ИНСТИТУТ ЯДЕРНОЙ ФИЗИКИ
им. Г.И. Будкера СО РАН

В.Р. Козак

БИБЛИОТЕКА РАДИОИНЖЕНЕРА
СПРАВОЧНИК ПО ЯЗЫКУ АБЕЛЬ
(Информационно-справочный материал)

НОВОСИБИРСК

БИБЛИОТЕКА РАДИОИНЖЕНЕРА

СПРАВОЧНИК ПО ЯЗЫКУ АБЕЛЬ

(Информационно-справочный материал)

Институт ядерной физики им. Г.И. Будкера
630090, Новосибирск 90, Россия

АННОТАЦИЯ

Издание предназначено для разработчиков радиоэлектронной аппаратуры. Пособие представляет из себя справочник по языку АБЕЛЬ- системы проектирования программируемых логических устройств.

Институт ядерной физики им.Г.И.Буджера СО РАН

СОДЕРЖАНИЕ

ГЛАВА 1 ВВЕДЕНИЕ

The Advanced Boolean Expression Language, ABEL, предназначен для использования при программировании и тестировании программируемых логических устройств таких как ППЗУ, ПЛМ и ПМЛ. Логический проект описывается на этом специализированном языке и языковый процессор ABEL транслирует это описание в формат, который может быть загружен непосредственно в программатор. Затем программатор использует загруженную информацию чтобы запрограммировать и протестировать реальное устройство.

ABEL обеспечивает особенности и конструкции языков высокого уровня, а также особенности, специфичные для логического проектирования:

* Гибкость: Разработчик может выбрать наиболее подходящий тип логического описания для ввода. Описание может выполнено, используя:
  - Уравнения Буля;
  - Таблицы истинности;
  - Диаграммы состояний.
* Логическая редукция.
* Моделирование устройства.
* Расширенные сообщения о синтаксических и логических ошибках.
* Конструкции структурного программирования: CASE, IF-THEN-ELSE.
* Макросы и директивы.

Это руководство детально описывает элементы и структуру языка ABEL. Оно сформировано как полная спецификация языка для новичка, так и как быстрый справочник для опытного разработчика. Руководство разделено на 4 основных главы, как показано в таблице 1-1 ниже.

Таблица 1-1. Главы и содержимое справочника по языку

Глава	Заглавие	Описание
1	Введение	Введение в руководство
2	Элементы	Описывает основные элементы языка, такие как ключевые слова, константы, выражения, уравнения и наборы (группы).
3	Структура	Объясняет как различные элементы встраиваются в различные структуры логического описания АБЕЛя.
4	Директивы	Обсуждает директивы, которые управляют обработку исходного файла.

1.1. ДРУГИЕ РУКОВОДСТВА ПО ЯЗЫКУ ABEL

Справочник по языку ABEL является одним из трех документов, описывающих ABEL. Другими двумя руководствами являются:

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ ABEL и
РУКОВОДСТВО ПО ПРИМЕНЕНИЮ ABEL

Первое из них описывает как использовать языковый процессор ABEL, чтобы преобразовать логическое описание в загрузочный файл программатора. Описана процедура установки языкового процессора и дан простой урок. Описываются в деталях все режимы процессора и объясняются выходные файлы.

Второе руководство предоставляет информацию и примеры, чтобы помочь разработчику, использующему ABEL. Даются спецификации, методы проектирования и полностью исходные файлы для типичной разработки с программируемой логикой. Представлен усложненный проект. Даны предложения, касающиеся использования АБЕЛя.

1.2. СОГЛАШЕНИЯ ОБ ОБОЗНАЧЕНИЯХ

Нижеследующая таблица (1-2) показывает соглашения об обозначениях, используемых в определениях и синтаксических описаниях в этом руководстве.

Таблица 1-2. Соглашение об обозначениях

Обозначение	Использование
Кавычки (")	Обрамляют ссылки по имени на объекты, содержащиеся в примерах, рисунках, листингах и таблицах.
БОЛЬШИЕ БУКВЫ	В синтаксических описаниях и диаграммах отображают, что слово или буква должна быть введена полностью. Сам объект может быть введен либо большими, либо маленькими буквами. Кроме этого, большими буквами выделяются ключевые слова.
маленькие буквы	В синтаксических описаниях и диаграммах означают, что вместо маленьких букв должно быть введено имя или значение, которое может вводиться как большими, так и маленькими буквами.
Квадратные скобки []	Окружают дополнительные параметры, которые могут присутствовать или быть опущены в зависимости от необходимости.
Многоточие	Обозначает, что предыдущий объект может быть при необходимости повторен.
Все другие знаки пунктуации	Апострофы, скобки, точки, запятые и прочие символы должны вводиться точно так, как показано.

Пример использования кавычек в тексте:
Если ключевое слово было введено как "Mod ule", оно может быть интерпретировано как два идентификатора, "Mod" и "ule".

Пример использования в тексте больших и маленьких букв, квадратных скобок и многоточий, так как они используются в описании синтаксиса:

MODULE modname [(arg[,arg]...)]

Это означает, что ключевое слово MODULE должно быть введено, но оно может быть введено как большими, так и маленькими буквами. Имя для модуля должно быть введено на месте "modname". Аргументы могут не быть, но если они присутствуют, то должны быть заключены в скобки. Ниже приведен правильный пример:

module ADDER (Carryin,add1,ADD2)

Определение синтаксиса АБЕЛя дается с помощью синтаксических описаний. Синтаксическое описание содержит полное и формальное определение синтаксиса языковых конструкций и определения индивидуальных частей, которые составляют конструкцию. Первая строка синтаксического описания (выделена жирным шрифтом) определяет синтаксис в соответствии с нотационными соглашениями, перечисленными выше. Строки, следующие за первой, определяют элементы этого синтаксиса.

ГЛАВА 2 ЭЛЕМЕНТЫ ЯЗЫКА

Эта глава описывает различные элементы языка проектирования ABEL. Эти элементы языка комбинируются в соответствии со структурой, описанной в главе 3, чтобы создать логическое описание АБЕЛя. На последующих страницах представлен каждый из элементов. Организация изложения- от простого к сложному.

2.1. ОСНОВЫ СИНТАКСИСА

Каждая строка исходного абелевского файла должна удовлетворять следующим синтаксическим правилам и ограничениям:

1. Строка может иметь длину до 131 символа.

2. Строки завершаются символом LINE FEED (шестнадцатеричное 0A), вертикальным табулятором (шестнадцатеричное 0B), или FORM FEED (шестнадцатеричное 0C). Возврат каретки в строке будет проигнорирован, адаптируясь таким образом к общепринятой последовательности завершения строки RETURN/LINE FEED. На большинстве компьютеров входная строка завершается просто нажатием клавиши RETURN или ENTER.

3. Ключевые слова, идентификаторы и числа, должны быть отделены друг от друга по крайней мере одним пробелом. Исключением из этого правила является список идентификаторов, разделенный запятыми, или выражения, где идентификаторы или числа разделены операторами, или места, где разделение обеспечивается скобками.

5. Пробелы не могут вставлены в середину слов, чисел, операторов или идентификаторов. Пробелы могут появляться в строках, комментариях, блоках и фактических аргументах. Например, если ключевое слово MODULE введено как "MOD ULE", оно будет интерпретировано как два идентификатора, MOD и ULE. Аналогично, если вы введете "102 05", подразумевая число 10205, оно будет интерпретировано как два числа, 102 и 5.

5. Ключевые слова (слова определенные как часть языка и слова имеющие специальное значение) могут быть напечатаны как маленькими, так и большими буквами, с одинаковым эффектом.

6. Идентификаторы (имена и метки определенные пользователем) могут быть напечатаны большими и маленькими буквами, а также их смесью, при их распознавании учитывается и их размер: идентификатор "output", напечатанный маленькими буквами не является тем же самым, что и идентификатор "Output", с большой буквой "O".

2.2. ИСПОЛЬЗУЕМЫЕ СИМВОЛЫ ASCII

Все большие и маленькие буквы, а также большинство других символов на клавиатуре, являются значащими (используемыми), см. таблицу 2-1.

Таблица 2-1. Используемые символы ASCII

a-z	маленькие буквы
A-Z	большие буквы
0-9	цифры
пробел	
табулятор	
! @ # $ % ^ & * ( ) - _ = + [{] }	
: ; ' " ` ~ \ | , . / ?	

2.3. ИДЕНТИФИКАТОРЫ

Идентификаторами являются имена, которые идентифицируют устройства, выводы или узлы устройства, группы, входные или выходные сигналы, константы, макросы и фиктивные параметры. Все эти понятия определяются в главе 2. Правила и ограничения для идентификаторов те же самые, не зависимо от того что этот идентификатор описывает.

Правила, распространяющиеся на идентификаторы, следующие:

1. Идентификаторы должны начинаться с символа из алфавита или подчеркивания.

2. Идентификаторы могут быть до 31 символа длиной. Все что длиннее, чем 31 символ, считается ошибкой и отмечается языковым процессором.

3. Все символы кроме первого (см. пункт 1) в идентификаторе могут быть большими и маленькими буквами, цифрами и подчеркиванием.

4. Пробелы не могут быть использованы в идентификаторе. Используйте подчеркивание, чтобы обеспечить разделение между частями слова.

5. Идентификаторы являются чувствительными к регистру ввода (к размеру букв): большие и маленькие буквы не являются тем же самым.

Пример правильных идентификаторов:

```
HELLO
hello
 KSinput
P_h
This_is_a_very_long_identifier
```

Отметьте использование подчеркивание чтобы разделить слова. Заметьте также, что маленькие буквы являются не тем же самым, что и большие. Таким образом "hello" является отличным идентификатором от "HELLO". Используйте различные регистры, чтобы сделать ваш исходный файл легко читаемым.

Пример неправильных идентификаторов:

7	Не начинается с буквы или подчеркивания.
$4	Не начинается с буквы или подчеркивания.
HEL.LO	Содержит точку.
b6 kj	Содержит пробел.

Заметьте, что последний из этих неправильных идентификаторов будет рассматриваться языковым процессором как два идентификатора, "b6" и "kj".

