1.2 Введение

AHDL (язык описания аппаратуры фирмы Altera) является высокоуровневым, модульным языком, полностью интегрированным в систему MAX+PLUS II. Он особенно хорошо подходит для проектирования сложной комбинационной логики, шин, конечных автоматов, таблиц истинности и параметрической логики. Вы можете использовать текстовой редактор системы MAX+PLUS II или любой другой для создания текстовых файлов проектов (AHDL Text Design Files (.**tdf**)). Затем Вы можете откомпилировать TDF файлы для получения выходных файлов, пригодных для последующего моделирования, временного анализа и программирования устройства. Кроме того компилятор системы MAX+PLUS II может создавать текстовые файлы экспортирования (AHDL Text Design Export Files (.**tdx**)) и текстовые выходные файлы (Text Design Output Files (.**tdo**)), которые можно сохранить как TDF файлы и повторно использовать в качестве файлов проекта.

Как работает AHDL?

Операторы и элементы AHDL являются мощным, многогранным и легким в использовании средством. Вы можете создавать весь иерархический проект с помощью AHDL или смешивать TDF файлы с другими типами файлов в один проект.

Хотя Вы можете воспользоваться любым текстовым редактором для создания TDF файлов, но только текстовой редактор системы MAX+PLUS II дает Вам возможность воспользоваться его преимуществами, когда Вы вводите, компилируете и отлаживаете Ваш AHDL проект.

AHDL проект легко вставить в иерархию проекта. В текстовом редакторе Вы можете автоматически создавать символ, представляющий TDF файл и вводить его в графический файл проекта (Graphic Design File (.**gdf**)). Аналогичным образом Вы можете объединять заказные функции и свыше 300 мегафункций и макрофункций, поставляемых Альтерой, включая функции библиотеки параметрических модулей (LPM), в любой TDF файл.

Вы можете использовать команды меню Assign или Assignment & Configuration File (.acf) для создания ресурса и выбора устройства. Вы можете также проверить синтаксис и выполнить полную компиляцию для отладки и прогона Вашего проекта. Любые появляющиеся ошибки автоматически локализуются процессором сообщений (Message Processor) и выделяются в окне текстового редактора.

1.3 Как пользоваться языком AHDL

В данном разделе описывается как разрабатывать проект на AHDL и предлагаются советы по созданию успешных проектов.

1.3.1 Вставка шаблонов AHDL

Текстовой редактор позволяет Вам вставить шаблон любого из операторов или разделов AHDL в текущий файл. Шаблоны AHDL - это простой способ ввода синтаксических конструкций языка AHDL, увеличивающий скорость и точность ввода проекта.

Для вставки шаблона AHDL в текущую позицию ввода:

1. Откройте диалоговое окно **AHDL Template** с помощью команды меню Template.
2. Выберите имя в окне Template Section.
3. Нажмите OK.

После ввода шаблона в Ваш TDF файл, Вы должны заместить все переменные в шаблоне на Вашу собственную логику. Каждое ключевое слово AHDL выделено заглавными буквами, а каждое имя переменной начинается с двух символов подчеркивания (_ _) чтобы помочь Вам идентифицировать их.

1.3.2 Создание текстового выходного файла

Вы можете создать один или больше текстовых выходных файлов проекта (Text Design Output Files (.**tdo**)), которые содержат AHDL эквивалент полностью оптимизированной логики для устройства, применяющегося в проекте. Кроме того Компилятор создает также один или больше выходных файлов назначения и конфигурации (Assignment & Configuration Output Files (.**aco**)).

Вы можете сохранить TDO файл как текстовой файл проекта, отредактировать его, определить его как проект с помощью команд меню File: **Project Name** или **Project Set Project to Current File** и перекомпилировать проект (Вы должны также сохранить ACO файл как файл Assignment & Configuration File если Вы хотите сохранить распределения для устройства).

TDO файлы облегчают обратную аннотацию и сохраняют имеющийся логический синтез проекта. Для проекта с несколькими устройствами TDO файлы позволяют Вам зафиксировать проект и схему расположения выводов каждого устройства в проекте.

Для создания TDO файла для проекта:

1. Включите опцию **Generate AHDL TDO File** в команде меню Processing.
2. Для начала компиляции выберите кнопку **Start** в окне компилятора или одну из команд в меню File: **Project Save & Compile** или **Project Save, Compile & Simulate** в любом из приложений MAX+PLUS II.

