3.1. ОБЩИЕ СВЕДЕНИЯ

Язык описания аппаратуры AHDL разработан фирмой Altera и предназначен для описания комбинационных и последовательностных логических устройств, групповых операций, цифровых автоматов (state machine) и таблиц истинности с учетом архитектурных особенностей ПЛИС фирмы Altera Он полностью интегрируется с системой автоматизированного проектирования ПЛИС MAX+PLUS II Файлы описания аппаратуры, написанные на языке AHDL, имеют расширение * TDF (Text design file) Для создания TDF-файла можно использовать как текстовый редактор системы MAX+PLUS II, так и любой другой Проект, выполненный в виде TDF-файла, компилируется, отлаживается и используется для формирования файла программирования или загрузки ПЛИС фирмы Altera

Операторы и элементы языка AHDL являются достаточно мощным и универсальным удобным в использовании средством описания алгоритмов функционирования цифровых устройств Язык описания аппаратуры AHDL дает возможность создавать иерархические проекты в рамках одного этого языка или же в иерархическом проекте использовать как TDF-файлы, разработанные на языке AHDL, так и другие типы файлов Для создания проектов на AHDL можно, конечно, пользоваться любым текстовым редактором, но текстовый редактор системы MAX+PLUS II предоставляет ряд дополнительных возможностей для ввода, компиляции и отладки проектов (см гл 2)

Проекты, созданные на языке AHDL, легко внедряются в иерархическую структуру Система MAX+PLUS II позволяет автоматически создать символ компонента, алгоритм функционирования которого описывается TDF-файлом, и затем вставить его в файл схемного описания (GDF-файл) Подобным же образом можно вводить собственные функции разработчика помимо порядка 300 макрофункций, разработанных фирмой Altera, в любой TDF-файл Для всех функций, включенных в макробиблиотеку системы MAX+PLUS II, фирма **Altera** поставляет файлы с расширением * **inc** которые используются в операторе включения **INCLUDE**

При распределении ресурсов устройств разработчик может пользоваться командами текстового редактора или операторами языка AHDL Кроме того, разработчик может только проверить синтаксис или выполнить полную компиляцию для отладки и запуска проекта Любые ошибки автоматически обнаруживаются обработчиком сообщений и высвечиваются в окне текстового редактора

При работе с AHDL следует соблюдать тек называемые "Золотые правила" (Golden Rules) Это позволит эффективно применять язык AHDL и избежать многих ошибок

- соблюдайте форматы и правила присвоения имен, описанные в руководстве по стилям AHDL, чтобы программа была читаемой и содержала меньше ошибок,
- несмотря на то что язык AHDL не различает прописные и строчные буквы, **Altera** рекомендует для улучшения читаемости использовать прописные буквы для ключевых слов,
- не применяйте вложенные конструкции условного оператора **If**, если можно использовать оператор выбора **Case**,
- строка в TDF-файле может быть длиной до 255 символов Однако следует стремится к длине строки, умещающейся на экране Строки заканчиваются нажатием клавиши **Enter**,
- новую строку можно начинать в любом свободном месте, т е на местах пустых строк, табуляции, пробелов Основные конструкции языка отделяются пустым пространством,
- ключевые слова, имена и числа должны разделяться соответствующими символами или операторами и/или одним или более пробелами,
- комментарии должны быть заключены в символы процента (%). Комментарий может включать любой символ, кроме символа процента, поскольку компилятор системы MAX+PLUS II игнорирует все, заключенное в символы процента Комментарии не могут быть вложенными;
- при соединении одного примитива с другим используйте только "разрешенные" связи между ними, не все примитивы могут соединяться друг с другом;
- используйте только макрофункции **EXPDFF, EXPLATCH, NANDLTCH и NORLTCH**, входящие в макробиблиотеку системы MAX+PLUS II Не создавайте свои собственные структуры перекрестных связей Избегайте многократного связывания вместе **EXPDFF, EXPLATCH, NANDLTCH** и **NORLTCH** Многочисленные примеры этих макрофункций должны всегда разделяться примитивами **LCELL**

***Общие "золотые прввилв" вводв проектв*:**

- если многочисленные двунаправленные или выходные выводы связаны вместе, разработчик не может использовать оператор **Pin Connection** для соединения выводов при функциональном моделировании с аппаратной поддержкой или функциональном тестировании,
- нет необходимости создавать прототипы функций для примитивов Однако разработчик может переопределить примитивы в объявлениях прототипов функций для изменения порядка вызова входов в вашем TDF-ФАЙЛЕ,
- не редактируйте файл **Fit** Если разработчик желает отредактировать назначения для проекта, необходимо сохранить сначала файл Fit как TDF-файл или сделать обратное назначение с помощью команды **Project Back-Annotate** и отредактировать их с помощью команд **Chip to Device, Pin/LC/Chip** и **Enter Assignments**;
- если разработчик хочет загрузить регистр по определенному фронту глобального тактового сигнала *Clock*, фирма Altera рекомендует, когда регистр загружен, использовать для управления вход *Clock Enable* одного из триггеров типа Enable: **DFFE, TFFE, JKFFE** или **SRFFE**,
- когда разработчик начинает работать с новым файлом проекта, сразу же необходимо задать семейство ПЛИС, на которые ориентирован проект, с помощью конструкции **Family**, для того чтобы в дальнейшем иметь возможность воспользоваться макрофункциями, специфичными для данного семейства Если разработчик не задаст семейство, оно будет считаться таким же, как и в текущем проекте,
- используйте опцию **Design Doctor** для проверки надежности логики проекта во время компиляции,
- предоставляемые по умолчанию фирмой Altera стили для логического синтеза имеют разные установки для разных семейств устройств, что обеспечивает более эффективное использование архитектуры каждого устройства Когда разработчик использует какой-нибудь из этих стилей, его установки изменятся, при переходе к другому семейству устройств После смены семейства необходимо проверить новые установки стиля

Общие "золотые праавиле" системы MAX+PLUS II:

- в начале работы над новым проектом рекомендуется сразу же задать его имя как имя нового проекта с помощью меню **Project Name**, для того чтобы в дальнейшем его было бы легко компилировать Позже разработчик всегда может изменить имя проекта,
- используйте иерархические возможности системы MAX+PLUS II для перемещения между файлами проекта Для открытия файла нижнего уровня иерархии откройте файл верхнего уровня и затем используйте окно **Hierarchy Display** или **Hierarchy Down** и откройте файл более низкого уровня. Если разработчик выберает **Open** или **Retrieve** для открытия файла нижнего уровня, считается, что он является файлом верхнего уровня нового дерева и все назначения ресурсов, устройств и зондов сохраняются только для этой новой иерархии, а не для текущего проекта,
- если разработчик создает вспомогательный файл для какого-нибудь проекта, пиктограмма этого файла появится в окне **Hierarchy Display**, в том случае, если используется то же имя, что и для проекта,
- не рекомендуется редактировать системные файлы MAX+PLUS II, в том числе файлы с расширением **.prb**, файлы **HIF** или **maxplus2.ini** либо файлы **<имя проекта>.ini**,
- если разработчик желает переименовать файл проекта или вспомогательный файл, следует использовать команду **Save As...** Не рекомендуется переименовывать файлы проекта вне среды MAX+PLUS II (например, из MS-DOS или Windows File Manager),
- после завершения работы над проектом рекомендуется выполнить команду архивирования проекта **Project Archive** для создания резервной копии всего проекта (всех его файлов) На архивированную копию не повлияют никакие последующие редактирования

3.2. ИСПОЛЬЗОВАНИЕ ЧИСЕЛ И КОНСТАНТ В ЯЗЫКЕ AHDL

3.2.1. Использование чисел

Числа используются для представления констант в булевых выражениях и уравнениях Язык AHDL поддерживает все комбинации десятичных, двоичных, восьмеричных и шестнадцатеричных чисел

Ниже приведен файл **decode1.tdf**, который представляет собой дешифратор адреса, генерирующий высокий активный уровень сигнала разрешения доступа к шине, если адрес равен шестнадцатеричному числу 370h

```
SUBDESIGN decode1
    (
      address[15..0] : INPUT;
      chip_enable . OUTPUT;
    )
BEGIN
    chip_enable = (address[15..0] == H"0370");
END;
```

В этом примере десятичные числа использованы для указания размерности массива бит, которым записывается адрес шины Шестнадцатеричным числом H"0370" записано значение адреса, при котором устанавливается высокий уровень сигнала

3.2.2. Использование констант

В файле AHDL можно использовать константы для описательных имен разных чисел. Такое имя, используемое на протяжении всего файла, может быть более информативным, чем число, например, имя UPPER_LI несет больше информации, чем, например, число 103. В языке AHDL константы вводятся объявлением CONSTANT

Приведенный выше файл можно записать по-другому, используя вместо числа H"0370" константу IO_ADDRESS.

```
CONSTANT IO_ADDRESS = H"0370";
SUBDESIGN decode2
    (
      a[15..0] : INPUT;
      ce : OUTPUT;
    )
BEGIN
    ce = (a[15..0] == IO_ADDRESS);
END;
```

Преимущество использования констант особенно заметно, если одно и то же число используется в файле несколько раз Тогда, если его нужно изменить, меняют его только один раз в объявлении константы

3.3. КОМБИНАЦИОННАЯ ЛОГИКА

Как известно, логическая схема называется комбинационной, если в заданный момент времени выходы являются только функциями входов в этот момент времени Комбинационная логика в языке AHDL реализована булевыми выражениями и уравнениями, таблицами истинности и большим количеством макрофункций В число примеров комбинаторных логических функций входят дешифраторы, мультиплексоры и сумматоры

3.3.1. Реализация булевых выражений и уравнений

Булевы выражения — это множества узлов, чисел, констант и других булевых выражений, выделяемых операторами, компараторами и, возможно, сгруппированные в заключающих круглых скобках Булево уравнение устанавливает равенство между узлом или группой и булевым выражением

В качестве примера приведен файл boole1.tdf, в котором даны два простых булевых выражения, представляющих два логических элемента

```
SUBDESIGN boole1
        (
          a0, a1, b : INPUT;
          out1, out2 · OUTPUT,
        )
    BEGIN
        out1 = a1 & !a0;
        out2 = out1 # b;
    END;
```

Здесь на выход out1 выдается результат логической операции И, примененной ко входу a1 и инвертированному входу a0, а на выход out2 выдается результат применения логической операции ИЛИ к выходу **out1** и входу **b** Поскольку эти уравнения обрабатываются одновременно, последовательность их следования в файле не важна

3.3.2. Объявление NODE (узел)

Узел, который объявляется в секции переменных VARIABLE в объявлении NODE, можно использовать для хранения промежуточных выражений

Это полезно делать, если булево выражение повторяется несколько раз и его целесообразно заменить именем узла. Приведенный выше файл boole1 tdf можно переписать по-другому.

```
SUBDESIGN boole2
    (
      a0, a1, b : INPUT,
      out : OUTPUT,
    )
VARIABLE
    a_equals_2 : NODE;
BEGIN
    a_equals_2 = a1 & !a0,
    out = a_equals_2 # b,
END,
```

Здесь объявляется узел a_equals_2 и ему присваивается значение выражения **a1 & !a0**. Использование узлов помогает экономить ресурсы устройств, если узел используется в нескольких выражениях

3.3.3. Определение групп

Группа может включать в себя до 256 элементов (бит), рассматриваться как совокупность узлов и участвовать в различных действиях как единое целое В булевых уравнениях группа может быть приравнена булевому выражению, другой группе, одному узлу, VCC, GND,1 или 0 В каждом случае значения группы разные.

Если группа определена, для краткого указания всего диапазона ставят две квадратные скобки [ ] Например, группу a[4 1] можно кратко записать как a[ ]

Примеры определения групп:

- при выполнении присваивания операции для двух групп обе должны иметь одинаковое число бит В приводимом ниже примере каждый бит первой группы соединяется с соответствующим битом второй группы, а именно d2 соединяется с q8, d1 с q7 и d0 с q6

```
d[2. 0] = q[8..6]
```

- при назначении группы одному узлу все биты группы присоединяются к узлу В приводимом ниже примере d1, d2 и d0 присоединяются все вместе к n

```
d[2 .0] = n
```

- при выполнении операции присваивания группы значения VCC или GND все биты группы присоединяются к этому значению В приводимом ниже примере d1, d2 и d0 присоединяются все вместе к VCC

```
d[2..0] = VCC
```

- при присваивании группе значения 1 младший бит группы присоединяется к VCC, а все другие биты — к значению GND В приводимом ниже примере только d0 присоединяется к VCC, значение 1 расширяется до представления B"001"

```
d[2 .0] = 1
```

3.3.4. Реализация условной логики

Условная логика делает выбор между режимами в зависимости от состояния сигнала на логических входах Для реализации условной логики используются операторы IF или CASE

В операторе IF оценивается одно или несколько булевых выражений и затем описываются режимы для разных значений этих выражений

В операторе CASE дается список альтернатив, которые имеются для каждого возможного значения некоторого выражения Оператор оценивает значение выражения и по нему выбирает режим в соответствии со списком

3 3 4 1 ЛОГИКА ОПЕРАТОРА IF

В качестве примера рассмотрим файл **priority.tdf**, в котором описан кодировщик приоритета, который преобразует уровень самого приоритетного активного входа в значение Он генерирует двухразрядный код, показывающий вход с наивысшим приоритетом, имеющий уровень **VCC**

```
SUBDESIGN priority
    (
      low, middle, high : INPUT;
      highest_level[1. 0] . OUTPUT,
    )
BEGIN
    IF high THEN
        highest_level[] = 3;
    ELSIF middle THEN
        highest_level[] = 2,
    ELSIF low THEN
        highest_level[] = 1;
    ELSE
        highest_level[] = 0,
    END IF,
END;
```

Здесь оцениваются входы **low**, **middle** и **high**, чтобы определить, равны ли они **VCC** На выходе получится код, соответствующий приоритету того входа, который был равен **VCC** Если ни один вход не запущен, значение кода станет **0**

3 3 4 2 ЛОГИКА ОПЕРАТОРА CASE

В качестве примера рассмотрим файл **decoder.tdf**, реализующий функции дешифратора, преобразующего код из двухразрядного в четырехразрядный Дешифратор преобразует двухразрядное число в прямом коде в четырехразрядный код "1 из 4"

```
SUBDESIGN decoder
    (
      code[1..0] . INPUT,
      out[3..0] . OUTPUT,
    )
BEGIN
    CASE code[] IS
WHEN 0 => out[] = B"0001",
WHEN 1 => out[] = B"0010";
WHEN 2 => out[] = B"0100",
WHEN 3 => out[] = B"1000",
    END CASE,
END;
```

Здесь группа входа **code [1..2]** может принимать значения 0, 1, 2, 3. В зависимости от реального кода активизируется только одна соответствующая ветвь оператора в данный моменг времени На-

пример, если на входе **code []** сигнал принимает значение 1, на выходе **out** устанавливается значение B"0010"

3.3.4.3. СРАВНЕНИЕ ОПЕРАТОРОВ IF И CASE

Операторы IF и CASE похожи. Иногда использование любого из них приводит к одним и тем же результатам.