2.3.1. РЕЗЕРВНЫЕ ИДЕНТИФИКАТОРЫ

Резервными идентификаторами являются ключевые слова, которые являются частью языка проектирования ABEL. Это означает, что они не могут быть использованы для обозначения микросхем, выводов, узлов, констант, групп, макросов или сигналов. Когда используется ключевое слово, оно относится только к функции этого ключевого слова. Если ключевое слово используется в неправильном контексте, языковым процессором регистрируется ошибка. Ниже, в таблице 2-2 перечислены эти резервированные идентификаторы:

Таблица 2-2. Резервные идентификаторы

CASE	GOTO	PIN
DEVICE	IF	STATE
ELSE	IN	STATE_DIAGRAM
ENABLE	ISTYPE	TEST_VECTORS
END	LIBRARY	THEN
ENDCASE	MACRO	TITLE
EQUATIONS	MODULE	TRUTH_TABLE
FLAG	NODE	

2.3.2. ВЫБИРАЕМЫЕ ИДЕНТИФИКАТОРЫ

Правильный выбор идентификаторов может сделать исходный файл легко читаемым и легко понимаемым. Это особенно важно в условиях, где больше чем один человек работает над одной и той же разработкой. Чтобы помочь сделать логическое описание самообъясняющим, таким образом ограничивая необходимость объемной документации, ниже даются следующие предложения:

* Выбирайте идентификаторы, которые соответствуют функции того, что вы описываете. Например, вывод, который будет использоваться как входной перенос в сумматоре, может быть назван Carry_In. Для простого вентиля ИЛИ, два входных вывода могут быть даны как IN1 и IN2, а выход может назван OR.
* Избегайте длинного ряда похожих идентификаторов. Например, не называйте выходы 16-разрядного сумматора ADDER_OUTPUT_BIT_1, ADDER_OUTPUT_BIT_2, и так далее. Такое группирование имен делает исходный файл трудным для чтения.
* Используйте подчеркивание чтобы отделить слова в вашем идентификаторе. THIS IS AN IDENTIFIER много легче читать, чем THISISANIDENTIFIER.
* Идентификаторы с буквами разного размера могут помочь сделать ваш исходный файл более читабельным: CarryIn.

2.4. СТРОКИ

Строки являются последовательностями ASCII знаков, заключенных в апострофы. Строки используются в операторах TITLE, MODULE и FLAG, а также в декларациях выводов, узлов и атрибутов. Пробелы в строках допустимы.

Правильные строки:

```
'Hello'
' Text with a space in front'
''
'The preceding line is a empty string'
'Punctuation? is even allowed!!'
```

Чтобы включить в текст апостроф, ему должна предшествовать обратная косая черта "\".

```
'It\'s easy to use ABEL'
```

преобразуется в строку "It's easy to use ABEL".

Чтобы включить в текст обратную косую черту, она должна быть написана дважды.

```
'He\\she can use backslashes in a string'
```

преобразуется в строку

"He\she can use backslashes in a string"

ПРИМЕЧАНИЕ:

Обратный апостроф (`) также трактуется как ограничитель строки и может быть использован вместе с прямым апострофом(').

2.5. КОММЕНТАРИИ

Комментарии являются другим средством сделать исходный файл легко понимаемым. Комментарий объясняет то, что не является прозрачным из самого исходного текста. Комментарии не воздействуют на значение кода. Комментарий начинается с кавычек и заканчивается либо кавычками, либо концом строки, в зависимости от того, что встретится раньше. Текст комментария следует за открывающими кавычками.

Комментарий не может быть встроен внутрь ключевых слов.

Примеры правильных комментариев:

```
MODULE Basic_Logic,"gives the module a name
TITLE 'ABEL design example: simple gates'; "title
"declaration section"
IC4 device 'P10L8'; "declare IC4 to be a P10L8
IC5 "decoder PAL" device 'P10H8';
```

Заметьте, что информация внутри апострофов, является не комментарием, а частью оператора.

2.6. ЧИСЛА

Все операции в АБЕЛе связанные с числовыми значениями, делаются с 32-битовой точностью. Таким образом, правильные числовые значения попадают в диапазон 0 - (2 в степени 32)-1. Числа представляются только в пяти формах. Четыре наиболее употребительные формы представляют числа по различному основанию. Пятая форма использует алфавитные символы, чтобы представить числовое значение.

Когда одно из четырех оснований используется для того, чтобы представлять число, это основание отображается символом, предшествующим числу. Нижеследующая таблица (2-3) перечисляет четыре основания, поддерживаемые АБЕЛем и их сопровождающие символы. Символы основания могут быть напечатаны как маленькими, так и большими буквами.

Таблица 2-3. Представление чисел по различному основанию

Название	Основание	Символ
Двоичное	2	^b
Восьмеричное	8	^o
Десятичное	10	^d
Шестнадцатеричное	16	^h

Когда определяется число и не имеет предшествующего символа основания, считается что оно введено с основанием умолчания. Нормально по умолчанию основанием является 10, поэтому числа представляются в десятичной форме, если им не предшествует символ, отображающий другое основание.

Для специальных применений основание, используемое по умолчанию, может быть изменено (см. "@RADIX", глава 4, для более подробной информации.

Ниже приводятся правильные спецификации чисел. По умолчанию считается основанием 10.

Спецификация	Десятичное значение
75	75
^h75	117
^b101	5
^o17	15
^h0F	15

Символ "^" должен быть введен как знак с клавиатуры. Он не является ключом управляющей последовательности, как в некотором другом популярном программном обеспечении.

Числа могут быть также определены символьными знаками. В этом случае в качестве числового значения используется числовое значение ASCII кода. Например, буква "a" является десятичным 97, шестнадцатеричным 61 в кодировании ASCII. Десятичное значение 97 может быть использовано, если "a" было определено как число.

Последовательность алфавитных символов сначала преобразуется в двоичные ASCII значения символов, затем эти значения составляют число (обычно большое).

Ниже даны некоторые примеры чисел, определенных символами:

Спецификация	Шестнадцатеричное значение	Десятичное значение
'a'	'h61	97
'b'	'h62	98
'abc'	'h616263	6382203

2.7. СПЕЦИАЛЬНЫЕ КОНСТАНТЫ

Константы- неизменяемые значения, могут быть использованы в АБЕЛевском логическом описании. Постоянные значения использованы в операторах присвоения, в таблицах истинности и тестовых векторах, а также иногда присваиваются идентификатору, который затем обозначает некоторое значение во всем модуле (смотри "Декларации", глава 3.5 и "Оператор MODULE", глава 3.2).

Постоянные значения могут быть либо цифровыми или одно из значений специальных нецифровых констант. Специальные значения перечислены ниже, в таблице 2-4.

Таблица 2-4. Значения специальных констант

Значение	Описание
.C.	Тактируемый вход (переход L-H-L)
.F.	Плавающий входной или выходной сигнал
.K.	Тактируемый вход (переход H-L-H)
.P.	Регистровая предустановка
.SVn.	n=2-9. Устанавливает вход в перенапряжение от 2 до 9.
.X.	Безразличное состояние
.Z.	Проверка входа или выхода на высокое сопротивление

Когда используется одна из специальных констант, она должна быть введена как показано в таблице 2-4, окруженная точками. Точки индицируют, что использовано специальная константа; без точек .C. будет считаться идентификатором "C". Специальные константы могут быть введены и маленькими, и большими буквами.

2.8. ОПЕРАТОРЫ, ВЫРАЖЕНИЯ И УРАВНЕНИЯ

Такие понятия как константы и имена сигналов могут быть собраны вместе в выражение. Выражение комбинирует, сравнивает или выполняет операции над понятиями, которые они включают, чтобы произвести единый результат. Операции, которые должны быть выполнены, (сложение и логическое И, например) отображаются операторами внутри выражения.

АБЕЛевские операторы разделены на четыре основных типа: логические, арифметические, отношения и присвоения. Каждый из этих типов обсуждается отдельно ниже, за этим следует описание как они комбинируются в выражения, затем сводка всех операторов и правила, распространяющиеся на них, и, наконец, объяснение как уравнения используют выражения.

2.8.1. ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Логические операторы используются в булевских выражениях. В АБЕЛЬ встроены стандартные логические операторы, которые используются в большинстве логических разработок; эти операторы перечислены в таблице 2-5.

Логические операторы обрабатывающие операнды размером больше чем один бит, выполняются побитово. Таким образом, 2#4 эквивалентно 6.

Таблица 2-5. Логические операторы

Оператор	Пример	Описание
!	!A	NOT - дополнение
&	A & B	AND - логическое И
#	A # B	OR - логическое ИЛИ
$	A $ B	XOR - исключающее ИЛИ
!$	A !$ B	XNOR- включающее ИЛИ

2.8.2. АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ

Арифметические операторы определяют арифметическое отношение между аргументами в выражении. Операторы сдвига включены в этот класс потому что каждый сдвиг влево на один бит является эквивалентным умножению на 2, а каждый сдвиг вправо на один бит является эквивалентным делению на 2. Таблица 2-6 перечисляет арифметические операторы.

Таблица 2-6. Арифметические операторы

Оператор	Пример	Описание
-	-A	двоичное дополнение
-	A - B	вычитание
+	A + B	сложение
*	A * B	умножение
/	A / B	беззнаковое целое деление
%	A % B	модуль: остаток от деления
<<	A << B	сдвиг A влево на B бит
>>	A >> B	сдвиг A вправо на B бит

Заметьте, что знак минус имеет различное значение в зависимости от использования. Когда он использован с одним операндом, это означает, что двоичное дополнение операнда должно быть сформировано. Когда знак минус найден между двумя операндами, двоичное дополнение второго операнда добавляется к первому.

Деление является беззнаковым целым делением: результат деления является положительным целым. Остаток от деления может быть получен использованием оператора модуля, "%".

Операторы сдвига выполняют логические беззнаковые сдвиги. При сдвиге вправо слева вдвигаются нули, и при сдвиге влево нули вдвигаются справа.

2.8.3. ОПЕРАТОРЫ ОТНОШЕНИЯ

Операторы отношения используются чтобы сравнивать два аргумента в выражениях. Выражения, сформированные операторами отношения, генерируют булевские значения ИСТИННО (truth) или ЛОЖНО (false). Таблица 2-7 перечисляет операторы отношения.

Таблица 2-7. Операторы отношения

Оператор	Пример	Описание
==	A == B	равно
!=	A != B	не равно
<	A < B	меньше чем
<=	A <= B	меньше чем или равно
>	A > B	больше чем
>=	A >= B	больше чем или равно

Операторы отношения являются беззнаковыми. Например, выражение -1 > 4 является истинным, так как двоичное дополнение 1 является 1111, которая составляет

15 в беззнаковом виде, а 15 больше чем 4. В этом примере предполагались четырехбитовые слова; на самом деле двоичное представление числа -1 будет 32 бита установленные в 1.

Некоторые примеры операторов отношения в выражениях приведены ниже:

Выражение	Значение
2 == 3	false
2 != 3	truth
3 < 5	truth
-1 > 2	truth

Логические значения "truth" и "false" представлены внутренне числами. Логическое "truth" равно -1, т.е. все 32 бита установлены в 1. Логическое "false" равно 0, т.е. все 32 бита установлены в 0. Это означает, что выражения, производящие в результате значения ИСТИННО или ЛОЖНО, могут быть использованы везде как число. Кроме этого, числовые выражения 0 и -1 могут быть подставлены в выражения как логическое значение.

