    int i;
    v2f32 *a1 = (v2f32*)a;
    v2f32 *b1 = (v2f32*)b;

    for (i = 0; i < 16; i++)
    {
        v2f32 fadd = __builtin_cmplx_fadd(a1[i], b1[i]);
        sum0 = sum0 + fadd[0] + fadd[1];

        v2f32 fsub = __builtin_cmplx_fsub(a1[i], b1[i]);
        sum1 = sum1 + fsub[0] + fsub[1];

        v4f32 fadds = __builtin_cmplx_faddsub(a1[i], b1[i]);
        sum2 = sum2 + __extractel_v4f32(fadds, 0) +
        __extractel_v4f32(fadds, 1) + __extractel_v4f32(fadds,
        2) + __extractel_v4f32(fadds, 3);

        v4f32 faddsc = __builtin_cmplx_faddsubconj(a1[i], b1[i]);
        sm = sm - __extractel_v4f32(faddsc, 0) -
        __extractel_v4f32(faddsc, 1) - __extractel_v4f32(faddsc, 2) -
        __extractel_v4f32(faddsc, 3);
    }
    return 0;
}
```

Версия документа	
2.01	Редактирование всего документа. Отделение соглашения о вызовах в отдельный файл.
2.02	В раздел «3.3 Список builtin-функций» внесено ограничение на выровненность указателей при преобразовании к типам v2i16, v2i32, и т.д. В приложении 1 соответствующим образом исправлены примеры.
2.03 (4.05.2016)	Добавлен раздел «5. Обращение к внешней памяти.» В раздел «4. Ассемблерные вставки» рекомендация об использовании volatile с ассемблерной вставкой указана как необходимая, добавлен список поддерживаемых регистров в области clobbers
2.04 (20.12.2016)	В разделе 4 добавлено уточнение о невозможности использования 8-битных переменных в ассемблерной вставке