If Then	Case

IF a[] == 0 THEN
  y = c & d;
ELSIF a[] == 1 THEN
  y = e & f;
ELSIF a[] == 2 THEN
  y = g & h;
ELSIF a[] == 3 THEN
  y = 1;
ELSE
  y = GND;
END IF;
	CASE a[]
	  WHEN 0
	  y = c & d;
	  WHEN 1
	  y = e & f;
	  WHEN 2
	  y = g & h
	  WHEN 3
	  y = 1;
	  WHEN OTHERS =>
	  y = GND;
	END CASE;

Однако между этими двумя операторами существует несколько важных различий.

- в операторе IF можно использовать любое булево выражение. Каждое выражение, записываемое в предложении, следующем за IF или ELSEIF, может не быть связанным с другими выражениями в операторе. В операторе CASE одно единственное выражение сравнивается с проверяемым значением;
- в результате интерпретации оператора IF может быть сгенерирована логика, слишком сложная для компилятора системы MAX+PLUS II. В приведенном ниже примере показано, как компилятор интерпретирует оператор IF. Если a и b — сложные выражения, то инверсия каждого из них даст еще более сложное выражение.

```
IF a THEN                IF a THEN
 c = d,                   c = d;
                         END IF;

ELSIF b THEN             IF !a & b THEN
 c = e,                   c = e,
                         END IF;

ELSE                     IF !a & !b THEN
 c = f;                   c = f;
END IF;                  END IF;
```

3.3.5. Описание дешифраторов

Дешифратор содержит комбинаторную логику, которая преобразует входные сигналы схемы в выходные значения или задает выходные значения для входных сигналов. Для создания дешифратора в языке AHDL используется объявление таблицы истинности TABLE.

Ниже приведен файл **7segment.tdf**, представляющий собой дешифратор, который задает схему управления светодиодами. Светодиоды светятся на семисегментном дисплее для индикации шестнадцатеричной цифры (от 0 до 9 и буквы от A до F).

```
% -a- %
% f|  |b %
% -g- %
% e|  |c %
% -d- %
% %
% 0 1 2 3 4 5 6 7 8 9 A b C d E F %
% %

SUBDESIGN 7segment
(
 i[3..0] : INPUT;
 a, b, c, d, e, f, g : OUTPUT;
)
BEGIN
 TABLE
 i[3..0] => a, b, c, d, e, f, g;

 H"0" => 1, 1, 1, 1, 1, 1, 0;
 H"1" => 0, 1, 1, 0, 0, 0, 0;
 H"2" => 1, 1, 0, 1, 1, 0, 1;
 H"3" => 1, 1, 1, 1, 0, 0, 1;
 H"4" => 0, 1, 1, 0, 0, 1, 1;
 H"5" => 1, 0, 1, 1, 0, 1, 1;
 H"6" => 1, 0, 1, 1, 1, 1, 1;
 H"7" => 1, 1, 1, 0, 0, 0, 0;
 H"8" => 1, 1, 1, 1, 1, 1, 1;
 H"9" => 1, 1, 1, 1, 0, 1, 1;
 H"A" => 1, 1, 1, 0, 1, 1, 1;
 H"B" => 0, 0, 1, 1, 1, 1, 1,
 H"C" => 1, 0, 0, 1, 1, 1, 0;
 H"D" => 0, 1, 1, 1, 1, 0, 1;
 H"E" => 1, 0, 0, 1, 1, 1, 1;
 H"F" => 1, 0, 0, 0, 1, 1, 1;
 END TABLE,
END,
```

В этом примере учтены все возможные шестнадцатеричные цифры (**i[3..0]**) и соответствующие состояния светодиодов (**a, b, c, d, e, f, g**), которые обеспечивают "начертание" цифры на дисплее. Изображение светодиодов на дисплее дано в виде комментария перед файлом.

Ниже приведен файл дешифратора адреса **decode3.tdf** для шестнадцатиразрядной микропроцессорной системы.

```
SUBDESIGN decode3
(
 addr[15..0], m/io : INPUT;
 rom, ram, print, sp[2..1] : OUTPUT;
)
BEGIN
 TABLE
 m/io, addr[15..0] => rom, ram, print, sp[];
 1, B"00XXXXXXXXXXXXXX" => 1, 0, 0, B"00";
 1, B"100XXXXXXXXXXXX" => 0, 1, 0, B"00";
 0, B"0000001010101110" => 0, 0, 1, B"00";
 0, B"0000001011011110" => 0, 0, 0, B"01";
 0, B"0000001101110000" => 0, 0, 0, B"10";
 END TABLE;
END;
```

В данном примере существуют тысячи возможных вариантов входа (адреса), поэтому было бы непрактично сводить их все в таблицу. Вместо этого, можно пометить символом "X" несущественные, не влияющие на выход, разряды. Например, выходной сигнал **rom** (ПЗУ) будет высоким для всех 16384 вариантов адреса **ad-**

dr[15..0], которые начинаются с 00. Поэтому вам нужно только указать общую для всех вариантов часть кода, т е 00, а в остальных разрядах кода поставить "X" Такой прием позволит сделать проект, требующий меньше устройств и ресурсов

Приведенный ниже пример **decode4.tdf** показывает использование стандартной параметризируемой функции **lpm_decode** в задаче разработки дешифратора, аналогичной примеру **decode1.tdf**

```
INCLUDE "lpm_decode.inc";

SUBDESIGN decode4
(
 address[15..0] : INPUT;
 chip_enable . OUTPUT;
)
BEGIN
 chip_enable = lpm_decode(.data[]=address[])
 WITH (LPM_WIDTH=16, LPM_DECODES=2^10)
 RETURNS (.eq[H"0370"]);
END;
```

3.3.6. Использование для переменных значений по умолчанию

Можно определить значения по умолчанию для узла или группы, которые будут автоматически использоваться, если в файле их значения не будут заданы Язык AHDL позволяет присваивать значение узлу или группе в файле неоднократно Если при этом произойдет конфликт, система автоматически будет использовать значения по умолчанию Если значения по умолчанию не были заданы, используется значение GND

Объявление значений по умолчанию DEFAULTS можно использовать для задания переменных в таблице истинности, операторах IF и CASE

Примечание. *Не следует путать значения по умолчанию для переменных со значениями по умолчанию для портов которые задаются в секции SUBDESIGN*

Ниже приводится файл **default1.tdf**, в котором происходит оценка входов и выбор соответствующего ASCII кода

```
SUBDESIGN default1
    (
      i[3..0] : INPUT;
      ascii_code[7. 0] . OUTPUT,
    )
 BEGIN
      DEFAULTS
 asci_code[] = B"00111111";% ASCII question mark"?"
% END DEFAULTS,
      TABLE
                          i[3  0] => asci_code[];
                  B"1000" => B"01100001", % "a" %
 B"0100" => B"01100010"; % "b" %
 B"0010" => B"01100011"; % "c" %
 B"0001" => B"01100100", % "d" %
      END TABLE,
 END,
```

Если значение входа совпадает с одним из значений в левой части таблицы, код на выходе приобретает соответствующее значение ASCII кода в правой части таблицы Если входное значение не совпадает ни с одним из (левых) табличных значений, выходу будет присвоено значение по умолчанию B"00111111" (вопросительный знак)

В приведенном ниже файле **default2.tdf** показано, как при многократном присваивании узлу разных значений возникает конфликт, и как он разрешается средствами AHDL

```
SUBDESIGN default2
    (
a, b, c . INPUT,
select_a, select_b, select_c · INPUT;
wire_or, wire_and . OUTPUT,
    )
 BEGIN
      DEFAULTS
              wire_or = GND,
              wire_and = VCC,
      END DEFAULTS,
      IF select_a THEN
                      wire_or = a;
                      wire_and = a,
      END IF,
  IF select_b THEN
                      wire_or = b;
                      wire_and = b,
      END IF,
      IF select_c THEN
                      wire_or = c;
                      wire_and = c;
      END IF,
 END;
```

В данном примере выход **wire_or** устанавливается равным **a**, **b**, или **c** в зависимости от состояния входных сигналов **select_a**, **select_b** и **select_c.** Если ни один из них не равен VCC, то выход **wire_or** принимает значение по умолчанию, равное GND

Если более чем один сигнал (**select_a**, **select_b** или **select_c**) равен VCC, то **wire_or** равно результату логической операции ИЛИ над соответствующими входными сигналами Например, если **select_a** и **select_b** равны VCC, то **wire_or** равно **a** ИЛИ **b**

С сигналом **wire_and** производятся аналогичные действия, но он становится равным VCC, когда входные сигналы **select** равны VCC, и равен логическому И от соответствующих входных сигналов, если более чем один из них равен VCC

3.3.7. Реализация логики с активным низким уровнем

Значение сигнала с низким активным уровнем равно GND Сигналы с низким активным уровнем могут быть использованы для управления памятью, периферийными микропроцессорными устройствами

Ниже приводится файл **daisy.tdf**, который представляет модуль арбитражной схемы для соединения цепочкой (daisy chain) Данный модуль делает запрос на доступ к шине для предыдущего (в цепочке) модуля Он получает запрос на доступ к шине от самого себя и от следующего (в цепочке) модуля Доступ к шине предоставляется тому модулю, у которого приоритет выше

```
SUBDESIGN daisy
    (
      /local_request . INPUT;
      /local_grant   OUTPUT,
      /request_in . INPUT; % from lower priority %
/request_out  . OUTPUT; % to higher priority %
/grant_in   INPUT; % from higher priority % /grant_out
 OUTPUT, % to lower priority %
    )
```

```
  BEGIN
        DEFAULTS
   /local_grant   =   VCC;   %   active-low   output   %
 /request_out  =  VCC;  %  signals  should  default  %
 /grant_out = VCC; % to VCC %
        END DEFAULTS;
   IF /request_in == GND # /local_request == GND THEN
    /request_out = GND;
   END IF;
   IF /grant_in == GND THEN
  IF /local_request == GND THEN
   /local_grant = GND;
  ELSIF /request_in == GND THEN
   /grant_out = GND;
        END IF;
   END IF;
  END;
```

Все сигналы в данном файле имеют активный низкий уровень Фирма Altera рекомендует помечать имена сигналов с низким активным уровнем, например первым символом в имени ставить символ "/", который не является оператором, и использовать его постоянно

В операторе IF проверяется, является ли модуль активным, т е равен ли он GND Если модуль оказывается активным, реализуются действия, записанные в операторе IF

В объявлении по умолчанию DEFAULTS считается, что сигналу присваивается значение VCC, если он не активен

3.3.8. Реализация двунаправленных выводов

Система MAX+PLUS II позволяет конфигурировать порты ввода/вывода (I/O) в устройствах Altera как двунаправленные порты. Двунаправленный вывод задается как порт BIDIR, который подсоединяется к выходу примитива TRI Сигнал между этим выводом и буфером с тремя состояниями является двунаправленным, и его можно использовать в проекте для запуска других логических схем

Приводимый ниже файл **bus_reg2.tdf** реализует регистр, который производит выборку значения, найденного на шине с тремя состояниями, а также может передать обратно на шину хранимое значение

```
SUBDESIGN bus_reg2
    (
       clk · INPUT;
       oe . INPUT;
       io : BIDIR;
    )
BEGIN
    io = TRI(DFF(io, clk,, ), oe);
END,
```

Двунаправленный сигнал **io**, запускаемый примитивом **TRI**, используется в качестве входа **d** для D-триггера (**DFF**) Запятые в конце списка параметров отделяют места для сигналов триггера **clrn** и **prn** Эти сигналы по умолчанию установлены в неактивное состояние

Двунаправленный вывод можно также использовать для подсоединения TDF-ФАЙЛа более низкого уровня к выводу с высоким уровнем Прототип функции для TDF-ФАЙЛа более низкого уровня должен содержать двунаправленный вывод в предложении RETURNS. В приведенном ниже файле **bidir1.tdf** даны четыре примера использования макрофункции **bus_reg2**

```
FUNCTION bus_reg2 (clk, oe) RETURNS (io);
SUBDESIGN bidir1
   (
      clk, oe : INPUT;
      io[3..0] : BIDIR;
   )
BEGIN
   io0 = bus_reg2(clk, oe);
   io1 = bus_reg2(clk, oe);
   io2 = bus_reg2(clk, oe);
   io3 = bus_reg2(clk, oe);
END,
```

3.4. ПОСЛЕДОВАТЕЛЬНОСТНАЯ ЛОГИКА

Логическая схема называется последовательностной, если выходы в заданный момент времени являются функцией входов не только в тот же момент, но и во все предыдущие моменты времени Таким образом, в последовательностную схему должны входить некоторые элементы памяти (триггеры) В языке AHDL последовательностная логика реализована цифровыми автоматами с памятью (state machines), регистрами и триггерами При этом средства описания цифровых автоматов занимают особое место. Кроме того, к последовательностным логическим схемам относятся различные счетчики и контроллеры.

3.4.1. Объявление регистров

Регистры используются для хранения значений данных и промежуточных результатов счетчика, тактирование осуществляется синхросигналом Регистр создается его объявлением в секции VARIABLE

Для подсоединения примера примитива, макрофункции или цифрового автомата к другой логике в TDF-ФАЙЛ можно использовать порты Порт примера описывается в следующем формате <имя примера>.<имя порта>.

Имя порта — это вход или выход примитива, макрофункции или цифрового автомата, что является синонимом имени вывода в файлах проектов (GDF-файл), *.WDF и других.