Пример:

```
A = D $ (B == C);
```

означает, что:

A будет эквивалентно дополнению D, если B равно C

A будет эквивалентно D, если B не равно C

2.8.4. ОПЕРАТОРЫ ПРИСВОЕНИЯ

Операторы присвоения являются специальным классом операторов, используемых в уравнениях, а не в выражениях. Уравнение присваивает значение выражения выходном сигналам. Смотрите главу 2.8.6 для более полного обсуждения уравнений.

Существует два оператора присвоения, нетактируемое и тактируемое. Нетактируемое или непосредственное присвоение происходит без какой-либо задержки, сразу же как только значение вычислено. Тактируемое присвоение происходит на следующий тактовый импульс от тактового входа, связанного с этим выходом. Таблица 2-8 показывает операторы присвоения.

Таблица 2-8. Операторы присвоения

Оператор	Описание
=	Нетактируемое (непосредственное) присвоение
:=	Тактируемое присвоение

2.8.5. ВЫРАЖЕНИЯ

Выражения являются комбинацией идентификаторов и операторов, которая производит один результат, когда вычисляется. В выражении могут быть использованы любые логические, арифметические или отношения операторы.

Выражения вычисляются в соответствии с вовлеченными операторами. Некоторые операторы имеют приоритет перед другими и их операции будут выполняться раньше. Каждый оператор имеет назначенный ему приоритет, который определяет порядок вычисления. Приоритет 1 является высшим приоритетом, а приоритет 4- низшим.

Таблица 2-9 суммирует операторы логические, арифметические и отношения, представленные группами в соответствии с их приоритетами.

Когда встречаются в одном выражении операции с равными приоритетами, они выполняются в том порядке, в котором находятся в выражении слева направо. Как и в обычной математике, могут быть использованы скобки чтобы изменить порядок вычисления, причем операции в самых внутренних скобках выполняются самыми первыми.

Некоторые примеры правильных выражений даются ниже. Заметьте как порядок операций и использование скобок влияют на результат.

Выражение	Результат	Комментарии
2*3/2	3	Операторы с равным приоритетом
2 * 3 / 2	3	Пробелы допустимы
2*(3/2)	2	Использование скобок
2+3*4	14	
2#4$2	4	
2#(4$2)	6	
2==^HA	0	Ложно
14==^HE	-1	Истинно

Таблица 2-9. Сводка операторов и приоритетов

Приоритет	Оператор	Описание
1	-	двоичное дополнение, отрицание
1	!	дополнение к единице, лог.НЕ
2	&	логическое И
2	<<	сдвиг влево
2	>>	сдвиг вправо
2	*	умножение
2	/	беззнаковое деление
2	%	модуль, остаток от деления
3	+	сложение
3	-	вычитание
3	#	логическое ИЛИ
3	$	исключающее ИЛИ
3	!$	включающее ИЛИ
4	==	эквивалентность
4	!=	неэквивалентность
4	<	меньше чем
4	<=	меньше чем или равно
4	>	больше чем
4	>=	больше чем или равно

2.8.6. УРАВНЕНИЯ

```
   [!]...[ENABLE] element = expression
или
   [!]...[ENABLE] element := expression
element        идентификатор, означающий сигнал или группу
               сигналов или набор, которому будет
               присвоено значение выражения.
expression     любое правильное выражение.
= и :=         оператор нетактируемого или тактируемого
               присвоения.
```

Уравнения присваивают значение выражения сигналу или набору сигналов в логическом описании. Идентификатор и выражение должны следовать правилам, уже установленным для этих элементов.

Уравнения используют два оператора присвоения: "=" (нетактируемое) и ":=" (тактируемое) описанных в главе 2.8.4.

Ключевое слово ENABLE используется чтобы разрешить выходные буферы с тремя состояниями. Идентификатор должен быть выходом типа "три состояния" и присвоение значения прилагается только к разрешению буфера, а не к самому сигналу.

Оператор дополнения "!" может быть использован, чтобы выразить отрицательную логику. Оператор дополнения предшествует имени сигнала и означает, что выражение в правой части уравнения должно быть дополнено прежде чем будет присвоено сигналу. Использование оператора дополнения в левой части уравнения обеспечивается как дополнительная возможность; уравнение для отрицательной логики может быть выражено дополнением выражения в правой части уравнения.

Примеры уравнений:

```
X=A&B;          " нетактируемое присвоение X
ENABLE Y=C#D;   " Y разрешено если C или D истинны
Y:=A&B;         " тактируемое присвоение Y
!A=B&C#D;       " то же что и A=!(B&C#D);
!!A=B&C#D;      " то же что и A=B&C#D;
```

МНОЖЕСТВЕННЫЕ ПРИСВОЕНИЯ ОДНОМУ ИДЕНТИФИКАТОРУ

Когда идентификатор появляется в левой части более чем одного уравнения, выражение присвоенное сначала обрабатывается по ИЛИ со следующим выражением и затем делается присвоение. Если идентификатор в левой части уравнения дополняется, дополнение выполняется после того как сделаны все операции ИЛИ.

Примеры:

Уравнения в тексте	Эквивалентное уравнение
A=B; A=C;	A = B # C;
A=B; A=C&D;	A = B # (C & D);

```
A=!B;
A=!C;          A = !B # !C;

!A=B;
!A=C;          A = ! (B # C);

!A=B;
A=!C;          A = !C # !B

!A=B;
!A=C;
A=!D;
A=!E;          A = !D # !E # ! ( B # C );
```

Заметьте, что когда оператор дополнения появляется в левой части уравнений множественного присвоения, в правой части уравнения сначала выполняется ИЛИ и только потом дополнение.

2.9. ГРУППЫ

Группой является набор сигналов и констант, которые обрабатываются как одно целое. Любые операции, приложенные к группе, прикладываются к каждому элементу группы. Группы упрощают логические описания и тестовые вектора АБЕЛя, позволяя ссылаться одним именем на группы сигналов.

Например, выходы B0-B7 восьмиразрядного мультиплексора могут быть собраны в группу, названную MULTOUT. Три линии адреса могут быть собраны в группу SELECT. Тогда мультиплексор может быть описан в понятиях MULTOUT и SELECT, а не индивидуальным определением каждого входного и выходного бита.

Группа представляется списком констант и сигналов, разделенных запятыми и окруженными квадратными скобками. Группы мультиплексора, описанные выше, будут выглядеть следующим образом:

```
Набор                  Описание
[B0,B1,B2,B3,B4,B5,B6,B7]  выходы (MULTOUT)
[S0,S1,S2]                 линии выбора (SELECT)
```

Заметьте, что для определений группы квадратные скобки не означают необязательное понятие. Квадратные скобки требуются, чтобы выделить группу.

Заметьте, что АБЕЛевские группы не являются математическими группами.

2.9.1. ОПЕРАЦИИ С ГРУППАМИ

Большинство операций применимо и к группам. Обычно это означает, что операция выполняется над каждым элементом группы, иногда индивидуально и иногда в соответствии с правилами булевой алгебры. Таблица 2-10 перечисляет операторы, которые могут быть использованы с группами. Окончание B описывает как эти операции применимы к группам.

Таблица 2-10. Правильные операции с группами

Оператор	Пример	Описание
=	A=5	присваивание
:=	A:=[1,0,1]	тактированное присваивание
!	!A	NOT - дополнение к 1
&	A & B	AND - логическое И
#	A # B	OR - логическое ИЛИ
$	A $ B	XOR - исключающее ИЛИ
!$	A !$ B	XNOR- включающее ИЛИ
-	-A	двоичное дополнение (отрицание)
-	A - B	вычитание
+	A + B	сложение
==	A == B	равно
!=	A != B	не равно
<	A < B	меньше чем
<=	A <= B	меньше чем или равно
>	A > B	больше чем
>=	A >= B	больше чем или равно

Для операций вовлекающих два или больше наборов, наборы должны иметь то же самое количество элементов. Выражение "[a,b]+[c,d,e]" является неправильным, так как наборы содержат различное число элементов.

Некоторые примеры наборов даны ниже.

Булевское уравнение

```
Chip_Sel = A15 & !A14 & a13;
```

представляет из себя адресный декодер, где A15, A14 и A13 являются тремя старшими битами 16-битового адреса. Декодер может быть легко выполнен операциями с набором.

Прежде всего, определим набор констант, содержащих адресные линии, так чтобы к этому набору можно было обращаться по имени. Это определение сделано в секции определения констант модуля (описанного в главе 3). Декларация выглядит так:

```
Addr = [A15,A14,A13];
```

которая декларирует набор констант Addr. Уравнение

```
Chip_Set = Addr == [1,0,1];
```

является функционально эквивалентным уравнению

```
Chip_Sel = A15 & !A14 & a13;
```

и Addr эквивалентен [1,0,1], что означает что Chip Set истинно, когда A15=1, A14=0. и A13=1. Заметьте, что уравнение с набором может быть записано также следующим образом

```
Chip_Set = Addr == 5;
```

так как двоичное 101 равно десятичному 5.

В примере выше, был декларирован и использован в операциях с наборами специальный набор старших бит 16-разрядного адреса. Может быть использован и полный адрес и та же функция может быть достигнута другими путями, два из которых показаны ниже.

Пример 1:

```
"declare some constants in declaration section
Addr=[a15,a14,a13,a12,a11,a10,a9,a8,a7,a6,a5,a4,a3,a2,a1,a0];
X=.X.; "simplify notation for don't care constant
Chip_Sel = Addr == [1,0,1,X,X,X,X,X,X,X,X,X,X,X,X,X];
```

Пример 2:

```
"declare some constants in declaration section
Addr=[a15,a14,a13,a12,a11,a10,a9,a8,a7,a6,a5,a4,a3,a2,a1,a0];
X=.X.;
Chip_Sel = (Addr >= ^HA000) & (Addr <= ^HBFFF);
```

2.9.2. ПРИСВОЕНИЕ И СРАВНЕНИЕ ГРУПП

Значения и значения наборов могут быть присвоены и сравнены с группой. Например:

```
sigset = [1,1,0] & [0,1,1];
```

результирует в присвоение "sigset" значения [0,1,0].

Числа в любом представлении могут быть присвоены или сравнены с группой. Предыдущее уравнение может быть записано как

```
sigset = 6 & 3;
```

Когда числа используются для присвоения или сравнения с группой, число преобразуется в его двоичное представление при использовании двух правил:

1. Если число значащих бит в двоичном представлении числа больше, чем число элементов в группе, то левые биты отбрасываются.

2. Если число значащих бит в двоичном представлении числа меньше, чем число элементов в группе, то левые биты числа дополняются нулями.