1.3.3 Использование чисел

Числа применяются для определения значений констант в булевских выражениях и уравнениях, в арифметических выражениях, а также значения параметров. AHDL поддерживает все комбинации десятичных, двоичных, восьмеричных и шестнадцатеричных чисел.

Файл decode1.tdf, приведенный ниже, описывает дешифратор адреса, который генерирует активный высокий сигнал разрешения кристалла, когда адрес равен 370 Hex.

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

В этом простом примере десятичные числа 15 и 0 используются для определения битов шины адреса. Шестнадцатеричное число H"0370" определяет декодируемый адрес.

1.3.4 Использование констант и оценочных функций

Вы можете использовать константу в AHDL файле, давая ей дескриптивное имя на число или текстовую строку. Аналогичным образом Вы можете использовать оценочную функцию, давая ей дескриптивное имя на арифметическое выражение. Это имя, которое можно использовать по всему файлу, может быть более информативным и читаемым, чем число, строка или арифметическое выражение. Например, числовая константа UPPER_LIMIT более информативна, чем число 130.

Константы и оценочные функции особенно полезны, если одно и тоже число, строка или арифметическое выражение повторяются несколько раз в файле: если оно изменяется, то требуется изменить только один оператор. В AHDL константы реализуются с помощью оператора Constant, а оценочные функции с помощью оператора Define.

AHDL снабжен также предопределенными оценочными функциями USED, CEIL, и FLOOR.

Файл decode2.tdf, приведенный ниже, имеет туже самую функциональность как и decode1.tdf, но использует константу IO_ADDRESS вместо числа H"0370".

```
CONSTANT IO_ADDRESS = H"0370";
SUBDESIGN decode2
```

```
(
  a[15..0] : INPUT;
  ce       : OUTPUT;
)
BEGIN
  ce = (a[15..0] == IO_ADDRESS);
END;
```

Вы можете определить константы и оценочные функции с помощью арифметических выражений. Компилятор оценивает арифметические операторы в арифметическом выражении и сокращает их до числовых значений. Логика для этих выражений не создается.

Файл strcmp.tdf, приведенный ниже, определяет константу FAMILY и использует ее в операторе Assert для проверки того, является ли текущее семейство устройств FLEX 8000.

```
PARAMETERS
(
DEVICE_FAMILY
% DEVICE_FAMILY является предопределенным параметром %
);

CONSTANT FAMILY = "FLEX8000";

SUBDESIGN strcmp
(
  a : INPUT;
  b : OUTPUT;
)
BEGIN
  IF (DEVICE_FAMILY == FAMILY) GENERATE
    ASSERT
      REPORT "Обнаружена компиляция для FLEX8000 "
      SEVERITY INFO;
    b = a;
```

```
  ELSE GENERATE
    ASSERT
      REPORT " Обнаружена компиляция для % семейства"
        DEVICE_FAMILY
      SEVERITY ERROR;
    b = a;
  END GENERATE;
END;
```

Файл minport.tdf, приведенный ниже, определяет оценочную функцию MAX, которая гарантирует минимальную ширину порта в разделе Subdesign.

```
PARAMETERS (WIDTH);
DEFINE MAX(a,b) = (a > b) ? a : b;
SUBDESIGN minport
(
  dataA[MAX(WIDTH,0)..0] : INPUT;
  dataB[MAX(WIDTH,0)..0] : OUTPUT;
)
BEGIN
  dataB[] = dataA[];
END;
```

1.3.5 Комбинаторная логика

Комбинационная логика реализуется на языке AHDL с помощью булевых выражений и уравнений, таблиц истинности, и множества мега и макрофункций. Примеры комбинационных функций включают дешифраторы, мультиплексоры и сумматоры.

1.3.5.1 Реализация булевых выражений и уравнений

Булевы выражения являются набором узлов, чисел, констант и других булевых выражений, разделенных операторами и/или компараторами и дополнительно сгруппированных с помощью скобок. Булево уравнение устанавливает узел или шину равной величине булевого выражения.

Файл boole1.tdf, приведенный ниже, демонстрирует два простых булевых выражения, представляющие два логических вентиля.

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

В этом файле выход out1 является логическим И входов a1 и инверсии a0, а выход out2 логическим ИЛИ out1 и b. Порядок следования их в файле не важен.

1.3.5.2 Именование булевых операторов и компараторов

Вы можете именовать булевы операторы и компараторы для облегчения ввода присваивания ресурсов и интерпретации раздела уравнений в файле отчета проекта.