Ниже приводится файл **bur_reg.tdf**, содержащий байтовый регистр, который фиксирует значения на входах **d** по выходам **q** на фронте синхроимпульса, когда уровень загрузки высокий

```
SUBDESIGN bur_reg
    (
       clk, load, d[7..0] : INPUT;
       q[7..0] : OUTPUT;
    )
VARIABLE
   ff[7..0] : DFFE;
BEGIN
   ff[].clk = clk;
   ff[].ena = load;
   ff[].d = d[];
   q[] = ff[].q;
END;
```

Как видно из файла, регистр объявлен в секции VARIABLE как D-триггер с разрешением (DFFE). В первом булевом уравнении в логической секции происходит соединение входа тактового сигнала

подпроекта к портам тактового сигнала триггеров ff[7 0] Во втором уравнении отпирающий тактовый сигнал соединяется с загрузкой В третьем уравнении входы данных подпроекта соединяются с портами данных триггеров ff[7 0]. В четвертом уравнении выходы подпроекта соединяются с выходами триггеров Все четыре уравнения выполняются одновременно

Можно также в секции VARIABLE объявить T-, JK- и SR-триггеры и затем использовать их в логической секции При использовании T-триггеров придется в третьем уравнении изменить порт **d** на **t**. При использовании JK- и SR-триггеров вместо третьего уравнения придется записать два уравнения, в которых происходит соединение портов **j** и **k** или **s** и **r** с сигналами

Примечание. При загрузке регистра по заданному фронту глобального тактового сигнала фирма Altera рекомендует использовать вход разрешения одного из регистров: DFFE, TFFE, JKFFE или SRFFE, чтобы контролировать загрузку регистра

3.4.2. Объявление регистровых выходов

Можно объявить регистровые выходы, если объявить выходы подпроекта как D-триггеры в секции VARIABLE

Приведенный ниже файл **reg_out.tdf** выполняет те же функции, что и предыдущий файл **bur_reg.tdf**, но имеет регистровые выходы

```
SUBDESIGN reg_out
    (
       clk, load, d[7  0] . INPUT,
        q[7 .0]   OUTPUT,
    )
    VARIABLE
                  q[7 .0] . DFFE,
    BEGIN
                  q[].clk = clk,
                  q[].ena = load,
                  q[] = d[],
    END,
```

При присвоении значения регистровому выходу в логической секции это значение формирует входные d сигналы регистров Выход регистра не изменяется до тех пор, пока не придет фронт синхросигнала Для определения тактирующего сигнала регистра нужно использовать описание в следующем формате <имя выходного вывода> clk для входа тактирующего сигнала регистра в логической секции Глобальный синхросигнал можно реализовать примитивом GLOBAL или выбором в диалоговом окне компилятора Logic Synthesis (логический синтез) опции **Automatic Global Clock**

Каждый отпирающий D-триггер-защелка, объявленный в секции VARIABLE, возбуждает выход с таким же именем, поэтому можно обращаться к **q** выходам объявленных триггеров без использования **q** порта этих триггеров

3.4.3. Создание счетчиков

Счетчиками называются последовательностные логические схемы для счета тактовых импульсов В некоторых счетчиках реализован счет вперед и назад (реверсивные счетчики), в некоторые счетчики можно загружать данные, а также обнулять их Счетчики обычно определяют как D-триггеры (DFF и DFFE) и используют операторы IF

Ниже приведен файл **ahdlcnt.tdf**, который реализует 16-битовый загружаемый счетчик со сбросом

```
SUBDESIGN ahdlcnt
    (
       clk, load, ena, clr, d[15..0] : INPUT;
 q[15..0] : OUTPUT;
 )
  VARIABLE
       count[15..0] . DFF,
  BEGIN
       count[].clk = clk,
       count[].clrn = !clr;
       IF load THEN
         . . . . . .
count[].d = d[];
       ELSIF ena THEN
  count[].d = count[] q + 1,
       ELSE
  count[].d = count[] q,
       END IF;
       q[] = count[];
  END,
```

В данном файле в секции VARIABLE объявлены 16 D-триггеров и им присвоены имена от **count0** до **count15.** В операторе IF определяется значение, загружаемое в триггеры по фронту синхросигнала (например, если сигнал загрузки имеет высокий уровень, то триггерам присваивается значение **d[ ]**)

3.5. ЦИФРОВЫЕ АВТОМАТЫ С ПАМЯТЬЮ (STATE MASHINE)

Цифровые автоматы так же, как таблицы истинности и булевы уравнения, легко реализуются в языке AHDL Язык структурирован, поэтому пользователь может либо сам назначить биты и значения состояний, либо предоставить эту работу компилятору системы MAX+PLUS

Компилятор, по уверениям производителя "использует патентованные перспективные эвристические алгоритмы", позволяющие сделать такие автоматические назначения состояний, которые минимизируют логические ресурсы, нужные для реализации цифрового автомата

Пользователю просто нужно нарисовать диаграмму состояний и построить таблицу состояний Затем компилятор выполняет автоматически следующие функции

- назначает биты, выбирая для каждого бита либо T-триггер, либо D-триггер,
- присваивает значения состояний,
- применяет сложные методы логического синтеза для получения уравнений возбуждения

По желанию пользователя можно задать в TDF-файле переходы в машине состояний с помощью объявления таблицы истинности

В языке AHDL для задания цифрового автомата нужно включить в TDF-файл следующие элементы

- объявление цифрового автомата (в секции VARIABLE),
- булевы уравнения управления (в логической секции),
- переходы между состояниями (в логической секции)

Цифровые автоматы в языке AHDL можно также экспортировать и импортировать, совершая обмен между файлами типа TDF и (GDF-файл) или * WDF; при этом входной или выходной сигнал задается как порт цифрового автомата в секции SUBDESIGN

3.5.1. Реализация цифровых автоматов (state machine)

Цифровой автомат задают в секции VARIABLE путем объявления имени цифрового автомата, его состояний и, возможно, выходных битов

Ниже приведен файл **simple.tdf**, который реализует функцию D-триггера

```
SUBDESIGN simple
        (
          clk    . INPUT;
          reset  : INPUT;
          d      : INPUT;
          q      : OUTPUT,
        )
    VARIABLE
       ss            · MACHINE WITH STATES (s0, s1);
    BEGIN
         ss clk = clk,
         ss reset = reset;
         CASE ss IS
              WHEN s0 =>
                   q = GND;
                   IF d THEN
                                  ss = s1;
                   END IF,
              WHEN s1 =>
                   q = VCC,
                   IF !d THEN
                                  ss = s0,
                   END IF;
          END CASE;
    END;
```

В данном файле в секции VARIABLE объявлен цифровой автомат (state machine) **ss** Состояния автомата определяются как **s0** и **s1** Биты состояний не определены

3.5.2. Установка сигналов Clock, Reset и Enable

Сигналы Clock, Reset и Enable управляют триггерами регистра состояний в цифровом автомате Эти сигналы задаются булевыми уравнениями управления в логической секции

В предыдущем примере (файл simple.tdf) синхросигнал цифрового автомата (Clock) формируется входом clk Асинхронный сигнал сброса цифрового автомата (Reset) формируется входом reset, имеющим высокий активный уровень Для подключения сигнала отпирания (Enable) нужно добавить в данный файл проекта строку "ena INPUT," в секцию SUBDESIGN, а также добавить в логическую секцию булево уравнение "ss ena = ena,"

3.5.3. Задание выходных значений для состояний

Для задания выходных значений можно использовать операторы IF и CASE В приведенном выше примере (файл **simple.tdf**) значение выхода q устанавливается равным GND, если цифровой автомат **ss** находится в состоянии **s0**, и равным VCC, когда она находится в состоянии **s1** Эти присваивания делаются в предложениях WHEN оператора CASE

Выходные значения можно также задавать в таблицах истинности, как будет описано в разделе "Присвоение битов и значений в машине состояний"

3.5.4. Задание переходов между состояниями

Переходы между состояниями определяют условия, при которых машина переходит в новое состояние Переходы в машине состояний задаются путем условного присвоения состояния в рамках одной конструкции, описывающей режим Для этой цели рекомендуется использовать оператор CASE или таблицу истинности

В приведенном выше примере (файл **simple.tdf**) переходы для каждого состояния определены в предложениях WHEN оператора CASE

3.5.5. Присвоение битов и значений в цифровом автомате

Бит состояния — это выход триггера, используемый для хранения одного бита значений цифрового автомата. В большинстве случаев для минимизации логических ресурсов следует предоставить компилятору системы MAX+PLUS II присвоение битов и значений состояния Однако пользователь может сделать это самостоятельно в объявлении цифрового автомата, если, например, он хочет, чтобы определенные биты были выходами цифрового автомата

Ниже приведен файл **stepper.tdf**, реализующий функцию контроллера шагового двигателя

```
SUBDESIGN stepper
 (
      clk, reset : INPUT;
   ccw, cw       : INPUT,
      phase[3..0] : OUTPUT;
 )
 VARIABLE
    ss          : MACHINE OF BITS (phase[3..0])
                          WITH STATES (
s0 = B"0001", s1 = B"0010", s2 = B"0100", s3 = B"1000"),
 BEGIN
    ss clk = clk;
    ss.reset = reset,
    TABLE
       ss,     ccw, cw   =>        ss;
       s0,     1,       x  =>                  s3;
       s0,     x,       1  =>                  s1,
       s1,     1,       x  =>                  s0;
       s1,     x,       1  =>                  s2;
       s2,     1,       x  =>                  s1;
       s2,     x,       1  =>                  s3;
       s3,     1,       x  =>                  s2,
       s3,     x,       1  =>                  s0;
    END TABLE;
 END,
```

В данном примере выходы **phase[3..0]**, объявленные в секции SUBDESIGN, объявляются так же, как биты цифрового автомата **ss** в объявлении цифрового автомата

3.5.6. Цифровые автоматы с синхронными выходами

Если выходы цифрового автомата зависят только от его состояния, их можно задать в предложении WITH STATES объявления цифрового автомата. Это сделает их менее подверженными ошибкам Кроме того, в некоторых случаях для логических операций потребуется меньше логических ячеек

Ниже приведен пример (файл moorel tdf), в котором реализован автомат Мура с четырьмя состояниями

```
SUBDESIGN moore1
(
 clk : INPUT;
 reset : INPUT;
  y : INPUT;
 z : OUTPUT;
)
VARIABLE        % current current %
    % state output %
 ss :MACHINE OF BITS (z)
                WITH STATES       (s0 = 0,
                                   s1 = 1,
                                   s2 = 1,
                                   s3 = 0);

 BEGIN
       ss.clk = clk;
       ss.reset = reset,
       TABLE
       % current current next %
       % state input state %
                ss,      y        =>       ss;
                s0,      0        =>       s0;
                s0,      1        =>       s2;
                s1,      0        =>       s0;
                s1,      1        =>       s2;
                s2,      0        =>       s2;
                s2,      1        =>       s3;
                s3,      0        =>       s3;
                s3,      1        =>       s1,
       END TABLE;
 END,
```

В данном примере состояния определены в объявлении цифрового автомата Переходы между состояниями определены в таблицв **next_state**, которая задана в объявлении таблицы истинности В данном примере машина имеет четыре состояния и только один бит состояния **z** Компилятор системы MAX+PLUS II автоматически добавляет еще один бит и делает соответствующие присвоения этой синтезированной переменной, для того чтобы получилась машина с четырьмя состояниями Такой цифровой автомат (state machine) требует, по крайней мере, двух битов

Если значения состояний используются как выходы (как в файле **moorel.tdf**), для проекта потребуется меньше логических ячеек, но, возможно, логические ячейки потребуют больше логики, чтобы возбудить входы триггера В этом случае модуль логического синтезатора компилятора, возможно, не сможет полностью минимизировать автомат

Другой способ построения цифрового автомата заключается в том, чтобы не делать присвоения состояний и явно объявить выходные триггеры.

Этот альтернативный метод использован в приведенном ниже файле **moore2.tdf**

```
SUBDESIGN moore2
        (
          clk . INPUT,
          reset : INPUT,
          y : INPUT;
          z · OUTPUT,
)
VARIABLE
ss . MACHINE WITH STATES (s0, s1, s2, s3)
zd . NODE,
    BEGIN
         ss.clk = clk;
         ss.reset = reset;
         z = DFF(zd, clk, VCC, VCC);
TABLE
%    current current    next          next %
%    state input        state         output %
     ss,   y    =>      ss,           zd;
     s0,   0    =>      s0,           0;
     s0,   1    =>      s2,           1;
     s1,   0    =>      s0,           0,
     s1,   1    =>      s2,           1;
     s2,   0    =>      s2,           1;
     s2,   1    =>      s3,           0;
     s3,   0    =>      s3,           0;
     s3,   1    =>      s1,           1,
END TABLE,
     END,
```

В данном примере, вместо того чтобы задать выходы присвоением значений состояниям в объявлении цифрового автомата, в объявление таблицы истинности добавлен один столбец под названием "**next output** (следующий выход)" В этом методе для синхронизации выходов используется D-триггер, вызов которого записан с помощью непосредственной ссылки

3.5.7. Цифровые автоматы с асинхронными выходами

В языке AHDL возможна реализация цифрового автомата с асинхронными выходами Выходы такого типа автоматов всегда изменяются, когда изменяются входы, независимо от состояния синхросигнала

В приведенном ниже фале **mealy.tdf** реализован автомат Мили с четырьмя состояниями и асинхронными выходами

```
SUBDESIGN mealy
    (
          clk   INPUT,
          reset   INPUT,
          y   INPUT,
          z   OUTPUT,
    )
 VARIABLE
       ss   MACHINE WITH STATES (s0, s1, s2, s3),
 BEGIN
          ss clk = clk,
          ss reset = reset,
          TABLE
 %   current current     current  next %
 %   state   input output state %
     ss,       y  =>     z,       ss,
     s0,       0  =>     0,       s0;
     s0,       1  =>     1,       s1;
     s1,       0  =>     1,       s1,
     s1,       1  =>     0,       s2;
     s2,       0  =>     0,       s2,
     s2,       1  =>     1,       s3;
     s3,       0  =>     0,       s3;
     s3,       1  =>     1,       s0;
        END TABLE,
 END,
```

3.5.8. Восстановление после неправильных состояний

Логика, сгенерированная для цифрового автомата компилятором системы MAX+PLUS II, будет работать так, как определено в TDF-файле Однако проекты с использованием цифровых автоматов часто допускают значения битов состояний, которые не присваиваются правильным состояниям Эти значения с не присвоенными битами состояний называются неправильными состояниями. Проект, который переходит в неправильное состояние, например в результате нарушений временных требований к установке или задержке, может реализовать ошибочные выходы Несмотря на рекомендации фирмы Altera по соблюдению временных требований к установке и задержке, пользователь может сделать восстановление цифрового автомата после неправильного состояния путем принудительного преобразования неправильного состояния к известному правильному в рамках оператора CASE

Для восстановления после неправильных состояний следует поименовать их все для данного автомата Предложение WHEN OTHERS в операторе CASE, которое принудительно преобразует состояния, применяется только к состояниям, которые были объявлены, а не упомянуты в предложении WHEN Данный метод работает, только если все неправильные состояния определены в объявлении цифрового автомата

Для n-битового цифрового автомата существует 2^n возможных состояний Поэтому нужно добавить воображаемые имена состояний, чтобы получилось такое их число

Ниже приведен файл **reoover.tdf**, в котором реализован цифровой автомат, который может восстанавливаться из неправильных состояний

```
SUBDESIGN recover
    (
       clk · INPUT;
       go . INPUT;
       ok . OUTPUT;
    )
 VARIABLE
sequence · MACHINE
OF BITS (q[2 .0]) WITH STATES (
idle,

                                        one,

                                        two,
three, four, illegal1, illegal2, illegal3);
 BEGIN
    sequence.clk = clk;
    CASE sequence IS
       WHEN idle =>
          IF go THEN
             sequence = one;
          END IF;
       WHEN one =>
          sequence = two,
       WHEN two =>
          sequence = three;
       WHEN three =>
          sequence = four,
       WHEN OTHERS =>
          sequence = idle,
       END CASE;
       ok = (sequence == four);
 END;
```

В данном примере цифровой автомат управляется тремя битами, поэтому он должен иметь 2^3, или 8 состояний. В объявлении заданы только 5 состояний. Следовательно, нужно туда добавить еще три воображаемых состояния **illegal1**, **illegal2**, **illegal3**.