Таким образом, следующие два присвоения являются эквивалентными:

```
[a,b] = ^B101011; [a,b] = ^B11;
```

Эквивалентны и эти два присвоения:

```
[a,b] = ^B01;
[a,b] = ^B1;
```

Присвоение группы

```
[a,b] = c & d;
```

эквивалентно двум присвоениям:

```
a = c & d;
b = c & d;
```

2.9.3. ОГРАНИЧЕНИЯ НА ГРУППЫ

К группам применимо следующее ограничение:

Поскольку присвоение групп применимо ко всем элементам группы, группы со смесью обычных и регистровых выходов не могут быть использованы в левой части уравнения. Присвоение может быть либо нетактируемым (для комбинационной логики) либо тактируемым (регистровое) в соответствии с использованным оператором присвоения (= или :=) и тактируемое присвоение комбинационному выходу или нетактируемое присвоение регистровому выходу вызывает ошибку, которая будет зарегистрирована языковым процессором.

2.10. БЛОКИ

Блоком является секция ASCII текста, заключенная в фигурные "{" и "}" скобки. Блоки используются в макросах и директивах. Текст, содержащийся в блоке, может содержать одну или несколько строк. Некоторые примеры блоков показаны ниже:

```
{ this is a block }
{
this is also a block, and it
spans more than one line
}
{ A = B # C;
D = [0,1]
```

Блоки и гнезда блоков могут быть полезны в макросах и при использовании в директивах (см. "Макродекларации" глава 3.5.5. и "Директивы" глава 4).

Если левая либо правая фигурная скобка требуется в качестве символа в блоке, перед ней ставится обратная косая черта, таким образом

```
{ !{ !} }
```

является блоком, содержащим символы " { } " с соответствующими пробелами.

2.11. АРГУМЕНТЫ И ПОДСТАНОВКА АРГУМЕНТОВ

В макросах, модулях и директивах могут быть использованы значения переменных. Эти значения называются аргументами конструкции, которая их использует. В АБЕЛе должно делаться различие между двумя типами аргументов: настоящими и фиктивными. Эти различие даются ниже.

Фиктивный аргумент	Идентификатор, который используется чтобы отметить, где должен быть подставлен настоящий аргумент в макро, модуле, директиве.
Настоящий аргумент	Аргумент (значение) используемое в макро, директиве или модуле. Настоящий аргумент подставляется вместо фиктивного. Настоящим аргументом может быть любой текст, включая идентификаторы, числа, строки, операторы, группы или любой другой элемент АБЕЛя.

Фиктивные аргументы определяются в макро декларациях и в телах макросов, модулей, директив. Фиктивному аргументу предшествует вопросительный знак, в том месте где должен быть подставлен настоящий аргумент. Вопросительный знак отличает фиктивные аргументы от других абелевских идентификаторов, встречающихся в исходном файле.

Возьмем в качестве примера следующую макро декларацию (макросы обсуждаются полностью в главе 3).

```
OR_EM MACRO (a,b,c){?a # ?b # ?c};
```

Это определяет макро по имени OR EM которая является логическим ИЛИ трех аргументов. Эти аргументы представлены в определении макро фиктивными аргументами a, b и c. В теле макро, которое окружено фигурными скобками, перед фиктивными аргументами стоят вопросительные знаки, чтобы отметить необходимость подстановки фактических аргументов.

Теперь, уравнение

```
D=OR_EM(x,y,z&1);
```

вовлечет макро OR EM с фактическими аргументами x, y и z&1. Это результирует в уравнение

```
D=x#y#z&1;
```

Пробелы являются существенными для фактических аргументов. Фактические аргументы подставляются точно как они появляются. Заметьте, что в примере выше фактический аргумент z&1 не содержит пробелов в уравнении, относящемуся к OR EM, и соответственно в расширенном уравнении аргументы появляются тоже без пробелов. Если бы был определен как "z & 1" (с пробелами), результирующее уравнение содержало бы эти пробелы. Например, уравнение

```
D=OR_EM(x,y,z & 1);
D=x#y#z & 1;
```

Подстановка аргументов происходит прежде, чем исходный файл проверяется на синтаксическую или логическую корректность. Это означает, что код проверяется на корректность с подставленными фактическими аргументами. Таким образом, если фактические аргументы нарушают синтаксические или логические правила, программа разбора обнаружит это и сообщит об ошибке.

В заключение:

Фиктивные аргументы резервируют место под фактические аргументы. Вопросительные знаки перед фиктивными аргументами отмечают, что должны быть подставлены фактические аргументы.

Фактические аргументы заменяют фиктивные перед тем, как исходный файл проверяется на корректность.

Пробелы являются существенными для фактических аргументов.

Дальнейшее обсуждение и примеры использования аргументов даны в ABEL Applications Guide, в описаниях модулей и макросов в главе 3 и в главе 4 в описании директив.

ГЛАВА 3 СТРУКТУРА

Эта глава описывает структуру полного АБЕЛевского логического описания. Элементы языка, описанные в главе 2, комбинируются с соответствующими структурами, чтобы определить булевские уравнения, автоматы или таблицы истинности. Должны быть сделаны начальные декларации и индицированы устройства, которые описываются. Могут также быть сделаны определения ожидаемых выходов из симуляции устройства.

3.1. ОСНОВЫ СТРУКТУРЫ

Исходный абелевский файл может быть разбит на независимые части, называемые модулями. Каждый модуль содержит одно или несколько завершенных логических описаний. На простейшем уровне исходный файл для абелевского языкового процессора состоит только из одного модуля; на более сложном уровне, неопределенного число модулей может быть скомбинировано в один исходный файл и обработано одновременно.

Каждый модуль состоит из нескольких различных секций, каждая со своей собственной уникальной функцией. Эти секциями являются:

Декларации устройств, выводов, узлов, констант, атрибутов и макросов.
Булевские логические уравнения.
Таблицы истинности.
Диаграммы состояний.
Тестовые вектора.

Некоторые или все эти части модуля могут существовать для данного применения. Декларации должны быть сделаны всегда. Рисунок 1-3 показывает структуру абелевского исходного файла.

Поскольку исходный файл является собранием одного или более модулей, каждый со своим началом и концом, разные исходные файлы могут быть скомбинированы, чтобы сформировать завершенный проект системы в одном исходном файле.

Рисунок 3-1. Структура абелевского исходного файла

```
1st Module Start
    Flags
    Title
    Declarations
        Constant Declarations
        Macro Definitions
        Device Declarations
        Pin and Node Assignments
        Attribute Declarations
    Library References
    Boolean Equations
    Truth Tables
    State Diagrams
    Test Vectors
1st Module End
2nd Module Start
2nd Module End
```

3.2. ОПЕРАТОРЫ MODULE И СТРУКТУРА

```
MODULE modname [ (dummy_arg[,dummy_arg]...) ]
       [ FLAG statement ]...
       [ TITLE statement ]...
       declarations
       [ EQUATIONS ]...
       [ TRUTH_TABLE ]...
       [ STATE_DIAGRAM ]...
       [ TEST_VECTORS ]...
   END [ modname ] [;]
```

modname	правильный идентификатор, именующий модуль.
dummy_arg	фиктивный аргумент.
FLAG	определяет параметры обработки (он передается языковому процессору).
TITLE	определяет заголовок модуля.
declarations	секция деклараций модуля, в которой определяются выводы, узлы, микросхемы и атрибуты.
EQUATIONS	область булевских уравнений.
TRUTHY_TABLE	область таблицы истинности.
STATE_DIAGRAM	область диаграммы состояний.
TEST_VECTORS	определение тестовых векторов.

Оператор module определяет начало модуля и должен употребляться в паре с оператором end, который определяет конец модуля.

Каждый модуль в исходном файле должен иметь уникальное имя.

Модуль содержит различные декларации, уравнения, таблицы истинности, диаграммы состояний и симуляционные таблицы необходимые для полноты логического описания. К структуре модуля применимы следующие три правила:

1. Если используется оператор FLAG, он должен быть первым ключевым словом, использованным после оператора MODULE.

2. Если использован оператор TITLE, он должен быть первым ключевым словом, использованным после оператора FLAG (если он существует). Если FLAG не использован, оператор TITLE должен быть первым ключевым словом, использованным после оператора MODULE.

3. В модуле должна существовать единственная секция деклараций. Все другие секции могут встречаться в модуле так часто, как нужно и в любом порядке.

Дополнительные фиктивные аргументы в операторе module позволяют передавать в модуль фактические аргументы, когда модуль обрабатывается языковым процессором. Фиктивный аргумент обеспечивает имя, чтобы ссылаться внутрь модуля. Везде в модуле, где встретится пустой аргумент, предваряемый вопросительным знаком "?", программой-разборщиком будет подставлен текущий фактический аргумент.

Пример:

```
MODULE MY_EXAMPLE(A,B)
        .
        .
C = ?B + ?A
        .
        .
END
```


В этом модуле с именем MY EXAMPLE, C примет значение "A+B", где A и B содержат фактические аргументы, переданные модулю, когда работает языковой процессор. Для большей информации о фиктивных аргументах смотрите главу 2.11 в этом руководстве.

3.3. ОПЕРАТОР FLAG

```
FLAG parameter[,parameter]...
```

parameter	строка, содержащая правильную командную строку параметров, как определено в ABEL User's Guide за исключением -I, -O, -N и все параметров для программы PARSE.

Оператор FLAG обеспечивает альтернативный метод определения параметров обработки, который воздействует на способ, каким языковой процессор обрабатывает исходный файл. Эти параметры обычно передаются из командной строки или из батч-файла, когда запускается языковой процессор. FLAG позволяет включить параметры обработки непосредственно в исходный файл. Недопустимы параметры программы PARSE, а также параметры -I, -O, -N. Полная информация о параметрах командной строки содержится в ABEL User's Guide.

FLAG полезен когда исходный файл требует специфического типа или уровня обработки, для того чтобы обработка была успешной. Например, проект может быть достаточно большой или сложный чтобы генерировать слишком много термов для определенной микросхемы, если на ступени редукции обработки не будет применен алгоритм редукции PRESTO. Вы можете также пожелать симулировать проект на третьем уровне трассировки.

Это делается следующей командой:

```
flag '-r2,-f3'
```

3.4. ОПЕРАТОР TITLE

```
TITLE string
```

Оператор title используется чтобы дать модулю ярлык, который появится как заголовок как в файле загрузки программатора, так и в документационном файле, созданными языковым процессором. Заголовок определяется в строке, которая следует за ключевым словом TITLE.

Использование оператора TITLE не является обязательным.

Если в строке заголовка будут найдены звездочки, они не появятся в заголовке загрузочного файла программатора, для того чтобы соответствовать стандарту JEDEC.

Пример оператора TITLE, который включает три строки и описывает логический проект, выглядит следующим образом:

```
module m6809a
title '6809 memory decode
Jean Designer
Data I/O Corp Redmond WA Jan 1984'
```

Для более подробной информации, касающейся загрузочного файла программатора, обратитесь к ABEL User's Guide.