Файл boole3.tdf, приведенный ниже, идентичен с файлом boole1.tdf, но использует именованные операторы. Имя оператора отделяется от оператора знаком двоеточия; имя может содержать до 32 символов.

```
SUBDESIGN boole3
(
  a0, a1, b  : INPUT;
  out1, out2 : OUTPUT;
```

```
)
BEGIN
  out1 = a1 tiger:& !a0;
  out2 = out1 panther:# b;
END;
```

Следующие отрывки из файла отчета показывают различие между boole3.rpt и boole1.rpt для первых двух уравнений.

```
-- boole3.rpt equations:
-- Node name is 'out1' from file "boole3.tdf" line 7, col 2
-- Equation name is 'out1', location is LC3_A1, type is output
        out1 = tiger~0;

-- Node name is 'tiger~0' from file "boole3.tdf" line 7, column 18
-- Equation name is 'tiger~0', location is LC2_A1, type is buried
        tiger~0 = LCELL( _EQ002);
          _EQ002 = !a0 &  a1;

-- boole1.rpt equations:
-- Node name is 'out1' from file "boole1.tdf" line 7, col 2
-- Equation name is 'out1', location is LC3_A1, type is output
        out1 = _LC2_A1;

-- Node name is ':33' from file "boole1.tdf" line 7, col 12
-- Equation name is '_LC2_A1', type is buried

        LC2_A1 = LCELL( _EQ001);
          _EQ001 = !a0 &  a1;
```

В зависимости от логики уравнения именованный оператор может представлять несколько имен узлов, однако, все имена относятся к имени оператора и, поэтому, узлы легче распознаются в файле отчета. В файле boole3.rpt единственный узел, tiger~0, создается для первого уравнения. В файле boole1.tdf компилятор связывает цепь ID :33 с тем же самым узлом.

После того, как Вы откомпилировали проект Вы можете использовать имена узлов, приведенные в файле отчета, для введения присваивания ресурса для дальнейшей компиляции, даже если логика проекта изменена. Имена логических ячеек, созданные из именованных операторов, остаются постоянными, если Вы изменили несвязанную с ними логику в файле. Например, Вы можете ввести присваивание для узла tiger~0. В противоположность этому, если операторы неименованы, доступны только ID номера цепей, и эти имена произвольно переназначаются при каждой компиляции.

1.3.5.3 Объявление узлов

Узел, который объявляется с помощью объявления Node в разделе Variable, можно использовать для хранения значения промежуточного выражения.

Объявления узлов особенно полезны, когда булево выражение используется повторно. Булево выражение можно заменить дескриптивным именем узла, которое легче читается.

Файл boole2.tdf, приведенный ниже, содержит ту же самую логику что и файл boole1.tdf, но имеет только один выход.

```
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;
```

Этот файл объявляет узел a_equals_2 и связывает его с выражением a1 & !a0. При использовании узлов можно сохранять ресурсы устройства, когда узел используется в нескольких выражениях.

Можно использовать как обычные узлы (NODE), так и тристабильные узлы (TRI_STATE_NODE). NODE и TRI_STATE_NODE различаются в том, что несколько присваиваний на них дают различные результаты.

Присваивания на узлы типа NODE связывают сигналы вместе с помощью функций ПРОВОДНОЕ-И или ПРОВОДНОЕ-ИЛИ. Значения по умолчанию, объявленные в операторах Defaults, определяют поведение: VCC представляет функцию ПРОВОДНОЕ-И, а GND представляет функцию ПРОВОДНОЕ-ИЛИ.

Присваивания на TRI_STATE_NODE привязывают сигналы к одному и тому же узлу.

Если только одной переменной назначается тип TRI_STATE_NODE, то она трактуется как NODE.

1.3.5.4 Определение шин

Шина, которая может включать до 256 членов (битов), трактуется как коллекция узлов и работает как одно целое. Имя шины можно определить с помощью имени с одним диапазоном, имени с двумя диапазонами или именем в последовательном формате.

В булевых уравнениях шина может приравниваться булеву выражению, другой шине, единственному узлу, VCC, GND, 1 или 0. В каждом из этих случаев значение шины различно. Оператор Options можно использовать для определения того, каким будет самый младший бит: наиболее значимым битом(MSB) или наименее значимым битом(LSB) или каким-либо другим.

Как только шина определена, скобки [ ] являются коротким способом определения всего диапазона. Например, a[4..1] можно также указать как a[]; b[5..4][3..2] можно представить как b[][].