3.6. РЕАЛИЗАЦИЯ ИЕРАРХИЧЕСКОГО ПРОЕКТА

В иерархической структуре проекта TDF-файлы, написанные на языке AHDL, можно использовать вместе с другими файлами проектов. На нижнем уровне проекта могут быть макрофункции, поставляемые фирмой Altera или разработанные пользователями

3.6.1. Использование макрофункций системы Altera MAX+PLUS II

В системе MAX+PLUS II есть большая библиотека, в которую входят 74 стандартные макрофункции, реализующие шины с последовательным опросом, функции оптимизации архитектуры и конкретные приложения. Библиотека представляет собой собрание блоков высокого уровня, используемых для создания проекта с иерархической логикой Во время инсталляции системы эти макрофункции автоматически записываются в каталог \maxplus2\max2lib и его подкаталоги, создаваемые в процессе инсталляции

В языке AHDL существуют два способа вызова (то есть вставки в качестве примера) макрофункции

- объявить переменную типа <macrofunction> в объявлении примеров INSTANCE в секции VARIABLE и использовать порты примера макрофункции в логической секции В этом способе важное значение имеют имена портов,
- использовать для макрофункции непосредственную ссылку в логической секции файла TDF В этом способе важен порядок портов

Входы и выходы макрофункций перечисляются в описании прототипов функций (FUNCTION PROTOTYPE). Прототипы функций можно записать в отдельный файл и указать его в своем файле с помощью директивы INCLUDE Такие Include-файлы создаются автоматически для данного проекта с помощью команды **Create Default Include File** Include-файл вставляется вместо вызывающей его директивы INCLUDE Для всех макрофункций системы MAX+PLUS II Include-файлы должны находиться в каталоге \maxplus2\max2inc

Ниже приведен файл macro1.tdf, который реализует 4-битовый счетчик, подсоединенный к дешифратору 4бит -> 16 бит. Соответствующие макрофункции вызываются объявлениями примеров в секции VARIABLE.

```
INCLUDE "4count";
INCLUDE "16dmux";

SUBDESIGN macro1
(
   clk        : INPUT;
   out[15..0] : OUTPUT;
)
VARIABLE
   counter : 4count;
   decoder : 16dmux;
```

```
BEGIN
    counter.clk = clk;
    counter.dnup = GND;
    decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
    out[15..0] = decoder.q[15..0];
END;
```

В данном файле используются директивы INCLUDE для импортирования прототипов функций для двух макрофункций фирмы Altera **4count** и **16dmux** В секции VARIABLE объявляются две переменные **counter** и **decoder** как примеры этих макрофункций В логической секции определяются входные порты для обеих макрофункций в формате <имя переменной-примера>.<имя порта> (Они ставятся в левой части булевых уравнений, а выходные порты — справа) Порядок портов в прототипе функции не важен, так как имена портов в логической секции перечисляются явно

Ниже приведен файл **mecro2.tdf**, выполняющий те же функции, что и предыдущий, но макрофункции в нем вызываются непосредственной ссылкой.

```
INCLUDE "4count";
INCLUDE "16dmux",

SUBDESIGN macro2
(
   clk          : INPUT;
   out[15..0]   : OUTPUT;
)
VARIABLE
   q[3..0]      : NODE;
BEGIN
   (q[3..0], ) = 4count (clk, , , , , GND, , , , ),
%

   out[15..0]  = 16dmux (.(d, c, b, a)=q[3..0]),
% equivalent in-line ref. with positional port association %
%     out[15..0]          =     16dmux     (q[3..0]);
%
END;
```

Вызов макрофункций **4oount** и **16dmux** осуществляется в логической секции непосредственной ссылкой (в правой части булевых уравнений)

Ниже приведены прототипы этих макрофункций, записанные в файлах **4count.inc 16dmux.inc:**

```
FUNCTION 4count (clk, clrn, setn, ldn, cin, dnup, d,
c, b, a)
        RETURNS (qd, qc, qb, qa, cout);
FUNCTION 16dmux (d, c, b, a)
        RETURNS (q[15 .0]);
```

Соединение портов показано в логической секции файла **maoro2.tdf** Порядок портов важен, так как должно быть однозначное соответствие между портами, описанными в прототипе функции и при ее реализации в логической секции В данном примере запятыми отделяются (но не перечисляются) порты, для которых не делается явного подключения

3.6.2. Создание и применение пользовательских макрофункций

В файлах, написанных на AHDL, можно легко создавать и использовать пользовательские макрофункции, выполнив следующие действия

- создать логику для макрофункции в файле проекта;
- определить порты макрофункции в объявлении прототипа функции

Прототип функции дает краткое описание функции: ее имя, а также входные, выходные и двунаправленные порты Можно также использовать машинные порты для макрофункций, которые импортируют или экспортируют функцию цифрового автомата.

Объявление прототипов функций может быть размещено в Include-файле, который вызывается в пользовательском файле. Используя команду Create Default Include File, можно автоматически создавать Include-файл с прототипом функции для любого файла проекта

- вставить в файл пример макрофункции с помощью объявления примера в секции VARIABLE или с помощью непосредственной ссылки в тексте,
- использовать макрофункцию в файле

3.6.3. Определение пользовательской макрофункции

Для использования макрофункции ее нужно либо включить в описание прототипа функции в TDF-файле, либо указать в директиве INCLUDE файла TDF имя Include-файла, содержащего прототип этой макрофункции Как уже упоминалось выше, Include-файлы можно создавать автоматически

Ниже приводится файл keyboard tdf, в котором реализован кодировщик для 16-клавишной клавиатуры

```
TITLE "Keyboard Encoder";
 INCLUDE "74151";
 INCLUDE "74154";
 INCLUDE "4count";
 FUNCTION debounce (clk, key_pressed)
       RETURNS (pulse);
 SUBDESIGN keyboard
    (
  clk . INPUT, % 50-KHz clock % col[3..0]   INPUT, %
keyboard columns % row[3. 0], d[3..0] : OUTPUT, % key-
board rows, key code % strobe · OUTPUT; % key code is
valid %
    )
  VARIABLE
  key_pressed   NODE; % VCC when key d[3  0] is
pressed % mux : 74151;
        decoder · 74154,
        counter : 4count;
        opencol[3..0] : TRI;
  BEGIN
  % Drive keyboard rows with a decoder and % % open-
collector outputs. %
           row[] = opencol[].out;
           opencol[].in = GND;
  opencol[] oe = decoder.(o0n,o1n,o2n,o3n);
decoder (b,a) = counter.(qd,qc);
         % Sense keyboard columns with a multiplexer %
           mux.d[3..0] = col[3  0],
           mux.(b,a) = counter.(qb,qa);
           key_pressed = !mux y;
  % Scan keyboard until a key is pressed. % % Drive
key's code onto d[] outputs %
           counter.clk = clk;
```

```
        counter.cin = !key_pressed;
        d[] = counter.(qd,qc,qb,qa),
% Generate strobe when key has settled %
        strobe = debounce(clk, key_pressed);
END;
```

В данном примере в директивах INCLUDE указываются файлы прототипов функции для стандартных макрофункций фирмы Altera **4count, 74151** и **74154**

Отдельно в файле дан прототип функции для макрофункции **debounce**, в котором описаны входы **cik** и **key_pressed** и выход **puise**

Примеры макрофункций **4count, 74151** и **74154** вызываются объявлением примеров в секции VARIABLE Пример макрофункции **debounce** вызывается непосредственной ссылкой в тексте логической секции

3.6.4. Импорт и экспорт цифровых автоматов (state machine)

Операции импорта и экспорта цифровых автоматов осуществляются между файлами TDF и другими файлами проекта путем задания входного или выходного порта как входа автомата (MACHINE INPUT) и его выхода (MACHINE OUTPUT) в секции SUBDESIGN Прототип функции, который представляет собой файл, содержащий машину состояний, должен указывать, какие входы и выходы принадлежат машине состояний Это осуществляется снабжением имен сигналов префиксом — ключевым словом Machine

Замечание. Порты типа MACHINE INPUT и MACHINE OUTPUT нельзя использовать в файле проекта верхнего уровня

Можно переименовать автомат, дав ему временное имя псевдоним) в объявлении MACHINE в секции VARIABLE Псевдоним цифрового автомата можно использовать в том файле, где создается цифровой автомат (state machine), или в том файле, где используется порт MACHINE INPUT для импортирования цифрового автомата Это имя можно потом использовать вместо первоначального имени цифрового автомата

Ниже приводится файл **ss_def.tdf**, который определяет и экспортирует цифровой автомат **ss** с портом **ss_out** типа MACHINE OUTPUT

```
SUBDESIGN ss_def
    (
      clk, reset, count   INPUT,
      ss_out   MACHINE OUTPUT,
    )
VARIABLE
      ss   MACHINE WITH STATES (s1, s2, s3, s4,
s5),
BEGIN
      ss_out = ss,
      CASE ss IS
           WHEN s1=>
IF count THEN ss = s2, ELSE ss = s1, END IF,
      WHEN s2=>
IF count THEN ss = s3, ELSE ss = s2; END IF,
      WHEN s3=>
IF count THEN ss = s4; ELSE ss = s3, END IF,
      WHEN s4=>
IF count THEN ss = s5; ELSE ss = s4, END IF;
      WHEN s5=>
IF count THEN ss = s1, ELSE ss = s5; END IF,
      END CASE;
      ss.(clk, reset) = (clk, reset);
END,
```

Ниже приводится файл **ss_use.tdf**, который импортирует цифровой автомат с портом **ss_in** типа MACHINE INPUT

```
SUBDESIGN ss_use
    (
      ss_in : MACHINE INPUT;
      out . OUTPUT;
    )
BEGIN
      out = (ss_in == s2) OR (ss_in == s4);
END;
```

Ниже приведен файл **top1.tdf**, в котором используются непосредственные ссылки в тексте для вставки примеров функций **ss_def** и **ss_use** В прототипах этих функций содержится ключевое слово MACHINE для указания, какие входы и выходы являются цифровыми автоматами с памятью (state machine)

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE
ss_out),
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);
DESIGN IS "top1" DEVICE IS "AUTO",
SUBDESIGN top1
    (
      sys_clk, /reset, hold   INPUT;
      sync_out   OUTPUT,
    )
VARIABLE
      ss_ref  MACHINE; % Machine Alias Declaration %
BEGIN
      ss_ref = ss_def(sys_clk, !/reset, !hold);
      sync_out = ss_use(ss_ref),
END,
```

Внешний цифровой автомат можно также реализовать в TDF-файле верхнего уровня с объявлением примера в секции VARIABLE

Ниже приведен файл **top2.tdf**, который имеет ту же функцию, что и файл **top1.tdf**, но для вызова макрофункций используется объявление примеров

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE
ss_out),
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);
DESIGN IS "top2' DEVICE IS "AUTO";
SUBDESIGN top2
      (
          sys_clk, /reset, hold : INPUT;
sync_out   OUTPUT,
      )
VARIABLE
sm_macro : ss_def;
sync : ss_use,
BEGIN
sm_macro.(clk, reset, count) = (sys_clk,
!/reset,!hold),
sync.ss_in = sm_macro.ss_out,
          sync_out = sync.out;
END,
```

3.7. УПРАВЛЕНИЕ СИНТЕЗОМ

3.7.1. Реализация примитивов LCELL и SOFT

Можно ограничить логический синтез с помощью замены переменных типа узел (NODE) примитивами SOFT и LCELL Переменные NODE и примитивы LCELL обеспечивают наилучшее управление логическим синтезом Примитивы SOFT обеспечивают более слабое управление логическим синтезом

Переменные NODE, которые объявляются в секции VARIABLE, не накладывают больших ограничений на логический синтез Во время синтеза модуль логического синтеза компилятора системы MAX+PLUS II заменяет каждый пример использования переменной NODE логикой, которую она представляет. Затем происходит минимизация логики до одной логической ячейки Этот метод обычно приводит к ускорению работы схемы, но в результате может получиться слишком сложная логика или же ее трудно свести к одной ячейке

Буферы SOFT обеспечивают лучшее управление использованием ресурсов, чем переменные NODE Модуль логического синтезатора выбирает, когда заменить примеры использования примитивов SOFT примитивами LCELL

Буферы SOFT могут помочь уничтожить логику, которая оказалась слишком сложной, и сделать проект проще, однако при этом может быть увеличено число логических операций и скорость выполнения программы соответственно уменьшится

Наиболее сильное управление процессом логического синтеза обеспечивается примитивами LCELL Модуль логического синтезатора минимизирует всю логику, которая запускает примитив LCELL, таким образом, чтобы можно было свести ее к одной логической ячейке Примитивы LCELL реализуются в виде одной логической ячейки (их нельзя убрать из проекта, даже если они имеют единственный вход). Если проект минимизирован до такой степени, что один примитив LCELL имеет единственный вход, в этом случае вместо примитивов LCELL можно использовать примитивы SOFT, которые убираются в процессе логического синтеза

Примечание При многоуровневом синтезе компилятор системы MAX+PLUS II автоматически помещает буферы SOFT в более выгодное место проекта, если включить опцию SOFT Buffer Insertion logic.