3.5. ДЕКЛАРАЦИИ

Секция деклараций модуля определяет описываемое устройство и соответствие между используемыми в модуле именами и выводами и узлами микросхемы. В секции деклараций определяются также константы, атрибуты и макросы. Декларации имеют эффект только в модуле, в котором они определены. Каждый модуль должен иметь свою собственную секцию деклараций.

Существует шесть типов операторов деклараций:

Device	Node	Macro
Pin	Constant	Attribute

Синтаксис и использование каждого из этих типов представлены в следующих главах.

3.5.1. ОПЕРАТОР ДЕКЛАРАЦИИ DEVICE

```
device_id[,device_id]... DEVICE real_device
device_id     идентификатор, используемый в модуле для ссылки
              на устройство (микросхему).
real_device   строка, описывающая реальные имена микросхем.
```

Оператор декларации устройства связывает имена устройств, используемых в модуле с фактическими микросхемами программируемой логики, на которых выполняется устройство. Имя устройства, используемое в логическом описании, определяется как DEVICE ID. Промышленное название устройства программируемой логики, отображается строкой REAL DEVICE. Список устройств, (микросхем) поддерживаемых АБЕЛем, может быть найден в приложении D руководства ABEL User's Guide. Это единственные устройства, которые могут быть определены с помощью REAL DEVICE.

Обязательно требуется завершающая точка с запятой.

Следующий пример определяет два имени устройства, U14 и U15, которые представляют ПЛМ типа F82S159:

```
U14,U15 device 'F82S159';
```

3.5.2. ОПЕРАТОР ДЕКЛАРАЦИИ PIN

```
pin_id[,pin_id]... PIN [IN device_id]
    pin#['attr[,attr]...'] [,pin#['attr[,attr]...']]...
pin_id        идентификатор, используемый в модуле для ссылки
              на вывод.
device_id     идентификатор, декларирующий имя устройства,
              отмечающий устройство, связанное с этими выво-
              дами.
pin#          номер вывода на реальном устройстве.
attr          строка которая определяет атрибуты вывода для
              устройств с программируемыми выводами. Возмож-
              ные значения:
                  pos       положительная полярность
                  neg       отрицательная полярность
                  reg       регистровый сигнал
                  com       комбинационный сигнал
                  latch     входной вывод лэтча
                  feed_pin  обратная связь от вывода
                  feed_reg  обратная связь от регистра
                  feed_or   обратная связь от матрицы ИЛИ
```

Оператор декларации выводов отражает соответствие между идентификаторами, используемыми в модуле, и выводами реальной микросхемы. Декларация может также определить атрибуты для выводов с программируемыми характеристиками выводов.

Когда списки pin ids и pin# использованы в одном операторе декларации, существует однозначная связь между идентификаторами и данными номерами. Должен быть перечислен каждый идентификатор связанный с выводом.

Когда декларировано больше чем одно устройство в одном модуле, должно быть дополнительно употреблено IN, так чтобы выводы были связаны с надлежащей микросхемой. Декларация устройства должна быть сделана перед декларацией выводов. Требуется завершающая точка с запятой. Пример простой декларации выводов приведен ниже:

```
Clock,Reset,S1 PIN IN U12 12,15,3 ;
```

Эта декларация выводов присваивает имя вывода Clock выводу 12 микросхемы U12, Reset- выводу 15, S1- выводу 3.

Атрибут вывода, ATTR, определяет атрибуты выводов. Атрибуты могут быть определены либо этим способом, либо оператором ISTYPE. Оператор ISTYPE и атрибуты обсуждаются в главе 3.5.6. Декларация выводов

```
F0 pin 13 = 'neg,reg';
```

определяет, что уравнения для F0, которые соответствуют выводу 13, должны быть оптимизированы для отрицательной полярности регистрового выхода.

3.5.3. ОПЕРАТОР ДЕКЛАРАЦИИ NODE

```
node_id[,node_id]... NODE [IN device_id]
    node#['attr[,attr]...'] [,node#['attr[,attr]...']]...
node_id       идентификатор, используемый в модуле для ссылки
              на узел в логическом проекте.
device_id     идентификатор, декларирующий имя устройства, от-
              мечающий устройство, связанное с этими узлами.
node#         номер узла для реального устройства.
attr          строка которая определяет атрибуты узла для уст-
              ройств с программируемыми выводами. Возможные
              значения:
                  pos       положительная полярность
                  neg       отрицательная полярность
                  reg       регистровый сигнал
                  com       комбинационный сигнал
                  latch     входной вывод лэтча
                  feed_pin  обратная связь от вывода
                  feed_reg  обратная связь от регистра
                  feed_or   обратная связь от матрицы ИЛИ
```

Оператор декларации узла отражает соответствие между идентификаторами, используемыми в модуле, и внутренними узлами реальной микросхемы. Узлы являются "псевдо-выводами"- это внутренние сигналы, которые недоступны с внешних выводов микросхемы, но они нужны чтобы запрограммировать перемычки, которые иным способом недоступны. Узлы микросхем, поддерживаемые АБЕЛем, перечислены в руководстве ABEL User's Guide. Декларация может также определить атрибуты для устройств с программируемыми характеристиками узлов.

Когда списки node ids и node# использованы в одном операторе декларации узлов, существует однозначное соответствие между идентификаторами и данными номерами. Должен быть перечислен каждый идентификатор связанный с узлом.

Когда декларировано больше чем одно устройство в одном модуле, должно быть дополнительно употреблено IN, так чтобы узлы были связаны с надлежащей микросхемой.

Декларация устройства должна быть сделана перед декларацией узлов.

Требуется завершающая точка с запятой. Пример простой декларации узлов приведен ниже:

```
A,B,C NODE IN U15 21,22,23 ;
```

Атрибут узла, ATTR, определяет атрибуты узлов. Атрибуты могут быть определены либо этим способом, либо оператором ISTYPE. Оператор ISTYPE и атрибуты обсуждаются в главе 3.5.6. Декларация узла

```
B node 22 = 'pos,com';
```

определяет, что узел 22 имеет положительную полярность и комбинационный выход.

3.5.4. ОПЕРАТОР ДЕКЛАРАЦИИ КОНСТАНТ

```
id[,id]... = expr[,expr]...;
```

id	идентификатор, именующий константу, которая должна быть использована в модуле.
expr	выражение, определяющее значение константы.

Оператор декларации константы определяет константы, которые должны быть использованы в модуле. Константой является идентификатор, который сохраняет постоянное значение во всем модуле.

Идентификаторам, перечисленным в левой части от знака равенства, присваиваются значения, перечисленные в правой части от знака равенства.

Существует однозначное соответствие между идентификаторами и выражениями и должно быть выражение для каждого идентификатора.

Требуется завершающая точка с запятой.

Константы полезны, когда некоторое значение должно использоваться много раз во всем модуле и особенно полезно, когда это значение может быть изменено в процессе логического проектирования. Вместо изменения значения во всем модуле, оно может быть изменено только один раз в декларации константы.

Некоторые примеры правильных деклараций констант:

```
ABC = 3*17;              " ABC присваивается значение 51
Y = 'Bc';                " Y = ^H4263;
X = .X.;                 " X имеет безразличное значение
ADDR = [1,0,15];         " ADDR является набором трех элементов
A,B,C = 5,[1,0],6;       " Здесь декларированы 3 константы
```



```
D pin 6 ;                " Смотри следующую строку
E = [5*7,D];             " Может быть включено имя сигнала
G = [1,2] + [3,4];       " Легальная операция над группой
A = B & C;               " Правильная операция с
                         " идентификаторами
A = [IB,C];              " Группа и идентификатор в
                         " правой части уравнения
```

3.5.5. ОПЕРАТОР ДЕКЛАРАЦИИ MACRO И МАКРОРАСШИРЕНИЯ

```
macro_id MACRO[(dummy_arg[,dummy_arg]...)] block;
```

macro_id	идентификатор, именующий макро.
dummy_arg	фиктивный аргумент.
block	блок (см. главу 2).

Оператор декларации макро определяет макро. Макросы используются чтобы включить абелевский код в исходный файл, не печатая и не копируя код откуда это нужно. Макро определяется однажды в секции декларации модуля, и затем используется везде внутри модуля так часто как это нужно. Макросы могут быть использованы только внутри модуля в котором они декларированы.

Где бы не встретилось macro id, вместо него будет подставляться текст, связанный с этим макро. С исключением фиктивных аргументов, весь текст в блоке (включая пробелы, конец строки и т.п.) будет подстановлен точно так, как он выглядит в блоке.

Фиктивные аргументы, использованные в декларации макро, позволяют использовать различные фактические аргументы в макро каждый раз, когда макро употребляется в модуле. Внутри макро фиктивным аргументам предшествует вопросительный знак "?", чтобы отметить, что АБЕЛем вместо фиктивного аргумента должен быть подставлен фактический. Это лучше показать на примере.

Уравнение

```
NAND3 MACRO (A,B,C) {!(?A & ?B & ?C)};
```

декларирует макро с именем NAND3 с фиктивными аргументами A,B и C. Макро определяет трехвходовой вентиль И-НЕ. Когда макро идентификатор встречается в исходном тексте, для A, B и C будут подставлены фактические аргументы.

Например уравнение

```
D = NAND3(Clock,Hello,Busy);
```

привлечет текст в блоке, связанном с макро в код, причем Clock будет подставлено вместо ?A, Hello вместо ?B и Busy вместо ?C. Все это даст в результате

```
D = !(Clock & Hello & Busy);
```

что и является трехвходовым И-НЕ.

Макро NAND3 было определено булевским уравнением, но оно может быть определено используя и другие абелевские конструкции, такие как таблица истинности, показанная ниже:

```
NAND3 MACRO (A,B,C,Y)
{ TRUTH_TABLE ([?A,?B,?C]->?Y)
[ 0 ..X..X. ]->1;
[.X., 0 ..X.]->1;
[.X...X., 0 ]->1;
[ 1 , 1 , 1 ]->0;};
```

В этом случае строка

```
NAND3(Clock,Hello,Busy,D)
```

нижеследующий текст

```
{ TRUTH_TABLE ([Clock,Hello,Busy]->D)
[ 0 ,.X.,.X.]->1;
[.X., 0 ,.X.]->1;
[.X...X., 0 ]->1;
[ 1 , 1 , 1 ]->0;
```

подставит в код. Этот текст является таблицей истинности для D, определяющей D как функцию трех входов Clock, Hello и Busy. Это та же самая функция что и описанная булевским уравнением несколько выше. Формат таблицы истинности обсуждается в главе 3.7.