Файл group1.tdf, приведенный ниже, демонстрирует булевы выражения, которые определяют несколько шин.

```
OPTIONS BIT0 = MSB;
CONSTANT MAX_WIDTH = 1+2+3-3-1;
```

```
% MAX_WIDTH = 2 %
SUBDESIGN group1
(
a[1..2], use_exp_in[1+2-2..MAX_WIDTH] : INPUT;
d[1..2],use_exp_out[1+2*2-4..MAX_WIDTH] : OUTPUT;
dual_range[5..4][3..2] : OUTPUT;
)
BEGIN
  d[] = a[] + B"10";
  use_exp_out[] = use_exp_in[];
  dual_range[][] = VCC;
END;
```

В этом примере оператор Options используется для определения того, что самый правый бит шины будет MSB, а десятичная 1 прибавляется к шине a[]. Если ко входу a[] прикладывается 00, то результатом этой программы будет d[] == 1. Шины use_exp_in[] и use_exp_out[] показывают как константы и арифметические выражения можно использовать для ограничения диапазонов шин.

Следующие примеры иллюстрируют использование шин:

- Когда шина приравнивается к другой шине того же самого размера, то каждый член справа приравнивается каждому члену слева в соответствующей позиции.
- Когда шина приравнивается к VCC или GND, все биты шины соединяются с этим значением.
- Когда шина приравнивается к 1, только наименее значимый бит шины соединяется со значением VCC. Остальные биты шины соединяются с GND.
- Когда приравниваются шины не одинакового размера, количество битов шины с левой стороны уравнения должно точно делиться на количество битов шины с правой стороны уравнения. Например, уравнение

  a[4..1] = b[2..1] правильно.

В этом уравнении биты отображаются следующим образом:

a4 = b2
a3 = b1
a2 = b2
a1 = b1

1.3.5.5 Реализация условной логики

Операторы If Then и Case идеально подходят для реализации условной логики. Операторы If Then оценивают одно или несколько булевых выражений и описывают поведение для различных значений выражения. Операторы Case являются списком альтернатив, которые доступны для каждого значения выражения. Они оценивают выражение, а затем выбирают направление действия на основе значения выражения.

Условную логику, реализуемую с помощью операторов If Then и Case, не следует путать с логикой, создаваемой условно оператором If Generate. Эта логика не обязательно является условной.

1.3.5.5.1 Оператор If Then

Файл priority.tdf, приведенный ниже, демонстрирует приоритетный шифратор, который преобразует уровень активного входа с наивысшим приоритетом в значение.

```
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;
```

В этом примере входы high, middle, и low оцениваются для определения того, является ли их уровни равными VCC. Оператор If Then активизирует уравнения, которые следуют за активной IF или ELSE областями и, если вход high высокий, то highest_level[] равен 3.

Если активизируется более одного входа, то оператор If Then оценивает приоритет входов в порядке следования областей IF и ELSIF (первая область имеет наивысший приоритет).

Если ни один из входов не активизирован, по срабатывает уравнение, следующие за ключевым словом ELSE.

1.3.5.5.2 Оператор Case

Файл decoder.tdf, приведенный ниже, описывает дешифратор 2 в 4 бита. Он преобразует 2-битный код в унарный код.

```
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;
```

В этом примере входной код шины имеет значения 0, 1, 2 или 3. В операторе Case за символом => следует активизируемое уравнение. Например, если code[] равен 1, то выход out1 устанавливается в B"0010". Поскольку все значения выражения различны, в одно время можно активизировать только одну область WHEN

1.3.5.5.3 Оператор If Then против оператора Case

Операторы If Then и Case подобны. В некоторый случаях Вы можете использовать любой из двух операторов для получения того же самого результата.

Но между ними существует важное различие:

- В операторе If Then можно использовать любые виды булевых выражений. Каждое выражение, следующее за IF или ELSIF областями, может быть несвязанно с другими выражениями в операторе. В операторе Case, напротив, только одно булево выражение сравнивается с константой в каждой WHEN области.
- Использование ELSIF предложения может привести к логике, которая слишком сложна для компилятора, так как каждое следующее друг за другом предложение ELSIF должно еще проверять, ложность предыдущих IF/ELSIF предложений. Следующий пример показывает как компилятор интерпретирует оператор If Then. Если a и b сложные выражения, тогда инверсия этих выражений даст, возможно, даже более сложные выражения.

Оператор If Then	Интерпретация компилятора
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;
```

1.3.5.6 Создание дешифраторов

В AHDL для создания дешифратора Вы можете использовать или оператор Truth Table или lpm_compare или lpm_decode функции.

Файл 7segment.tdf, приведенный ниже, является дешифратором для комбинации светоизлучающих диодов (LED). LED отображают шестнадцатеричные числа.