Ниже приводятся две версии файла TDF — с переменными NODE и с примитивами SOFT В версии **nodevar** переменная **odd_parity** объявлена как NODE, затем ей присваивается булево выражение d0 $ d1 $... $ d8 В версии **softbuf** компилятор заменит некоторые примитивы SOFT примитивами LCELL во время обработки данных для лучшего использования ресурсов устройства

```
TDF with NODE TDF
Variables.
SUBDESIGN nodevar
(
:
)
VARIABLE
odd_parity : NODE;
BEGIN
odd_parity =
d0 $ d1 $ d2
$ d3 $ d4 $ d5
$ d6 $ d7 $ d8;
END;
```

```
TDF with SOFT Primitives.
SUBDESIGN softbuf
(
:
)
VARIABLE
odd_parity : NODE;
BEGIN
odd_parity =
SOFT(d0 $ d1 $ d2)
$ SOFT(d3 $ d4 $ d5)
$ SOFT(d6 $ d7 $ d8);
END;
```

3.7.2. Значения констант по умолчанию

Логический синтезатор автоматически выполняет подключение к GND всех выходов таблицы истинности, если не удовлетворяется ни одно из условий входа таблицы Для присвоения выходам таблицы истинности значения VCC можно использовать одно или несколько объявлений языка AHDL по умолчанию С помощью этих объявлений можно задать значения по умолчанию для соответствующих выходов

Например, если большинство выходов таблицы истинности равны 1, можно задать значение по умолчанию VCC.

Примечание. Не следует путать значения по умолчанию для переменных и портов которые присваиваются в секции SUBDESIGN

3.7.3. Присвоение битов и значений в цифровом автомате

Логический синтезатор автоматически минимизирует число битов состояния, требуемое для цифрового автомата При этом оптимизируется как использование устройства, так и характеристики его работы Однако некоторые цифровые автоматы могут работать быстрее, при значениях состояния, использующих число битов, большее минимального Для контроля этих случаев пользователь сам объявляет биты и значения для цифрового автомата

3.8. ЭЛЕМЕНТЫ ЯЗЫКА AHDL

3.8.1. Зарезервированные ключевые слова

Зарезервированные ключевые слова используются для следующих целей:

- для обозначения начала, конца и переходов в объявлениях языка AHDL,
- для обозначения предопределенных констант, т е GND и VCC

Ключевые слова можно использовать как символические имена только если они заключены в одинарных кавычках (') Их можно также использовать в комментариях

Для того чтобы получить контекстовую помощь по ключевому слову, убедитесь, что ваш файл сохранен с расширением **tdf**, затем нажмите одновременно две кнопки **Shift+F1** в окне текстового редактора Text Editor и щелкните кнопкой мыши Button 1 на ключевом слове

Altera рекомендует все ключевые слова набирать прописными буквами.

Список всех зарезервированных ключевых слов языка AHDL

FUNCTION	OTHERS	
CASE	TABLE	JKFFE
BITS	SRFFE	NCLUDE
DFF	VCC	NODE
DFFE	WHEN	NOR
ELSE	WITH	NOT
END	XNOR	OPTIONS
EXP	XOR	OR
AND	GLOBAL	OUTPUT
BEGIN	GND	RETURNS
BURIED	INPUT	SOFT
BIDIR	IF	SRFF
CARRY	IS	STATES
CASCADE	JKFF	SUBDESIGN
CLIQUE	LATCH	TFF
CONNECTED_PINS	LCELL	TFFE
CONSTANT	MACHINE	THEN
DEFAULTS	MACRO	TITLE
DESIGN	MCELL	TRI
DEVICE	NAND	VARIABLE
ELSIF	OF	X

3.8.2. Символы

Ниже приведены символы, имеющие определенное значение в языке AHDL В этот перечень не включены символы, используемые в булевых выражениях как операторы и для операций сравнения

Символ	Функция
_ (подчеркивание)	Используемые пользователем идентификаторы
- (тире)	Символы в символических именах
— (два тире)	Начинает комментарий в стиле VHDL, который продолжается до конца строки
% (процент)	Заключает с двух сторон комментарий стиля AHDL
( ) (круглые скобки)	Заключают и определяют последовательные имена групп. Заключают имена выводов в секции подпроекта (Subdesign Section) и в прототипах функций. Заключают (необязательно) входы и выходы таблиц в объявлении Truth Table. Заключают состояния в объявлении цифрового автомата State Machine. Заключают более приоритетные операции в булевых выражениях Заключают необязательные варианты в секции проекта Design Section (внутри объявления назначения ресурсов Assignment)
[ ] (квадратные скобки)	Заключают диапазон значений в десятичном имени группы
'...' (одинарные кавычки)	Заключают символические имена
" " (двойные кавычки)	Заключают строку в объявлении названия Title. Заключают цифры в не десятичных номерах Заключают путь в объявлении Include Могут (необязательно) заключать имя проекта и устройства в секции проекта Design Section Могут (необязательно) заключать имя в объявлении назначения клики графа Clique Assignment
. (точка)	Отделяет символические имена переменных в макрофункции или примитиве от имен портов. Отделяет имя файла от расширения
... (многоточие)	Разделяет наименьшее и наибольшее значение в диапазонах
; (точка с запятой)	Заканчивает объявления и секции в языке AHDL
, (запятая)	Разделяет элементы последовательных групп и списков
: (двоеточие)	Отделяет символические имена от типов в объявлениях и назначениях ресурсов
@"собака"	Присваивает символические узлы выводам устройства и логическим ячейкам в объявлениях назначения ресурсов Resource Assignment
= (равенство)	Присваивает значения по умолчанию GND и VCC входам в секции подпроекта Subdesign. Присваивает установочные значения в вариантах Присваивает значения состояниям в машине состояний Присваивает значения в булевых уравнениях
=> (стрелка)	Отделяет входы от выходов в объявлениях таблицы истинности Truth Table. Отделяет предложения с WHEN от булевых выражений в операторе Case

3.8.3. Имена в кавычках и без кавычек

В языке AHDL есть три типа имен

- *Символические имена* — это определяемые пользователем идентификаторы Они используются для обозначения следующих частей TDF
  - — внутренних и внешних узлов (вершин),
  - — констант,
  - — переменных цифрового автомата, битов состояний, имен состояний,
  - — примеров (Instance)
- *Имена подпроекта* — это определяемые пользователем имена для файлов проекта более низкого уровня Имя подпроекта должно быть таким же, как имя файла TDF.
- *Имена портов* — это символические имена, идентифицирующие вход или выход примитива или макрофункции.

В файле **Fit** вашего проекта могут появиться генерируемые компилятором имена выводов, с символом "тильда" (~). Этот символ зарезервирован для имен, генерируемых компилятором, пользователю запрещается его использовать для обозначения имен выводов, узлов (вершин), групп (шин)

Существуют две формы записи для всех трех типов имен (символических, подпроекта и портов). в *кавычках* (') и *без кавычек*

Если разработчик создает символ по умолчанию для файла TDF, который включает в себя имена портов в кавычках, собственно кавычки не входят в имена выводов

Ниже в таблице указаны все возможные варианты записи имен в языке AHDL·

Разрешенные символы	Имя подпроекта		Симаолическое имя		Имя порта	
	Без кавычек	В кавычках	Без кавычек	В кавычках	Без кавычек	В кавычках
A-Z	+	+	+	+	+	+
a-z	+	+	+	+	+	+
0-9	+	+	+	+	+	+
Подчеркивание (_)	+	+	+	+	+	+
Косая черта (/)	—	—	+	+	+	+
Тире (—)	—	+	—	+	-	+
Только цифры (0-9)	+	+	—	+	+	+
Ключевое слово	—	+	—	+	—	+
Максимальное число символов	8	8	32	32	32	32

Примеры

Символические имена без кавычек

a, /a . разрешенные

-foo, node, 55 неразрешенные

Символические имена в кавычках

'-bar', 'table', '1221' разрешенные

'bowling4$', 'has a space ',

'a_name_with_more_than_32_characters' неразрешенные

3.8.4. Группы

Символические имена и порты одного и того же типа можно объявить и использовать как группы в булевых выражениях и уравнениях Группа, в которую может входить до 256 элементов (или "битов"), обрабатывается как коллекция узлов (вершин) и считается единым целым

Группа в логической секции файла TDF может состоять из узлов или чисел Одиночные узлы и константы GND и VCC могут дублироваться для образования групп в булевых выражениях и уравнениях

В файле **Fit** могут появиться генерируемые компилятором имена выводов, с символом "тильда" (~) Этот символ зарезервирован для имен, генерируемых компилятором, пользователю запрещается его использовать для обозначения имен выводов, узлов (вершин), групп (шин).

3 8 4 1 ФОРМА ЗАПИСИ ГРУПП

Существует две формы записи при объявлении групп

1 *Десятичное имя группы* состоит из символического имени (или имени порта), за которым следует диапазон десятичных чисел, заключенный в квадратные скобки, например a[4. 1] После этого имени группы указывается только один диапазон Всего сим-

волическое имя (или имя порта) вместе с самым длинным (по написанию) номером в диапазоне может содержать до 32 символов

Если группа определена, квадратные скобки обеспечивают самый краткий путь задания всего диапазона Вместо диапазона можно также указывать только одно десятичное число, например a[5] Однако такая форма записи означает единственное символическое имя, а не имя группы, и эквивалентно имени a5

2. *Последовательное имя группы* состоит из заключенного в скобки перечня символических имен, имен портов или чисел, разделенных запятыми, например (a, b, c). В этом перечне могут быть также десятичные имена групп, например (a, b, c[5 .1]). Такая запись используется для задания имен портов.

Например.

reg. (d, clk, clrn, prn)

В следующем примере показаны варианты записи одной и той же группы

b[5. 0]

(b5, b4, b3, b2, b1, b0)

b[]

3 8 4 2 ДИАПАЗОН И ПОДДИАПАЗОН ГРУПП

Диапазоны в десятичных именах групп обозначаются десятичными номерами, обычно перечисляемыми в порядке убывания Для того чтобы указать диапазон в порядке возрастания или в обоих порядках, разработчик должны установить опцию **BITO** в объявлении опций **Options**, тогда компилятор не будет выдавать предупреждающие сообщения

Поддиапазоны включают подмножество узлов, заданных в объявлении группы, их можно указывать различными способами Например, если объявили группу c[5 1], Вы можете использовать следующие поддиапазоны этой группы c[3 1], c4, c[5], (c2, c4)

В левой части булевых уравнений или в ссылках (reference) по тексту программы (in-line) на макрофункции или примитивы в написании имени группы можно использовать запятые вместо перечисления имен.

При указании диапазона вместо чисел можно использовать константы. Например, **q[MAX..0]** является разрешенной формой записи, если константа **MAX** определена в объявлении **Constant**

3.8.5. Числа в языке AHDL

В языке AHDL можно использовать десятичные, двоичные, восьмеричные и шестнадцатеричные числа в любой комбинации В таблице приведен синтаксис записи чисел в языке AHDL для каждой системы счисления.

Система счисления	Значения
Десятичная	<последовательность цифр 0–9>
Двоичная	B"<последовательность из 0, 1, X>", где символ X обозначает безразличное значение
Восьмеричная	O"< последовательность цифр 0–7>" или Q"< последовательность цифр 0–7>"
Шестнадцатеричная	X"< последовательность цифр 0–9, букв A–F>" или H"< последовательность цифр 0–9, букв A–F>"

Примеры разрешенной записи чисел в языке AHDL:

B"0110X1X10"

Q"4671223"

H"123AECF"

К числам в языке AHDL применяются следующие правила

- компилятор системы MAX+PLUS II всегда интерпретирует числа как группы двоичных цифр,
- булевых выражениях числа нельзя присваивать одиночным узлам (вершинам). Вместо этого нужно использовать константы **VCC** и **GND**.

3.8.6. Булевы выражения

Булевы выражения состоят из операндов, разделенных логическими и арифметическими операторами и компараторами и (необязательно) сгруппированных с помощью круглых скобок. Выражения используются в булевых уравнениях, а также в других конструкциях языка, таких, как операторы **Case** и **if**

Существуют следующие применения булевых выражений

- операнд Пример `a, b[5..1], 7, VCC`
- встроенная в текст (in-line) ссылка (reference) на примитив или макрофункцию
- префиксный оператор (! или -), примененный к булеву выражению Пример. `!c`
- два булевых выражения, разделенные двоичным (не префиксным) оператором Пример `d1 $ d3`
- заключенное в круглые скобки булево выражение Пример: `(!foo & bar)`

Результат каждого булева выражения должен иметь ту же ширину, что и узел или группа (в левой стороне уравнения), которому он, в конечном счете, присваивается

3.8.7. Логические операторы

В таблице приведены логические операторы для булевых выражений

Оператор:	Пример:	Описание:
!	!tob	Дополнение (префиксное обращение)
NOT	NOT tob	
&	bread & butter	Логическое И
AND	bread AND butter	
!&	a[3..1] !& b[5..3]	Обращение логического И
NAND	a[3 1] NAND b[5..3]	
#	trick # treat	Логическое ИЛИ
OR	trick OR treat	
!#	c[8..5] !# d[7..4]	Обращение логического ИЛИ
NOR	c[8..5] NOR d[7 4]	
$	foo $ bar	Исключающее ИЛИ
XOR	foo XOR bar	
!$	x2 !$ x4	Обращение исключающего
XNOR	x2 XNOR x4	логического ИЛИ

Каждый оператор представляет собой логический вентиль с двумя входами, исключение составляет оператор NOT, являющийся префиксным инвертором Для записи логического оператора можно использовать его имя или символ

Выражения, в которых используются эти операторы, интерпретируются по-разному в зависимости от того, что представляют собой операнды одиночные узлы (вершины), группы или числа Кроме того, выражения с оператором NOT интерпретируются не так как другие логические операторы

3.8.8. Выражения с оператором NOT

С оператором NOT можно использовать три типа операндов

- если операнд представляет собой одиночный узел, константы GND или VCC, выполняется одна операция обращения Пример `!a`

- если операнд представляет собой группу узлов, каждый элемент группы проходит через инвертор. Пример !a[4 .1] интерпретируется как (!a4, !a3, !a2, !a1),
- если операнд представляет собой число, он обрабатывается как двоичное число, т е как группа соответствующего числа битов, где обращается каждый бит Пример !9 операнд интерпретируется как двоичное число !B"1001" (группа из четырех элементов), результат B"0110"

3.8.9. Выражения с операторами AND, NAND, OR, NOR, XOR и XNOR

Существует четыре комбинации операндов с двоичными (не префиксными) операторами, и каждая из них интерпретируется по-особому