Другие примеры для макросов:

```
A macro {W = S1 & s2 & s3 ;};
"macro w/ no dummy args
B MACRO (d) {!?d}; "macro w/ 1 dummy argument
```

и когда они появятся в логическом описании

```
A
X = W + B(inp);
Y = W + B()C; "note the blank actual argument
```

результатом будет следующая последовательность.

```
"note leading space from block in A
W = S1 & S2 & S3;
X = W + !inp;
Y = W + !C;
```

3.5.6. ОПЕРАТОР ISTYPE

```
signal[,signal]... ISTYPE [IN device_id]'attr[,attr]...';
```

signal	идентификатор, вывода или узла.	
device_id	идентификатор, декларирующий имя устройства, отмечающий устройство, связанное с этими узлами.	
attr	pos	положительная полярность
	neg	отрицательная полярность
	reg	регистровый сигнал
	com	комбинационный сигнал
	latch	входной вывод лэтча
	feed_pin	обратная связь от вывода
	feed_reg	обратная связь от регистра
	feed_or	обратная связь от матрицы ИЛИ

Оператор ISTYPE определяет атрибуты или характеристики выводов или узлов для микросхем с программируемыми характеристиками. Атрибуты используются, чтобы сформировать правильную логику для микросхемы и оптимизировать уравнения для нее. Если для микросхемы не определены никакие атрибуты, они принимаются атрибутами умолчания. Если атрибуты определены для микросхемы без программируемых характеристик, языковой процессор сообщит об ошибке.

Когда в слева от оператора ISTYPE перечислен более чем один "signal", атрибуты в правой части относятся ко всем сигналам.

Когда декларировано больше чем одно устройство в одном модуле, должно быть дополнительно употреблено IN, так чтобы сигналы были связаны с надлежащей микросхемой. Декларация устройства должна быть сделана перед появлением оператора ISTYPE в исходном тексте. Декларация выводов и узлов, используемых в операторе ISTYPE должна быть сделана до этого оператора.

Пример оператора ISTYPE следует ниже:

```
F0,A istype = 'neg,latch';
```

Этот оператор декларации определяет F0 и A как лэтчи отрицательной полярности. Как F0 так и A должны быть определены в модуле ранее.

Определение каждого из атрибутов изложено ниже:

POS (положительная полярность).

POS отмечает, что соответствующий вход или выход имеет положительную полярность. Микросхема будет запрограммирована чтобы отражать это условие и любые уравнения относящиеся к этому сигналу будут оптимизированы для этой полярности. POS может быть введен большими, маленькими и буквами смешанного размера.

NEG (отрицательная полярность).

NEG отмечает, что соответствующий вход или выход имеет отрицательную полярность. Микросхема будет запрограммирована чтобы отражать это условие и любые уравнения относящиеся к этому сигналу будут оптимизированы для этой полярности. NEG может быть введен большими, маленькими и буквами смешанного размера.

REG (регистровый сигнал).

REG отмечает, что соответствующий вход или выход является регистровым, а не комбинационным. Микросхема будет запрограммирована для этого условия и сигнал будет изменяться только при подаче тактового импульса.

COM (комбинационный сигнал).

COM отмечает, что соответствующий вход или выход является комбинационным. Микросхема будет запрограммирована для этого условия.

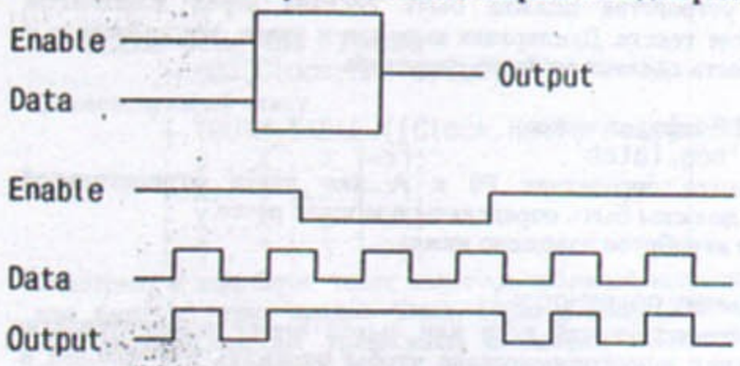
LATCH

Атрибут LATCH отмечает, что соответствующий вход является является лэтчем. LATCH применим только ко входу и вызовет ошибку при приложении ко входу. Лэтчи имеют линию разрешения и работают следующим образом:

1. Если на входе разрешения стоит высокий потенциал, входной сигнал лэтча проходит сквозь лэтч на выход.
2. Если на входе разрешения стоит низкий потенциал, последнее значение входной линии перед переходом линии разрешения из высокого потенциала в низкий, будет удерживаться на линии выхода.

Рисунок 3-2 показывает изображение лэтча и временную диаграмму для его операций.

Рисунок 3-2. Схема и временная диаграмма лэтча



FEED_PIN, FEED_REG, FEED_OR (спецификация обратной связи).

Некоторые микросхемы допускают определение пути внутренней обратной связи. Обратная связь может иметь место от выходного вывода, от выхода внутреннего регистра или от выхода вентиля ИЛИ, как показано на рисунке 3-3.

_Атрибуты feed pin, feed reg или feed or определяют какой путь обратной связи использовать: от вывода микросхемы, от внутреннего регистра или от матрицы ИЛИ, соответственно. В любое данное время может быть использован только один атрибут: если для некоторого сигнала будет определено больше одного атрибута обратной связи, иметь эффект будет только последний определенный атрибут. Кроме этого, атрибут обратной связи является значащим только для микросхем, для которых пользователь может определить путь обратной связи. Атрибуты обратной связи имеют смысл только для выходов. Атрибут может быть введен буквами большими, маленькими и смешанного размера.

3.6. ОПЕРАТОР EQUATIONS

```
EQUATIONS [IN device_id]
```

device_id	ранее определенный идентификатор устройства, которое означает микросхему, связанную с этими уравнениями.

Оператор EQUATIONS определяет начало группы уравнений, относящихся к этой микросхеме. Уравнения определяют логические функции с помощью булевской алгебры.

Оператор EQUATIONS сопровождается уравнениями, относящимися к микросхеме, которая отмечается идентификатором устройства. В модулях только с одним устройством, идентификатор микросхемы не обязателен; в других случаях он должен быть определен, так чтобы уравнения были употреблены по отношению к надлежащей микросхеме.

Уравнения, которые сопровождают оператор EQUATIONS, являются обычными абелевскими уравнениями, как это описано в главе 2.8.6.

Ниже показан пример простой секции уравнений:

```
EQUATIONS IN IC13
A = B & C # A;
[W,Y] = 3;
!F = B == C;
```


3.7. ОПЕРАТОР ТАБЛИЦЫ ИСТИННОСТИ (TRUTH TABLES)

Таблицы истинности являются альтернативным способом описания логического проекта в АБЕЛе и могут быть использованы вместо или в дополнение к булевским уравнениям и диаграммам состояний. Таблицы истинности определяют выходы как функции различных входных комбинаций в табличной форме.

Таблица истинности определяется как заголовок, описывающий формат таблицы, и самой таблицей.

3.7.1. СИНТАКСИС ЗАГОЛОВКА ТАБЛИЦЫ ИСТИННОСТИ

```
TRUTH_TABLE [IN device_id] (inputs->outputs)
        или
TRUTH_TABLE [IN device_id] (inputs:>reg_outs)
        или
TRUTH_TABLE [IN device_id] (inputs:>reg_outs->outputs)
```

device_id	идентификатор, декларирующий имя устройства, отмечающий устройство, связанное с этими узлами.
inputs	входы логической функции.
outputs	выходы логической функции.
reg_outs	регистровые (тактируемые) выходы.
->	отображает выходную функция для входов для выходов комбинационного типа.
:>	отображает выходную функция для входов для выходов регистрового типа.

Заголовок таблицы истинности может иметь одну из трех форм, показанных выше, в зависимости от того, имеет ли микросхема комбинационный выход или регистровый или оба из них.

Во всех трех формах когда в модуле используется более одной микросхемы, нужно указывать идентификатор микросхемы.

Входы и выходы (как комбинационные так и регистровые) таблицы истинности являются либо отдельными сигналами, либо, более часто, группами сигналов. Если использован только один сигнал, то его имя определено. Группа сигналов, использованная на входах или на выходах, определяется обычной групповой нотацией с сигналами, окруженными квадратными скобками и разделенными запятыми (см. главу 2.9.).

Синтаксис, показанный в первой форме, определяет формат таблицы истинности с простыми комбинационными выходами.

Вторая форма описывает формат таблицы истинности с регистровыми выходами. Символ ":>" предшествующий выходам, отмечает что эти выходы от комбинационных. Значение входов тоже определяет значения выходов, но выходы являются регистровыми или тактируемыми: они будут содержать новое значение (как это определяется входами) после следующего тактового импульса.

Третья форма является более сложной, определяя таблицу истинности и с комбинационными и с регистровыми выходами. В этом формате чтобы обеспечить различные спецификации для различных типов выходов, используются разные символы: "->" и ":>".

Примеры заголовков с сопровождающими их таблицами истинности даются после обсуждения формата таблиц истинности.

3.7.2. ФОРМАТ ТАБЛИЦЫ ИСТИННОСТИ

Таблица истинности определена в форме, описанной внутри скобок в заголовке. Таблица истинности является списком входных комбинаций и результирующих выходов. Могут быть перечислены все или некоторые из возможных входных комбинаций.

В качестве примера следующая таблица истинности определяет функцию ИСКЛЮЧАЮЩЕЕ-ИЛИ с двумя входами (A и B), одним входом разрешения (en) и одним выходом (C).

```
TRUTH_TABLE IN IC16 ([en,A,B]->C)
[0,.X.,.X.]->.X.; "don't care enable off
[1, 0 , 0 ]-> 0 ;
[1, 0 , 1 ]-> 1 ;
[1, 1 , 0 ]-> 1 ;
[1, 1 , 1 ]-> 0 ;
```

Все значения, перечисленные таблице должны быть константами, числовыми, декларированными или специальными константами ".X.". Каждая строка таблицы (каждая перечисление входов/выходов) должна иметь в конце точку с запятой.

В то время как заголовок определяет имена входов и выходов, таблица определяет значения входов и результирующие значения выходов.

Ниже следует пример показывающий описание таблицей истинности простого автомата с четырьмя состояниями и одним выходом. Текущее состояние описывается сигналами A и B, которые образуют группу. Следующее состояние описывается регистровыми выходами C и D, которые также собраны в группу. Автомат просто просчитывает через различные состояния, приводя выход E в низкий потенциал, когда A равно 1, а B равно 0.

```
TRUTH_TABLE IN IC17 ([A,B]:>[C,D]->E)
0:>1->1;
1:>2->0;
2:>3->1;
3:>0->1;
```

Отметим, что входные и выходные комбинации определяются единственным постоянным значением, а не групповой нотацией. Эта запись эквивалентна следующей:

```
[0,0]:>[0,1]->1;
[0,1]:>[1,0]->0;
[1,0]:>[1,1]->1;
[1,1]:>[0,0]->1;
```

3.8. ДИАГРАММА СОСТОЯНИЙ (STATE DIAGRAM)

Диаграммы состояний являются еще одним способом описания логики в АБЕЛе. Диаграммы состояний легко описывают операции последовательных автоматов, выполненных на программируемой логике. Автоматы поддерживаются только на микросхемах с D-триггерами.