- если операнд представляет собой одиночный узел, константы GND или VCC, оператор выполняет логическую операцию над двумя элементами Пример (a&b),
- если оба оператора являются группами узлов, оператор применяется к соответствующим узлам каждой группы, производя ряд операций на битовом уровне между группами Группы должны быть одинакового размера Пример (a,b) # (c,d) интерпретируется как (a#c, b#d),
- если один оператор представляет собой одиночный узел, константы GND или VCC, а другой операнд – группу узлов, то одиночный узел или константа дублируются для образования группы такого же размера, что и второй операнд Затем выражение интерпретируется как групповая операция Пример a & b[2 1] интерпретируется как (a&b2, a&b1),
- если оба операнда представляют собой числа, более короткое (в смысле числа битов в двоичном представлении) число дополняется незначащими нулями, чтобы сравняться по числу битов с другим операндом Затем выражение интерпретируется как групповая операция Пример: в выражении (3#B) 3 и 8 преобразуются в двоичные числа B"0011" и B"1000" Результатом является B"1011",
- если один операнд представляет собой число, а другой – узел или группу узлов, то число усекается или расширяется до размера группы При усечении значащих битов генерируется сообщение об ошибке Выражение затем интерпретируется как групповая операция Пример (a,b,c)&1 интерпретируется как (a&0, b&0 c&1)

Выражение с константой VCC интерпретируется не так как выражение с операндом 1 В первом выражении приведенного ниже примера число 1 расширяется по числу битов двоичного представления, чтобы соответствовать размеру группы Во втором уравнении узел VCC дублируется для образования группы того же размера Затем каждое уравнение интерпретируется как групповая операция

```
' operation
      (a, b, c) & 1 = (0, 0, c)
      (a, b, c) & VCC = (a, b, c).
```

3.8.10. Арифметические операторы

Арифметические операторы используются для выполнения арифметических операций сложения и вычитания над группами и числами В языке AHDL применяются следующие арифметические операторы

Оператор	Пример	Описание
+ (унарный)	+1	Положительное значение
- (унарный)	-a[4 1]	Отрицательное значение
+	count[7 0] + delta[7 0]	Сложение
-	nghtmost_x[] - leftmost_x[]	Вычитание

Унарные плюс (+) и минус (–) являются префиксными операторами Оператор + не влияет на свой операнд, и разработчик может использовать его для иллюстративных целей (т. е. для указания положительного числа) Оператор – интерпретирует свой операнд в двоичном представлении (если он таковым не является с самого начала) Затем он выполняет операцию унарного минуса, т е получает дополнение операнда как двоичного числа

К другим арифметическим операторам применяются следующие правила.

- операции выполняются между двумя операндами, которые должны быть группами узлов или числами,
- если оба операнда представляют собой группы узлов, то они должны иметь одинаковый размер,
- если оба операнда представляют собой числа, то более короткое (в двоичном представлении) число расширяется (дополняется незначащими нулями), чтобы сравняться по числу битов с другим операндом,
- если один операнд представляет собой число, а другой является группой узлов, то число усекается или расширяется в двоичном представлении до размера группы. Если при этом усекаются значащие биты, компилятор системы MAX+PLUS II генерирует сообщение об ошибке.

3.8.11. Компараторы (Операторы сравнения)

Компараторы используются для сравнения одиночных узлов или групп Существует два типа компараторов логические и арифметические В языке AHDL реализованы следующие компараторы

Компаратор	Пример.	Описание:
== (логический)	addr[19..4] == B"B800"	Равны ?
!= (логический)	b1 != b3	Не равны ?
< (арифметический)	fame[] < power[]	Меньше чем ?
<= (арифметический)	money[] <= power[]	Меньше либо равно ?
> (арифметический с)	love[] > money[]	Больше чем ?
>= (арифметический)	delta[] >= 0	Больше либо равно ?

Оператор (==) используется исключительно в булевых выражениях

Логические компараторы могут сравнивать одиночные узлы, группы узлов и числа При сравнении групп узлов или чисел размер групп должен быть одинаковым Компилятор системы MAX+PLUS II выполняет побитовое сравнение групп и возвращает значение VCC, если результат — истина, и GND, если результат — ложь

Арифметические компараторы могут сравнивать только группы узлов и числа, и группы должны иметь одинаковый размер Компилятор выполняет над группами операции беззнакового сравнения значений, т е каждая группа интерпретируется как положительное двоичное число и сравнивается с другой группой

3.8.12. Приоритеты в булевых уравнениях

Операнды, разделенные логическими и арифметическими операторами и компараторами, оцениваются по правилам приоритетов, перечисленным ниже (приоритет 1 является наивысшим) Равноприоритетные операции выполняются по очереди, слева направо. Порядок выполнения может быть изменен с помощью заключения в круглые скобки

Приоритет	Оператор	Компаратор
1	-	(отрицание)
1	!	(логическое НЕ)
2	+	(сложение)
2	-	(вычитание)
3	==	(равно?)
3	!=	(не равно?)
3	<	(меньше чем)
3	<=	(меньше либо равно)
3	>	(больше чем)
3	>=	(больше либо равно)
4	&	(AND)
4	!&	(NAND)
5	$	(XOR)
5	!$	(XNOR)
6	#	(OR)
6	!#	(NOR)

3.8.13. Примитивы

В системе MAX+PLUS II есть разнообразные примитивные функции для проектирования электрических схем Поскольку логические операторы, порты и некоторые объявления языка AHDL заменяют примитивы в файлах TDF, написанных на языке AHDL, примитивы языка AHDL являются подмножеством примитивов, которые имеются для файлов графического проекта (GDF-файл), как будет показано ниже

В файлах TDF нет необходимости использования прототипов функций языка AHDL для примитивов Однако путем включения в ваш TDF-файл прототипа функции разработчик может переопределить порядок вызова входов примитива

3 8 13 1 ПРИМИТИВЫ БУФЕРОВ

```
CARRY   LCELL (MCELL)
CASCADE SOFT
EXP     TRI
GLOBAL (SCLK)
```

Примитив **CARRY** (перенос)

```
Прототип функции: FUNCTION CARRY (in)
    RETURNS (out);
```

Примитив CARRY определяет логику "переноса на выход" (carry-out) для одной функции и действует как "перенос на вход" (carry-in) для другой функции Функция переноса (carry) осуществляет логику быстрого переноса по цепочке для таких функций, как сумматоры и счетчики Примитив CARRY поддерживается только для устройств семейства FLEX 8000 Для других устройств он игнорируется

Если разработчик использует примитив CARRY некорректно, он игнорируется и компилятор выдает соответствующее предупреждающее сообщение

Вы можете позволить компилятору автоматически вставлять и убирать CARRY во время синтеза логики, для этого нужно выбрать опцию логики с переносом по цепочке **Carry Chain** или стиль синтеза логики, который включает эту опцию

Примитив **CASCADE** (каскад)

Прототип функции `FUNCTION CASCADE (in) RETURNS (out);`

Буфер CASCADE определяет функцию каскадного выхода из логической схемы И (AND) или ИЛИ (OR) и действует как входной каскад для другой логической схемы И или ИЛИ. Функция входного каскада позволяет организовать каскадный процесс, который представляет собой быстрый выход, расположенный на каждой комбинационной логической ячейке и участвующий в операции ИЛИ или И с выходом соседней комбинационной логической ячейки в устройстве С помощью примитива CASCADE логическая ячейка И или ИЛИ, которая управляет примитив CASCADE и логическая ячейка И или ИЛИ, которая управляется примитивом CASCADE, помещаются в одном устройстве, причем первый символ после проведения над ним логической операции И или ИЛИ переходит во второй символ Примитив CASCADE поддерживается только для устройств семейства FLEX 8000 Для других устройств он игнорируется

При использовании примитива CASCADE, нужно соблюдать следующие правила

- примитив CASCADE может формировать сигнал или управляться одной единственной схемой И или ИЛИ,
- инверсии логической схемы ИЛИ (OR) рассматривается как схема И (AND) и наоборот Логическими эквивалентами схем И (AND) являются BAND, BNAND и NOR Логическими эквивалентами схем ИЛИ (OR) являются BOR, BNOR, and NAND,
- два примитива CASCADE не могут формировать сигнал для одной и той же логической схемы,
- примитив CASCADE не может формировать сигнал для логической схемы XOR,
- примитив CASCADE не может формировать сигнал для примитива вывода OUTPUT или OUTPUTC или регистра,
- в соответствии с правилами Де Моргана (De Morgan) об инверсии для каскадированных логических схем И или ИЛИ нужно, чтобы все примитивы в каскадированной цепочке были одного и того же типа Каскадированная логическая схема И (AND) не может формировать сигнал для каскадированной схемы ИЛИ (OR) и наоборот

Если разработчик использует примитив CASCADE некорректно, он игнорируется и компилятор выдает соответствующее предупреждающее сообщение

Вы можете позволить компилятору автоматически вставлять и убирать примитивы CASCADE во время синтеза логики, для этого нужно выбрать опцию логики с переносом по цепочке **Cascade Chain** или стиль синтеза логики, который включает эту опцию

Примитив **EXP** (расширитель)

Прототип функции `FUNCTION EXP (in) RETURNS (out);`

Расширительный буфер EXP определяет необходимость дополнительного места при вычислении произведения в проекте Произведение из расширителя в устройстве инвертируется

Примитив EXP поддерживается только семейством устройств MAX5000/EPS464 и MAX 7000 В других устройствах он рассматривается как логическая схема НЕ (NOT) Для каждого отдельного устройства нужно ознакомиться с его техническим паспортом, чтобы

узнать, как конкретное устройство использует логические ячейки и расширительные буферы для вычисления произведений.

Будет или нет использоваться расширитель для произведения, зависит от полярности логики, требуемой вычисляемыми функциями Например, если буфер EXP питает две логические схемы И (AND) (т е логическое умножение), а вторая схема И имеет инвертированный вход, то примитив EXP, питающий инвертированный вход, во время логического синтеза убирается, создавая, таким образом, положительную логику Примитив EXP, питающий неинвертированный вход, не убирается, и для реализации логики используется расширитель для произведения Обычно логический синтезатор определяет, куда вставлять и откуда убирать буферы Фирма Altera рекомендует, чтобы только опытные разработчики использовали примитив EXP в своих проектах

В устройствах, содержащих много логических блоков (LAB), выход буфера EXP может формировать сигнал только в пределах одного LAB Примитив EXP дублируется для каждого LAB, где он требуется Если проект содержит много расширительных буферов, логический синтезатор может преобразовать их в буферы LCELL, чтобы сбалансировать использование расширителя для произведения и логической ячейки

Не используйте примитивы EXP для создания намеренной задержки или асинхронного импульса Задержка этих элементов изменяется с температурой, напряжением питания и при процессах изготовления устройства, поэтому может возникнуть режим состязания и получится ненадежная схема

Примитив **GLOBAL** (глобальный)

Прототип функции `FUNCTION GLOBAL (in)`
`RETURNS (out),`

Буфер GLOBAL указывает, что сигнал (signal) должен использовать глобальный синхронный сигнал Clock, сигналы Clear, Preset или Output Enable вместо сигналов, сгенерированных внутренней логикой или поступающих через обычные выводы вход/выход (I/O) Глобальные сигналы используются в различных семействах устройств в соответствии с их синхронизацией

Если входной вывод формирует сигнал непосредственно на вход примитива GLOBAL, то выход примитива GLOBAL может быть использован, чтобы формировать сигнал на входы Clock, Clear, Preset или Output Enable к примитиву Должно существовать непосредственное соединение от выхода GLOBAL ко входу регистра или буфера TRI Если буфер GLOBAL формирует сигнал Output Enable буфера TRI, может потребоваться логическая схема НЕ (NOT)

Один единственный вход может проходить через GLOBAL при формировании сигнала для регистров и сигналов Clock, Clear или Preset или вход буфера TRI для сигнала Output Enable

Глобальные синхросигналы распространяются быстрее, чем асинхронные (array) сигналы, и могут освобождать ресурсы устройства для реализации другой логики Для осуществления синхронизации части или всего проекта следует использовать примитив GLOBAL Для того чтобы проверить, имеют ли регистры глобальную синхронизацию, нужно просмотреть файл отчета Report File обрабатываемого проекта

Если ваш проект устройства MAX 5000 сочетает асинхронный режим работы и режим с глобальной синхронизацией, а модуль компилятора Fitter не может выбрать подходящий, проблема может быть решена удалением буфера GLOBAL Если аналогичная проблема встретится вам в проекте MAX 7000, замените асинхронный режим режимом с глобальной синхронизацией В качестве альтернативы использованию примитива GLOBAL разработчик может предоставить компилятору возможность автоматически выбирать, каким будет существующий сигнал проекта, глобальным Clock, Clear, Preset или Output Enable, посредством команды **Logic Synthesis**.

Примитив **LCELL**

Прототип функции `FUNCTION LCELL (in)`
`RETURNS (out);`

Буфер LCELL выделяется для логической ячейки проекта. Буфер LCELL вырабатывает истинное значение и дополнение логической функции и делает их доступными для всей логики устройства

> Для обратной совместимости с более ранними версиями системы MAX+PLUS II имеется буфер MCELL, функционально такой же, как и буфер LCELL В новых проектах следует использовать только буфер LCELL

Буфер LCELL всегда занимает одну логическую ячейку Он не удаляется из проекта во время логического синтеза

> Не используйте примитивы LCELL для создания задержки или асинхронного импульса Задержка этих элементов изменяется с температурой, напряжением питания и в процессе изготовления изделия, поэтому могут возникнуть режимы состязаний

С помощью команды компилятора **Automatic LCELL insertion разработчик** может дать указание компилятору автоматически вставлять буферы LCELL в проект, если это необходимо для трассировки проекта в том случае, когда назначения определенных пользователем ресурсов и имеющиеся у устройства, не обеспечивают возможности внесения изменений в проект

Примитив **SOFT**

Прототип функции `FUNCTION SOFT (in)`
`RETURNS (out),`

Буфер SOFT устанавливает, что логическая ячейка может понадобиться в проекте При обработке проекта процессором логический синтезатор исследует логику, управляющую примитивом и определяет, нужна ли логическая ячейка Если нужна, то буфер SOFT преобразуется в LCELL, в противном случае, SOFT удаляется

Если компилятор показывает, что проект слишком сложный, разработчик можете редактировать проект, вставляя буферы SOFT для предотвращения расширения логики Например, разработчик может добавить буфер SOFT в комбинаторный выход макрофункции для резвязки двух комбинаторных цепей Разработчик может также включить логическую опцию **SOFT Buffer insertion** в стили логического синтеза по умолчанию в проект, чтобы компилятор вставлял буферы SOFT автоматически

Примитив **TRI**

Прототип функции `FUNCTION TRI (in, oe)`
`RETURNS (out);`

Примитив TRI представляет собой трехстабильный буфер с сигналами входным, выходным и Output Enable (Отпирание выхода) Если уровень сигнала Output Enable высокий, выход будет управляться входом При низком уровне сигнала Output Enable выход переходит в состояние высокого импеданса, что позволяет использовать I/O как входной вывод По умолчанию сигнал Output Enable устанавливается в значение VCC

Если сигнал Output Enable буфера TRI подсоединяется к VCC или логической функции, которая будет минимизировать до тождест-

венной единицы, буфер TRI может быть преобразован в буфер SOFT во время логического синтеза

При использовании буфера TRI нужно соблюдать следующие правила

- буфер TRI может управлять только одним выводом BIDIR или BIDIRC Разработчики должны использовать вывод BIDIR или BIDIRC, если после буфера TRI применяется обратная связь Если буфер TRI формирует сигнал на логику, он должен также формировать сигнал для BIDIR или BIDIRC Если он формирует сигнал на вывод BIDIR или BIDIRC, он не может формировать сигнал на какие-либо другие выходы,
- если сигнал Output Enable не привязывается к VCC, буфер TRI должен формировать сигнал на вывод OUTPUT, OUTPUTC, BIDIR или BIDIRC Внутренние сигналы могут не иметь третьего состояния

Примитивы **FLIPFLOP** (ждущий мультивибратор) **и LATCH** (триггер-защелка)

DFF SRFF
DFFE SRFFE
JKFF TFF
JKFFE TFFE
LATCH

Примитивы **Latch** описывают D-триггер (триггер-защелка), управляемый уровнями тактовых сигналов

Когда входной сигнал разрешения D-триггера ENA (Latch Enable) имеет высокий уровень, D-триггер пропускает сигнал от D к Q Если вход ENA имеет низкий уровень, сохраняется состояние Q, несмотря на входной сигнал D

Для устройств, не поддерживающих сигнал Latch Enable, логический синтез генерирует логические уравнения, содержащие D-триггеры с сигналами Latch Enable Эти логические уравнения корректно эмулируют логику, заданную в вашем проекте

Все ждущие мультивибраторы (Flipflop) переходят в рабочее состояние по положительному перепаду сигнала

Примитив **LATCH**

Прототип функции FUNCTION LATCH (D, ENA)
RETURNS (Q),

Входы		Выход
ENA	D	Q
L	X	Qo*
H	L	L
H	H	H

* Qo = уровень Q перед тактовым импульсом

Примитив **DFF** (D-триггер типа Flipflop)

Прототип функции FUNCTION DFF (D, CLK, CLRN, PRN)
RETURNS (Q),

Входы				Выход
PRN	CLRN	CLK	D	Q
L	H	X	X	H
H	L	X	X	L
L	L	X	X	Запрещенное состояние
H	H	↓	L	L
H	H	↓	H	H
H	H	L	X	Qo *
H	H	H	X	Qo

* Qo = уровень Q перед тактовым импульсом

Примитив **DFFE**

Прототип функции FUNCTION DFFE (D, CLK, CLRN, PRN, ENA)
RETURNS (Q),

Входы					Выход
CLRN	PRN	ENA	D	CLK	Q
L	H	X	X	X	L
H	L	X	X	X	H
L	L	X	X	X	Illegal
H	H	L	X	X	Qo*
H	H	H	L	↓	L
H	H	H	H	↓	H
H	H	X	X	L	Qo*

* Qo = уровень Q перед тактовым импульсом

Примитив **TFF** (Триггер T-типа, Flipflop)

Прототип функции FUNCTION TFF (T, CLK, CLRN, PRN)
RETURNS (Q),

Входы				Выход
PRN	CLRN	CLK	T	Q
L	H	X	X	H
H	L	X	X	L
L	L	X	X	Illegal
H	H	↓	L	Qo*
H	H	↓	H	Toggle
H	H	L	X	Qo*

* Qo = уровень Q перед тактовым импульсом

Примитив **TFFE**

Прототип функции FUNCTION TFFE (T, CLK, CLRN, PRN, ENA)
RETURNS (Q),

Входы					Выход
CLRN	PRN	ENA	T	CLK	Q
L	H	X	X	X	L
H	L	X	X	X	H
L	L	X	X	X	Illegal
H	H	L	X	X	Qo*
H	H	H	L		Qo*
H	H	H	H		Toggle
H	H	X	X	L	Qo*

Примитив **JKFF** (Триггер JK-типа, Flipflop)

Прототип функции: FUNCTION JKFF (J, K, CLK, CLRN, PRN)
RETURNS (Q),

Входы					Выход
PRN	CLRN	CLK	J	K	Q
L	H	X	X	X	H
H	L	X	X	X	L
L	L	X	X	X	Illegal
H	H	L	X	X	Qo*
H	H	↓	L	L	Qo*
H	H	↓	H	L	H
H	H	↓	L	H	L
H	H	↓	H	H	Toggle

* Qo = уровень Q перед тактовым импульсом

Примитив **JKFFE**

Прототип функции: FUNCTION JKFFE (J, K, CLK, CLRN, PRN, ENA)
RETURNS (Q);

Входы						Выход
ENA	PRN	CLRN	CLK	J	K	Q
X	L	H	X	X	X	H
X	H	L	X	X	X	L
X	L	L	X	X	X	Illegal
X	H	H	L	X	X	Qo*
H	H	H	⊓	L	L	Qo*
H	H	H	⊓	H	L	H
H	H	H	⊓	L	H	L
H	H	H	⊓	H	H	Toggle
L	H	H	X	X	X	Qo*

* Qo = уровень Q перед тактовым импульсом

Примитив **SRFF** (Триггер SR-типа Flipflop)

Прототип функции. FUNCTION SRFF (S, R, CLK, CLRN, PRN)
RETURNS (Q);

Входы					Выход
PRN	CLRN	CLK	S	R	Q
L	H	X	X	X	H
H	L	X	X	X	L
L	L	X	X	X	Illegal
H	H	L	X	X	Qo*
H	H	⊓	L	L	Qo*
H	H	⊓	H	L	H
H	H	⊓	L	H	L
H	H	⊓	H	H	Toggle

* Qo = уровень Q перед тактовым импульсом

Примитив **SRFFE**

Прототип функции· FUNCTION SRFFE (S, R, CLK, CLRN, PRN, ENA)
RETURNS (Q);

Входы						Выход
ENA	PRN	CLRN	CLK	S	R	Q
X	L	H	X	X	X	H
X	H	L	X	X	X	L
X	L	L	X	X	X	Illegal
X	H	H	L	X	X	Qo*
H	H	H	L	L	L	Qo*
H	H	H	⊓	H	L	H
H	H	H	⊓	L	H	L
H	H	H	X	H	H	Toggle
L	H	H	X	X	X	Qo*

3.8.14. Порты

Порт представляет собой вход или выход примитива, макрофункции или цифрового автомата. Порт может появляться в трех местах файла.

- порт, являющийся входом или выходом текущего файла, объявляется в секции подпроекта Subdesign,
- порт, являющийся входом или выходом текущего файла, может быть назначен (присвоен) выводу, логическому элементу или кристаллу в секции проекта Design;
- порт, являющийся входом или выходом примера (instance) примитива или файл проекта более низкого уровня, используется в логической секции Logic.

3 8 14 1. ПОРТЫ ТЕКУЩЕГО ФАЙЛА

Порт, являющийся входом или выходом текущего файла, объявляется в секции подпроекта Subdesign следующим образом:

<имя порта> : <тип порта> [= <значение порта по умолчанию>]

Есть следующие типы портов:

```
INPUT
MACHINE INPUT
OUTPUT
MACHINE OUTPUT
BIDIR
```

Если TDF-файл является файлом верхнего уровня иерархии, имя порта является синонимом имени вывода. Для портов типа INPUT и BIDIR может быть определено по умолчанию значение VCC или GND Это значение используется, только если порт остается неподключенным в случае использования примера данного TDFa в файле более высокого уровня.

Входные и выходные порты, объявленные в секции подпроекта Subdesign, могут быть назначены выводам, логическим элементам, кристаллам, кликам и логическим опциям при назначении ресурсов Resource Assignment в секции проекта Design. Назначения выводов могут быть сделаны только на верхнем уровне иерархии. Любые назначения выводов на более низких иерархических уровнях игнорируются.

В приводимом ниже примере в секции подпроекта Subdesign объявлены входные, выходные и двунаправленные порты, а в секции проекта Design два входных порта назначаются (присваиваются) выводам

```
DESIGN IS top
BEGIN
                DEVICE IS AUTO
                BEGIN
                        foo @ 1, bar @ 2 : INPUT;
                END;
END,
SUBDESIGN top
(
   foo, bar, clk1, clk2 : INPUT = VCC;
% VCC – это аначение порта по умолчанию %
   a0, a1, a2, a3, a4 : OUTPUT;
   b[7..0] : BIDIR;
)
```

Между файлами TDF, (GDF-файл) или *.WDF можно осуществлять импортирование и экспортирование цифровых автоматов (state machine), если разработчик определит входной или выходной порт как MACHINE INPUT или MACHINE OUTPUT в секции подпроекта Subdesign В прототипе функции (Function Prototype), описывающем этот файл, должно быть указано, какие порты являются цифровыми автоматами с памятью (state machine). Порты MACHINE INPUT и MACHINE OUTPUT можно использовать только в файлвх более низких уровней в иерархии проекта.

3 8 14 2 ПОРТЫ ПРИМЕРОВ (INSTANCE)

Соединение порта, являющегося входом или выходом функции в файл проекта более низкого уровня или примитивы, осуществляется в логической секции Logic Для соединения примитива, макрофункции или цифрового автомата с другими частями TDF файла нужно вставить примитив или макрофункцию как ссылку с ключевым словом **in-line** или дать в конструкции объявления примера **Instance** или объявить цифровой автомат в конструкции **State Machine**, а затем использовать порты функции в логической секции

Если разработчик для вставки примера примитива или макрофункции использует ссылку с ключевым словом **in-line**, важен порядок перечисления портов, но не их имена Этот порядок определяется в прототипе функции для примитива или макрофункции

Если для вставки примера разработчик использует конструкцию объявлвния примера **Instance**, важны имена портов, а не порядок их перечисления Имена портов даются в следующем формате

<имя примера> <имя порта>

Здесь <имя примера> представляет собой определенное пользователем имя функции, а <имя порта> совпадает с именем порта, объявленного в логической секции файла TDF более низкого уровня или совпадает с именем вывода в файле проекта другого типа Это <имя порта> является синонимом имени штыревого вывода для символа, который представляет пример данного файла проекта в файле (GDF-файл)

В приводимом ниже примере триггер (flipflop) типа D объявлен как переменная **reg** в секции объявления переменных **VARIABLE** и затем используется в логической секции

```
VARIABLE
            reg : DFF;
BEGIN
     reg.clk = clk
            reg.d = d
            out = reg.q
END;
```

Все поставляемые фирмой Altera примитивы и макрофункции имеют предопределенные имена портов (штыревые выводы), которые описываются в прототипе функции Наиболее широко используемые имена портов приведены в таблице

Имя порта	Описание
q	Выход двухстабильного мультивибратора (flipflop) или тригера-защелки (latch)
d	Информационный вход в триггер D-типа (flipflop) или триггер-защелку (latch)
t	Вход переключения (Toggle) триггера T-тила (flipflop)
j	Вход J триггера J-K-типа (flipflop)
k	Вход K триггера J-K-типа (flipflop)
s	Вход S триггера S-R-типа (flipflop)
r	Вход R триггера S-R-типа (flipflop)
clk	Вход тактового сигнала (Clock) триггера (flipflop)
ena	Вход отпирания тактового сигнала (Clock Enable) для триггера, отпирания защелки (Latch Enable) и отпирания цифрового автомата (Enable)
.prn	Вход установки триггера (flipflop) Preset с низким активным уровнем
clrn	Вход сброса триггера (flipflop) Clear с низким активным уровнем
reset	Вход цифрового автомата Reset с высоким активным уровнем
oe	Вход Output Enable (отпирание выходв) примитива TRI
.in	Основной вход примитивов TRI, SOFT, GLOBAL и LCELL
out	Выход примитивов TRI, SOFT, GLOBAL и LCELL

3.9. СИНТАКСИС ЯЗЫКА AHDL

3.9.1. Лексические элементы

Синтаксис лексических элементов языка AHDL описан ниже с использованием формул Бэкуса–Наура

<основание системы счисления> = B | Q | H | O | X | b | q | h | o | x

<число с основанием> ::= < основание системы счисления > " <цифра> { <цифра> }"

<цифра> ::= <буква> | <десятичная цифра>

<столбец> ..= <десятичная цифра >:1:2

<символ комментария> ··= <любой символ, кроме %> | %%

<комментарий> = % { < символ комментария > } %

| — { < символ комментария > } <конец строки>

<десятичное число> = <десятичная цифра>:1 10

<имя проекта> = <имя файла>

<вывод устройства> ::= <символ имени>·1·3

<десятичная цифра> = 0 | 1 | | 9

<имя файла> ·= (<символ имени>) 1 8 | ' (<символ имени в кавычках>) 1 8 '

<лабиринт> ::= <строка матрицы> <столбец>

<буква> = A | | Z | a | | z

<логический элемент> ::= LC <десятичная цифра>:1:3

| LC <десятичная цифра>.1.2 <лабиринт>

| MC <десятичная цифра>:1.3

| MC <десятичная цифра> 1 2 <лабиринт>

<символ имени> ··= <буква> | <десятичная цифра> | _

<число> ::= <десятичное число> | <число с основанием>

<имя порта> = (<символ имени> | /)·1 32 | ' (<символ имени в кавычках>

| /) 1 32 '

< символ имени в кавычках > = <символ имени> | -

<строка матрицы> ::= <буква>

<символ строки> = <любой символ, кроме " и символа перехода на новую строку> | ""

<строка> ::= " { <символ строки> }"

<символическое имя> = (<имя символа> | /) 1.32 | ' (<символ имени в кавычках>

| /) 1·32 '

Символическое имя без кавычек не может состоять из одних цифр

3.9.2. Основные конструкции языка AHDL

С помощью формул Бэкуса–Наура синтаксис файла TDF можно описать следующим образом