Спецификация диаграммы состояний требует использования конструкции STATE DIAGRAM, которая определяет автомат, и операторы IF-THEN-ELSE, CASE и GOTO, которые определяют операции автомата.

Сначала обсуждается конструкция STATE DIAGRAM, а затем представлен синтаксис IF-THEN-ELSE, CASE и GOTO.

3.8.1. ОПЕРАТОР STATE DIAGRAM

```
STATE_DIAGRAM [IN device_id] state_reg

[   STATE state_exp: [equation]      ]
[                    [equation]      ]
[                                    ]
[                    trans_stmt      ]
```

device_id	идентификатор, декларирующий соответствующую микросхему.
state_reg	идентификатор, или набор идентификаторов, описывающие сигналы, определяющие текущее состояние автомата.
state_id	выражение дающее текущее состояние автомата.
equation	выражение определяющее выходы автомата.
trans_stmt	оператор IF-THEN-ELSE, CASE или GOTO.

Конструкция STATE DIAGRAM определяет автомат, соответствующий микросхеме в модуле. Автомат стартует из одного из состояний, обозначенным state exp. Уравнения, перечисленные после этого state exp, вычисляются и затем оператор_перехода (trans stmt), вычисленный после следующего тактового импульса, вынуждает автомат перейти в следующее состояние.

Уравнения, связанные с состоянием, являются не обязательными. Каждое состояние должно иметь оператор перехода. Если не встретилось ни одного из условий перехода, то следующее состояние автомата не определено. (Для некоторых микросхем переходы в неопределенное состояние вызывают переход в состояние сброшенного регистра). Состояний может быть определено так много, как это нужно.

Когда в модуле описано более одной микросхемы, требуется секция "IN device id".

Ниже приводится пример простого автомата, который продвигается из одного состояния в следующее, устанавливая выходы в текущее состояние, и затем начинает все сначала. Заметьте, что состояния не должны определяться в каком-либо порядке. Заметьте также что состояние 2 определяется выражением, а не константой. Регистр состояний состоит из сигналов "a" и "b".

```
state_diagram in u15[a,b]
    state 3     :y=3;
                 goto 0;
    state 1     :y=1;
                 goto 2;
    state 0     :y=0;
                 goto 1;
    state 1+1   :y=2;
                 goto 3;
```

Следующая диаграмма состояний определяет более сложный автомат, где state reg определен группой констант, содержащей сигналы a и b. Считая, что автомат стартует с состояния 1 (a=0, b=1), последовательность состояний должна быть 1,4,2,3,2,4, 1,4,2,3,2,4, 1...

```
current_state=[a,b] " constant declaration
    STATE_DIAGRAM current_state
        state 1: w=1;
                 y=1;
```

```
                GOTO 4;
state 2:  IF y==3 THEN 3
                  ELSE 4;
state 3:  w=2;
          y=w;
state 4:  y=3;
          CASE w==1:2;
               w==2:1;
          ENDCASE;
```

3.8.2. ОПЕРАТОР IF-THEN-ELSE

```
IF expression THEN state_exp ELSE state_exp;
expression    любое правильное выражение.
state_exp     выражение идентифицирующее следующее состояние.
```

Оператор IF-THEN-ELSE является легким способом описать переход из одного состояния в другое для автомата. Сначала вычисляется выражение, следующее за ключевым словом IF и если результат будет истинным, то автомат переходит в состояние отображенное state exp за ключевым словом THEN. Если результат выражения будет ложным, то машина переходит в состояние отображенное за ключевым словом ELSE.

Группировка IF-THEN-ELSE может быть расщеплена на несколько строк, но всегда требуется секция ELSE и завершающая точка с запятой.

Примеры:

```
if A==B then 2 else 3; "if A equals B goto state 2
if x-y then j else k; "if x-y is not 0 goto j, else goto k
if A then b*c else 4; if A is true (non-zero) goto state b*c
```

ЦЕПЬ ОПЕРАТОРОВ IF-THEN-ELSE

```
IF expression THEN state_expression
ELSE
    IF expression THEN state_expression
    ELSE
        IF expression THEN state_expression
        ELSE state_exp;
```

Операторы IF-THEN-ELSE могут легко быть каскадированы. Любое число операторов IF-THEN-ELSE может быть объединено в цепочку, но последний оператор должен содержать выражение состояния в ELSE и должен завершаться точкой с запятой. Пример цепочного оператора IF-THEN-ELSE приводится ниже:

```
if a==0 then 1
else
  if a==1 then 2
  else
    if a==2 then 3
    else 0;
```


Часто цепочки операторов IF-THEN-ELSE могут быть более четко выражены оператором CASE. Пример, показанный выше, в следующей главе выполнен с оператором CASE.

Смотрите главу 3.8.1 для примера использования IF-THEN-ELSE как они используются в диаграмме состояний.

3.8.3. ОПЕРАТОР CASE

```
CASE [ expression :state_exp; ]
     [ expression :state_exp; ]
     [ expression :state_exp; ]

ENDCASE;
expression    любое правильное выражение.
state_exp     выражение идентифицирующее следующее состояние.
```

Оператор CASE является легким способом описать переходы автомата, когда существует множество возможных условий, которые воздействуют на переходы.

Выражения, содержащиеся внутри ключевых слов CASE-ENDCASE должны быть взаимно исключающими, что означает, что только одно из выражений может быть истинно в данное время. Если два или более выражения являются истинны внутри одного оператора CASE, то результирующие уравнения не определены.

Автомат будет переходить в состояние, отображенное state exp, следующее за выражением производящим истинное значение. Если никакое выражение не является истинным, то результат не определен и результирующие действия зависят от используемой микросхемы. (Для микросхем с D-триггерами следующим состоянием будет состояние сброшенного регистра). По этой причине вы должны покрывать все возможные условия в выражениях оператора CASE. Если выражение производит числовое значение, а не логическое, то 0 является ложным, а ненулевое значение истинным.

Пример:

```
case a==0 : 1;
     a==1 : 2;
     a==2 : 3;
     a==3 : 0;
endcase;
```

Другие примеры с оператором CASE представлены в главе 3.8.1.

3.8.4. ОПЕРАТОР GOTO

```
GOTO state_exp;
state_exp     выражение идентифицирующее следующее состояние.
```

Оператор GOTO вызывает безусловный переход в состояние, выражающееся state exp.

Пример:

```
GOTO 0;     "goto state 0
GOTO x+y;   "goto the state x+y
```

3.9. ТЕСТОВЫЕ ВЕКТОРА

Тестовые вектора определяют ожидаемые функциональные операции логического устройства, определяя выходы микросхемы как функции входов. Тестовые вектора используются для симуляции внутренней модели устройства и функционального тестирования реальной запрограммированной микросхемы.

Специальная симуляционная утилита, SIMULATE, поставляется как часть программного пакета ABEL. SIMULATE симулирует операции модели микросхемы приложением входов определенных тестовыми векторами к состояниям перемычек созданными языковым процессором. SIMULATE обсуждается подробнее в ABEL User's Guide.

Функциональное тестирование реальной микросхемы выполняется логическим программатором после того как микросхема запрограммирована. Это может быть сделано, так как тестовые вектора стали частью загрузочного файла программатора, который загружается в логический программатор.

Тестовые вектора пишутся для каждой отдельной микросхемы в модуле, так что различные характеристики каждой микросхемы могут быть рассмотрены раздельно в течении симуляции.

Тестовые вектора для микросхемы определяются таблицей. Таблица состоит из заголовка и самих векторов. Заголовок отмечает, что за ним следуют тестовые вектора и определяет формат таблицы. Эти вектора определяют функции от входа до выхода.

3.9.1. ФОРМАТ ТАБЛИЦЫ ТЕСТОВЫХ ВЕКТОРОВ

```
TEST_VECTORS [IN device_id] [note] (inputs->outputs)
[ invalues -> outvalues; ]
                :
                .
                .
```

device_id	идентификатор, декларирующий соответствующую микросхему.
note	строка, использованная чтобы описать тестовые вектора.
inputs	идентификатор или группа идентификаторов, определяющая имена входных сигналов микросхемы.
outputs	идентификатор или группа идентификаторов, определяющая имена выходных сигналов микросхемы.
invalues	входное значение или группа входных значений.
outvalues	выходное значение или группа выходных значений.

Форма тестовых векторов определяется заголовком. Каждый вектор определен в формате описанном внутри скобок в операторе заголовка. В заголовке может быть определена дополнительная строка примечания. Эта строка примечания часто используется чтобы описать векторный тест и она включается в выходной файл симуляции, в файл документации и в загрузочный файл программатора в формате JEDEC.

Таблица перечисляет входные комбинации и их результирующие выходы. Могут быть перечислены все или некоторые из возможных входных комбинаций.

Все значения, определенные в таблице, должны быть константами, либо декларированными, либо числовыми или специальной константой ".X.". Каждая строка таблицы (каждое перечисление входов/выходов) должно заканчиваться точкой с запятой.

Ниже приводится простая таблица тестовых векторов:

```
TEST_VECTORS
([A,B]->[C,D])

[0,0]->[1,1];
[0,1]->[1,0];
[1,0]->[0,1];
[1,1]->[0,0];
```

Следующая таблица векторов эквивалентна таблице определенной выше, так как значения для групп могут быть определены числовыми константами:

```
TEST_VECTORS
([A,B]->[C,D])

0->3;
1->2;
2->1;
3->0;
```

ГЛАВА 4 ДИРЕКТИВЫ

Директивы обеспечивают многие дополнительные возможности, которые могут воздействовать на содержимое исходного файла, когда он обрабатывается. Секции исходного абелевского кода могут быть включены условно, в исходный файл может быть перенесен код из другого файла, в течении обработки исходного файла могут быть напечатаны сообщения.

Директивы предназначены для разработчика, который понимает основы АБЕЛя и хочет использовать более сложные структуры. Таблица 4-1 перечисляет возможные директивы, которые затем описываются в последующих разделах. В главе 4.5 руководства ABEL Applications Guide даны примеры использования директив чтобы создать тестовые вектора.

Некоторые из директив принимают аргументы, которые используются чтобы задать некоторые условия. Когда это имеет место, аргумент может быть фактическим аргументом или фиктивным аргументом, которому предшествует вопросительный знак.

Правила применимые к фактическим и фиктивным аргументам представлены в главе 2.11.

Таблица 4-1. Директивы

@ALTERNATE	@IFIDEN	@MESSAGE
@CONST	@IFNB	@PAGE
@EXPR	@IFNDEF	@RADIX
@EXIT	@IFNIDEN	@REPEAT
@IF	@INCLUDE	@STANDARD
@IFB	@IRP	
@IFDEF	@IRPC	

4.1. ДИРЕКТИВА @ALTERNATE

@ALTERNATE

@ALTERNATE позволяет переключить стандартный набор абелевских операторов на альтернативный набор. Это сделано для пользователей, которые чувствуют себя более комфортабельно с альтернативным набором вследствии их знакомства с операторами, используемыми в других языках.