<файл AHDL > ··=

<оператор AHDL >

{ < оператор AHDL > }

< оператор AHDL > . =

<название>

| <задание константы>

| <прототип>

| <оператор включения >

| < варианты>

| < секция проекта >

| <секция подпроекта>

```
<операторы>  =
<оператор>
{ <оператор> }

<оператор>  =
<булево уравнение>
| <оператор выбора>
| <условный оператор>
| <оператор таблицы>
| <присвоение по умолчанию>

<название>  =
TITLE <строка>;

<задание константы>  =
CONSTANT <символическое имя> = <число>,

<прототип> ::=
FUNCTION <макрофункция> ( <список входов> )
RETURNS ( <список выходов> );
| FUNCTION <примитив> ( <список входов> )
RETURNS ( <список выходов> ),

<макрофункция> :'=
<имя проекта>

<примитив> :.=
<символическое имя>

<список входов> ::=
<группа прототипа> {, <группа прототипа> }

<список выходов> :.=
<группа прототипа> {, <группа прототипа> }

<оператор включения> ::=
INCLUDE <строка> ,

<вариант> ::=
<вариант устройства>
| <вариант 0-го бита AHDL >
| <логическая опция>

<вариант устройства> ..=
| SECURITY = (ON | OFF)
| TURBO = (ON | OFF | DEFAULT)

<вариант 0-го бита AHDL > .:=
BIT0 = (ANY-произвольный | LSB-младший | MSB-старший)

<логическая опция> ' =
CARRY_CHAIN = (AUTO | IGNORE | MANUAL | DEFAULT)
| CASCADE_CHAIN = (AUTO | IGNORE | MANUAL | DEFAULT}
| DECOMPOSE_GATES = (ON | OFF | DEFAULT)
| DUPLICATE_LOGIC_EXTRACTION = (ON | OFF | DEFAULT)
| EXPANDER_FACTORING = (ON | OFF | DEFAULT)
| MINIMIZATION = (FULL | PARTIAL | NONE | DEFAULT)
| MULTI-LEVEL_FACTORING = (ON | OFF | DEFAULT)
| NOT_GATE_PUSH-BACK = (ON | OFF | DEFAULT)
| OPTIMIZE = (AREA | DELAY | ROUTING | DEFAULT)
| PARALLEL_EXPANDERS = (ON | OFF | DEFAULT)
| PERIPHERAL_REGISTER = (ON | OFF | DEFAULT)
| REDUCE_LOGIC = (ON | OFF | DEFAULT)
| REFACTORIZATION = (ON | OFF | DEFAULT)
| REGISTER_OPTIMIZATION = (ON | OFF | DEFAULT)
| RESYNTHESIZE_NETWORK = (ON | OFF | DEFAULT)
| SLOW_SLEW_RATE = (ON | OFF | DEFAULT)
| SOFT_BUFFER_INSERTION = (ON | OFF | DEFAULT)
| STYLE = <имя стиля>
| SUBFACTOR_EXTRACTION = (ON | OFF | DEFAULT)
| TURBO = (ON | OFF | DEFAULT)
| XOR_SYNTHESIS = (ON | OFF | DEFAULT)

<имя стиля> ' =
FAST
| WYSIWYG
| NORMAL
| <стиль пользователя>

<стиль пользователя> ' = <символическое имя>
```

Каждый <вариант> за исключением BIT0 и имени стиля может быть сокращен до 3 первых символов по обе стороны от знака равенства (=) Однако фирма Altera рекомендует писать полные имена для удобства документирования

3.9.3. Синтаксис объявления названия

```
<название>  =
TITLE <строка>,
```

3.9.4. Синтаксис оператора включения

```
<оператор включения> ::=
INCLUDE <имя файла> ,
```

Описываемый в операторе включения файл должен иметь расширение **inc**, а <имя файла> не должно содержать путь

3.9.5. Синтаксис задания константы

```
<задание константы>  =
CONSTANT <символьное имя> = <число>,
```

3.9.6. Синтаксис прототипа функции

```
<прототип>  =
FUNCTION <макрофункция> ( <список входов> )
RETURNS ( <список выходов> ),
| FUNCTION <примитив> ( <список входов> )
RETURNS ( <список выходов> ),

<макрофункция>  =
<имя проекта>

<примитив> '':=
<символьное имя>

<список входов> ::=
<группа прототипа> {, <группа прототипа> }
```

```
<список выходов>  =
<группа прототипа> {, <группа прототипа> }
```

3.9.7. Синтаксис оператора вариантов

```
<варианты>  =
OPTIONS <вариант> {, <вариант> },

<вариант>  =
<вариант устройства>
| <вариант 0-го бита AHDL>
| <логическая опция>

<вариант устройства>  =
| SECURITY = (ON-вкл | OFF-откл)
| TURBO = (ON-вкл | OFF-откл | DEFAULT-по умолчанию)

<вариант 0-го бита AHDL> ::=
BIT0 = (ANY-произвольный | LSB-младший | MSB-старший)

<логическая опция>  =
CARRY_CHAIN = (AUTO-авто | IGNORE-игнорировать | MANUAL-
ручной | DEFAULT-по умолчанию)
| CASCADE_CHAIN = (AUTO | IGNORE | MANUAL | DEFAULT)
| DECOMPOSE_GATES = (ON-вкл | OFF-откл | DEFAULT-по
умолчанию)
| DUPLICATE_LOGIC_EXTRACTION = (ON | OFF | DEFAULT)
| EXPANDER_FACTORING = (ON | OFF | DEFAULT)
| MINIMIZATION = (FULL-полная | PARTIAL-частичная | NONE-нет
| DEFAULT-по умолчанию)
| MULTI-LEVEL_FACTORING = (ON | OFF | DEFAULT)
| NOT_GATE_PUSH-BACK = (ON | OFF | DEFAULT)
| OPTIMIZE = (AREA-площадь | DELAY-задержка | ROUTING-мар-
шрутизация | DEFAULT-по умолчанию)
| PARALLEL_EXPANDERS = (ON | OFF | DEFAULT)
| PERIPHERAL_REGISTER = (ON | OFF | DEFAULT)
| REDUCE_LOGIC = (ON | OFF | DEFAULT)
| REFACTORIZATION = (ON | OFF | DEFAULT)
| REGISTER_OPTIMIZATION = (ON | OFF | DEFAULT)
| RESYNTHESIZE_NETWORK = (ON | OFF | DEFAULT)
| SLOW_SLEW_RATE = (ON | OFF | DEFAULT)
| SOFT_BUFFER_INSERTION = (ON | OFF | DEFAULT)
| STYLE = <имя стиля>
| SUBFACTOR_EXTRACTION = (ON | OFF | DEFAULT)
| TURBO = (ON | OFF | DEFAULT)
| XOR_SYNTHESIS = (ON | OFF | DEFAULT)

<имя стиля> ::=
FAST
| WYSIWYG
| NORMAL
| <стиль пользователя>

<стиль пользователя> ..= <символьное имя>
```

1 Каждый <вариант> кроме BIT0 и имени стиля может быть сокращен до трех символов по обеим сторонам знака равенства (=). Однако фирма Altera рекомендует использовать полное имя для удобства документации

2 Типы вариантов, которые могут быть включены в оператор OPTIONS, определяются местоположением этого оператора и правилами построения содержимого файла

3.9.8. Синтаксис секции подпроекта Subdesign

```
<секция подпроекта>  =
SUBDESIGN <имя проекта>
(
{ <список сигналов> ; }
<список сигналов> [ , ]
)
[ <переменные> ]
BEGIN
<операторы>
END,

<список сигналов> ..=
<список портов>  <тип порта>

<тип порта> ..=
INPUT [ = VCC | = GND ]
| OUTPUT
| BIDIR [ = VCC | = GND ]
| MACHINE INPUT
| MACHINE OUTPUT
```

Ключевые слова **BEGIN** и **END** обрамляют логическую секцию, которая является телом секции подпроекта

3.9.9. Синтаксис секции переменных

```
<переменные>  =
VARIABLE
<список портов>  <тип переменной> ,
{ <список портов>  <тип переменной> , }

<тип переменной>  =
NODE
| <макрофункция>
| <примитив>
| <цифровой автомат (state machine)>
| <псевдоним цифрового автомата>

<макрофункция>  =
<имя проекта>

<примитив>  =
<символьное имя>
```

3.9.10. Синтаксис объявления цифрового автомата

```
<символьное имя>  <цифровой автомат (state machine)>,

<цифровой автомат (state machine)>  =
MACHINE [ <биты> ] <состояния>

<биты>  =
OF BITS ( <список портов> )
```

<состояния> ::=
WITH STATES (<состояние> {, <состояние> })

<состояние> ::=
<символьное имя> [= <значение состояния>]

<значение состояния> ::=
<число>
| <символьное имя>

3.9.11. Синтаксис объявления псевдонима цифрового автомата

<символьное имя> : MACHINE;

Объявление псевдонима цифрового автомата не задает биты или имена состояний. Эта информация импортируется из секции подпроекта через порт MACHINE INPUT файла более высокого уровня в иерархии проекта или же через прототип функции из файла более низкого уровня в иерархии проекта

3.9.12. Синтаксис логической секции

Логическая секция, заключенная между ключевыми словами **BEGIN** и **END**, представляет собой тело секции подпроекта Информация о синтаксисе приводится отдельно для каждого варианта логической секции.

3.9.13. Синтаксис булевых уравнений

<булево уравнение> ::=
<левая часть> = <правая часть>,

3.9.14. Синтаксис булевых уравнений управления

<символьное имя>.clk = <правая часть>;
[<символьное имя> reset = <правая часть>;]
[<символьное имя> ena = <правая часть>,]

Элемент <символьное имя> в булевом уравнении управления должен быть объявлен так же, как в объявлении цифрового автомата в секции переменных.

3.9.15. Синтаксис оператора Case

<оператор выбора> ::=
CASE <правая часть> IS
WHEN <выбор> => <операторы>
{ WHEN <выбор> => <операторы> }
[WHEN OTHERS => <операторы>]
END CASE;

<выбор> ::=
<группа констант> (, <группа констант> }

3.9.16. Объявлениe по умолчанию

<объявление по умолчанию> ::=
DEFAULTS
<задание значения> ;
{ <задание значения> ; }
END DEFAULTS;

<задание значения> ::=
<левая часть> = <группа констант>

3.9.17. Синтаксис условного оператора IF

<условный оператор> ::=
IF <правая часть> THEN
<операторы>
{ ELSIF <правая часть> THEN
<операторы> }
[ELSE <правая часть> THEN
<операторы>]
END IP,

3.9.18. Синтаксис встроенных (in-line) ссылок на макрофункцию или примитив

<макрофункция> (<список правых частей>)

<примитив> (<список правых частей> }

3.9.19. Синтаксис объявления таблицы истинности

<опервтор таблицы> .=
TABLE <входы> => <выходы> ;
<входные значения> => <выходные значения> ;
{ <входные значения> => <выходные значения> ; }
END TABLE,

<входы> ::=
<правая часть> {, <правая часть> }

<выходы> .:=
<левая часть> {, <левая часть> }

<входные значения> ::=
<группа констант> (, <группа констант> }

<выходные значения> ::=
<группа констант> {, <группа констант> }

3.9.20. Синтаксис порта

Порт, являющийся в текущем файле входом или выходом, объявляется в секции подпроекта в следующем формате:

<имя порта> : <тип порта> [= <значение порта по умолчанию>]

Имеются следующие типы портов:
INPUT
MACHINE INPUT
OUTPUT
MACHINE OUTPUT
BIDIR

Порты, являющиеся входами и выходами примера (instance} примитива или макрофункции, используются в следующем формате:

<имя примера>.<имя порта>

<имя примера> ::=
<символьное имя>

3.9.21. Синтаксис группы

Группы могут быть записаны в следующих двух нотациях

1. **Нотация <десятичное имя группы>** состоит из символьного имени, за которым следует диапазон десятичных чисел, заключенный в квадратные скобки, например a[4 .1] Допускается указание только одного диапазона после символьного имени

```
<десятичное имя группы> ::=
<символьное имя> [ <диапазон> ]

<диапазон>  =
<десятичное число>
 | <десятичное число>. <десятичное число>
```

Если группа уже определена, можно для краткости вместо диапазона указывать только пустые скобки [ ]

Вместо диапазона можно также использовать одно только число, например a[5] Однако в такой нотации обозначается только одно символьное имя, а не имя группы, и это эквивалентно записи a5

2. **Нотация <последовательное имя группы>** состоит из списка символьных имен, портов или чисел, разделенных запятой и заключенных в круглые скобки, например (a, b, c) Десятичные имена группы также можно перечислять в круглых скобках

```
<последовательное имя группы> ::= ( <список правых частей> )

<список правых частей>  =
[ <правая часть> ] {, [ <правая часть> ] }
```

Данная нотация полезна для задания портов примера (instance) функции, которая объявляется в объявлении примера

3.9.22. Синтаксические группы и списки

```
<группа констант>  =
<символьное имя>
 | <число>
 | VCC
 | GND
 | ( <список группы констант> )

<список группы констант>  =
[ <группа констант> ] {, [ <группа констант> ] }

<группа точек>  =
<имя порта>
 | <символьное имя> [ ]
 | <символьное имя> [ <диапазон> ]
 | ( <список группы точек> )

<список группы точек> ::=
[ <группа точек> ] {, [ <группа точек> ] }

<левая часть> ::=
< левая часть >  <группа точек>
 | <символьное имя>
 | <символьное имя> [ ]
 | <символьное имя> [ <диапазон> ]
 | ! < левая часть >
 | ( <список левых частей> )

<список левых частей>  =
[ < левая часть > ] {, [ < левая часть > ] }

<группа портов> ..=
<имя порта>
 | <символьное имя> [ <диапазон> ]
 | ( <список портов> )

<список портов>  =
<группа портов> {, <группа портов> }

<группа прототипа> ::=
<группа портов>
 | MACHINE <символьное имя>

<диапазон>  =
<десятичное число> [ . <десятичное число> ]

<правая часть>  =
<правая часть>  <группа точек>
 | ! <правая часть>
 | <число>
 | <символьное имя>
 | <символьное имя> [ ]
 | <символьное имя> [ <диапазон> ]
 | VCC
 | GND
 | <правая часть> == <правая часть>
 | <правая часть> >= <правая часть>
 | <правая часть> > <правая часть>
 | <правая часть> <= <правая часть>
 | <правая часть> < <правая часть>
 | <правая часть> != <правая часть>
 | <правая часть> # <правая часть>
 | <правая часть> !# <правая часть>
 | <правая часть> & <правая часть>
 | <правая часть> !& <правая часть>
 | <правая часть> $ <правая часть>
 | <правая часть> !$ <правая часть>
 | <правая часть> + <правая часть>
 | <правая часть> — <правая часть>
 | - <правая часть>
 | + <правая часть>
 | <макрофункция> ( <список правых частей> )
 | <примитив> ( <список правых частей> )
 | ( <список правых частей> )

<список правых частей> ::=
[ <правая часть> ] {, [ <правая часть> ] }
```