Альтернативный набор остается активным до употребления директивы @STANDARD или до достижения конца файла.

Альтернативный набор операторов приведен в таблице 4-2.

Таблица 4-2. Альтернативный набор операторов

Абелевский оператор	Альтернативный оператор	Описание	
!	/	НЕ	(NOT)
&	*	И	(AND)
#	+	ИЛИ	(OR)
$	:+:	Искл.ИЛИ	(XOR)
!$	:*:	Вкл.ИЛИ	(XNOR)

Заметьте, что использование альтернативного набора операторов исключает использование в АБЕЛе операторов сложения, умножения и деления, поскольку в альтернативном наборе они представляют логические операторы И, ИЛИ и НЕ.

4.2. ДИРЕКТИВА @CONST

```
@CONST id = expression
```

id	любой правильный идентификатор.
expression	любое правильное выражение.

@CONST позволяет сделать в исходном файле декларацию новой константы вне обычной (и обязательной) секции декларации.

Директива @CONST предназначена для использования внутри макросов, так что они могут определять свои собственные константы. Константы, определенные с помощью @CONST, переопределяют любые предыдущие декларации констант. Декларирование идентификатора таким способом как константы вызывает ошибку, если идентификатор был использован ранее в исходном файле как что-то иное, нежели константа (например, macro, pin, device).

Пример:

```
@CONST count = count + 1;
```

4.3. ДИРЕКТИВА @EXIT

```
@EXIT
```

Директива @EXIT вынуждает программу предварительного разбора прекратить обработку исходного файла с установкой бита ошибки. (Биты ошибки позволяют операционной системе определить что произошла ошибка обработки).

4.4. ДИРЕКТИВА @EXPR

```
@EXPR [block] expression;
```

block	некоторый блок.
expression	любое правильное выражение.

@EXPR вычисляет данное выражение и преобразует его в строку цифр, в базе исчисления, которая установлена по умолчанию. Эта строка и блок затем вставляются в исходный файл в той точке, где встретилась директива @EXPR. Выражение должно произвести правильные цифры.

Пример:

```
@expr {ABC} ^B11;
```

Считая, что база исчисления по умолчанию равна десяти, этот пример вызовет вставление текста ABC3 в исходный файл.

4.5. ДИРЕКТИВА @IF

```
@IF expression block
```

expression	правильное выражение, которое производит числовое значение.
block	правильный блок текста, как это описано в главе 2.

@IF используется чтобы включить секцию абелевского исходного кода в зависимости от результирующего значения выражения. Если выражение не равно нулю (логическое ИСТИННО), то блок текста включается как часть исходного.

В выражении допускается подстановка фиктивных аргументов.

Пример:

```
@IF (A > 17) { C = D $ F; }
```

4.6. ДИРЕКТИВА @IFB (IF Blank)

```
@IFB (arg) block
```

arg	либо фактический аргумент, либо фиктивный аргумент, которому предшествует вопросительный знак.
block	некоторый блок текста.

@IFB включает текст, содержащийся внутри блока, если аргумент пуст (содержит 0 символов).

Примеры:

```
@IFB()
{ text here will be included
with the rest of the source file.
}

@IFB (hello)
{ this text will not be included }

@IFB (?A)
{ this text will be included if no value is substituted
for A. }
```


4.7. ДИРЕКТИВА @IFDEF (IF Defined)

```
@IFDEF id block
```

id	идентификатор.
block	некоторый блок текста.

@IFDEF включает текст, содержащийся внутри блока, если этот идентификатор определен.

Пример:

```
A pin 5;
@ifdef A {Base = ^hE000; }
"the above assignement is made because A was
"defined
```

4.8. ДИРЕКТИВА @IFIDEN (IF Identical)

```
@IFIDEN (arg1,arg2) block
```

arg1,2	либо фактический аргумент, либо фиктивный аргумент, которому предшествует вопросительный знак.
block	некоторый блок текста.

@IFIDEN включает текст, содержащийся внутри блока, если аргументы arg1 и arg2 являются идентичными.

Пример:

```
@ifiden (?A,abcd) {?A device 'P16R4'}
```

Делается декларация для P16R4 если фактический аргумент, подставленный для A является идентичным "abcd".

4.9. ДИРЕКТИВА @IFNB (IF Not Blank)

```
@IFNB (arg) block
```

arg	либо фактический аргумент, либо фиктивный аргумент, которому предшествует вопросительный знак.
block	некоторый блок текста.

@IFB включает текст, содержащийся внутри блока, если аргумент не пуст (содержит ненулевое количество символов).

Примеры:

```
@IFNB()
{ ABEL source here will not be included
with the rest of the source file.
}
@IFNB (hello)
{ this text will be included }
@IFNB (?A)
{ this text will be included if a value is substituted for A.
}
```

4.10. ДИРЕКТИВА @IFNDEF (IF Not Defined)

```
@IFNDEF id block
```

id	идентификатор.
block	некоторый блок текста.

@IFNDEF включает текст, содержащийся внутри блока, если этот идентификатор не определен.

Пример:

```
@ifndef A {Base = ^hE000; }
"if A not defined, the block is inserted in the text
```

4.11. ДИРЕКТИВА @IFNIDEN (IF Not Identical)

```
@IFnIDEN (arg1,arg2) block
```

arg1,2	либо фактический аргумент, либо фиктивный аргумент, которому предшествует вопросительный знак.
block	некоторый блок текста.

@IFIDEN включает текст, содержащийся внутри блока, если аргументы arg1 и arg2 не являются идентичными.

Пример:

```
@ifniden (?A,abcd) {?A device 'P16R8'}
```

Делается декларация для P16R8 если фактический аргумент, подставленный для A не является идентичным "abcd".

4.12. ДИРЕКТИВА @INCLUDE

```
@INCLUDE filespec
```

filespec	строка, определяющая имя файла по правилам используемой операционной системы.

@INCLUDE вызывает копирование в исходный абелевский файл содержимого файла, определенного спецификацией. Включение информации начнется с точки расположения директивы @INCLUDE. Спецификация файла может включать спецификацию диска или пути, которые отображают местонахождение файла. Если не дано никакой спецификации диска или пути, то предполагается что файл находится либо на диске и пути текущих, либо на диске или пути определенных параметром -II (см. главу 4.2 ABEL User's Guide).

Пример:

```
@INCLUDE 'macros.abl' "file specification
```


4.13. ДИРЕКТИВА @IRP (Indefinite Repeat)

```
@IRP dummy_arg (arg[,arg]...) block
```

dummy_arg	фиктивный аргумент.
arg	фактический аргумент либо имя фиктивного аргумента, которому предшествует вопросительный знак "?".
block	блок текста.

@IRP вызывает повторение некоторого блока в исходном файле n раз, где n равно числу аргументов содержащихся в скобках. Каждый раз когда блок повторяется, фиктивный аргумент принимает значение следующего в последовательности аргумента.

Пример:

```
@IRP A (1,^H0A,0)
{ B = ?A ;
}
```

это преобразуется в:

```
B=1;
B=^H0A;
B=0;
```

Полученная последовательность будет вставлена в исходный файл на месте расположения директивы @IRP.

Заметим, что если директива будет определена следующим образом:

```
@IRP A (1,^H0A,0) { B = ?A ; }
```

то результирующий текст будет выглядеть следующим образом:

```
B=1; B=^H0A; B=0;
```

Весь текст появился в одной строке потому что блок в определении @IRP не содержит конца строки. Помните, что пробелы и конец строки являются существенными в блоках.

4.14. ДИРЕКТИВА @IRPC (Indefinite Repeat, Character)

```
@IRPC dummy_arg (arg) block
```

dummy_arg	фиктивный аргумент.
arg	фактический аргумент либо имя фиктивного аргумента, которому предшествует вопросительный знак "?".
block	блок текста.

@IRPC вызывает повторение некоторого блока в исходном файле n раз, где n равно числу символов содержащихся в "arg". Каждый раз когда блок повторяется, фиктивный аргумент принимает значение следующего в последовательности символа.

Пример:

```
@IRPC A (Cat)
{ B = ?A ;
}
```

это преобразуется в:

```
B=C;
B=a;
B=t;
```

Полученная последовательность будет вставлена в исходный файл на месте расположения директивы @IRPC.

4.15. ДИРЕКТИВА @MESSAGE

```
@MESSAGE string
string          некоторая строка.
```

@MESSAGE печатает сообщение, определенное в "string" на терминал. Это может быть использовано чтобы наблюдать процесс обработки на стадии предварительного разбора.

Пример:

```
@message 'includes completed'
```

4.16. ДИРЕКТИВА @PAGE

```
@PAGE
```

Эта директива посылает в файл листинга предварительного разбора символ FORM FEED. Если никакой листинг не создается, @PAGE не имеет эффекта.

4.17. ДИРЕКТИВА @RADIX

```
@RADIX expr
expr            правильное выражение, которое производит
                числа 10 или 16 чтобы отметить новую базу
                исчисления по умолчанию.
```

@RADIX используется чтобы изменить базу исчисления по умолчанию системы. Нормально по умолчанию база равна 10 (десятичная). Эта директива полезна, когда нужно определить много чисел в базе не десятичной, а, скажем, в двоичной. Можно включить директиву @RADIX и все числа, которые не имеют явно указанной базы, будут трактоваться в новой базе, в этом случае в двоичной. (См. главу 2.6.)

Заново определенная база исчисления по умолчанию остается действующей до тех пор пока не встретится другая директива @RADIX или пока не будет достигнут конец модуля.

Когда база исчисления по умолчанию установлена 16, все числа, которые в этой базе и начинаются с буквы, должны начинаться с ведущего нуля.

Примеры:

```
@radix 2;           "change default base to binary
@radix 11011/11+1;  "change back to decimal
```


4.18. ДИРЕКТИВА @REPEAT

```
@REPEAT expr block
expr            правильное выражение, которое производит
                число.
block           некоторый блок текста.
```

@REPEAT включает текст, содержащийся внутри блока, "n" раз, где "n" определяется выражением.

Нижеследующий пример использования директивы

```
@repeat 5 {H,}
```

дает в результате текст "H,H,H,H,H", который вставляется в исходный файл. Директива @REPEAT полезна при генерировании длинных таблиц истинности и наборов тестовых векторов. Примеры использования @REPEAT могут быть найдены в ABEL Applications Guide.

4.19. ДИРЕКТИВА @STANDARD

```
@STANDARD
```

Директива @STANDARD переключает набор используемых абелевских операторов обратно к стандартному. Действие директивы обратно директиве @ALTERNATE.