```
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 выходной набор для всех 16 возможных входных наборов i[3..0] описан в операторе Truth Table

Файл decode3.tdf, приведенный ниже, является дешифратором адреса для реализации 16-битной 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"100XXXXXXXXXXXXX"    => 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 этом примере существуют тысячи входных наборов и описывать их все в операторе Truth Table непрактично. Вместо этого Вы можете использовать логический уровень X для указания того, что выход не зависит от соответствующего входа. Например, в первой строчке оператора TABLE выход rom должен быть высоким для всех 16,384 входных наборов addr[15..0], начинающихся с 00. Следовательно Вам необходимо точно определить только общую часть входного набора, а для остальных входов использовать символ X.

При использовании символов X Вы должны гарантировать отсутствие наложений между битовыми комбинациями в таблице истинности. Язык AHDL предполагает что одновременно только одно условие в таблице истинности может быть истинно.

Файл decode4.tdf, приведенный ниже, использует функцию lpm_decode для получения такой же функциональности как и файл 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;
```

1.3.5.7 Использование для переменных значений по умолчанию

Вы можете определить значение по умолчанию для узла или шины, который используете, когда его величина не указана где-нибудь в другом месте файла. AHDL позволяет Вам также присваивать значение узлу или шине более одного раза в одном файле. Если эти присваивания конфликтуют, то значение по умолчанию используется для разрешения конфликтов. При отсутствии определения значения по умолчанию ему присваивается значение GND.

Значение по умолчанию определяется с помощью оператора Defaults для переменных, использующихся в операторах Truth Table, If Then, и Case.

Вы не должны путать значения по умолчанию для переменных со значениями по умолчанию для портов, которые присваиваются в разделе Subdesign.

Файл default1.tdf, приведенный ниже, оценивает входы и выбирает один из пяти ASCII кодов, основываясь на входах.

```
SUBDESIGN default1
(
  i[3..0]             : INPUT;
  ascii_code[7..0] : OUTPUT;
)
BEGIN
  DEFAULTS
ascii_code[] = B"00111111"; % ASCII  код "?" %
  END DEFAULTS;

  TABLE
    i[3..0]    => ascii_code[];
    B"1000" => B"01100001"; % "a" %
    B"0100" => B"01100010"; % "b" %
    B"0010" => B"01100011"; % "c" %
    B"0001" => B"01100100"; % "d" %
  END TABLE;
END;
```

Когда входной набор совпадает с одним из наборов, приведенным с левой стороны оператора Truth Table, выходы устанавливаются в соответствии с комбинацией справа. Если совпадения не происходит, выходы принимают значения по умолчанию B"00111111".

Файл default2.tdf, приведенный ниже, иллюстрирует как возникают конфликты, когда одному узлу присваивается более одного значения и как эти конфликты разрешаются языком 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 этом примере wire_or присваиваются значения a, b, или c, в зависимости от значений сигналов select_a, select_b, и select_c. Если ни один из этих сигналов не равен VCC, тогда wire_or принимает значение GND.

Если больше одного из сигналов select_a, select_b, или select_c принимают значение VCC, тогда сигнал wire_or является логическим ИЛИ соответствующих входных значений .

Сигнал wire_and работает таким же образом, за исключением того, что по умолчанию он устанавливается в VCC, когда ни один из "select" сигналов не равен VCC и равен логическому И соответствующих входов, когда более одно сигнала принимает значение VCC.

1.3.5.8 Реализация логики с активными низкими уровнями

Активный низкий сигнал становится активным, когда его значение равно GND. Активные низкие сигналы могут быть полезны при управлении памятью, периферийными устройствами и микропроцессорными кристаллами.

Файл daisy.tdf, приведенный ниже, является модулем схемы арбитра по методу дейзи-цепочки. Он принимает запросы на доступ к шине от самого себя и от следующего модуля в цепочке. Доступ к шине предоставляется модулю с наивысшим приоритетом, запросившим его.

```
SUBDESIGN daisy
(
/local_request     : INPUT;
/local_grant       : OUTPUT;
/request_in        : INPUT; % от младшего приоритета %
/request_out       : OUTPUT; % к старшему приоритету %
/grant_in: INPUT; % от старшего приоритета %
/grant_out         : OUTPUT;% к младшему приоритету %
)
BEGIN
        DEFAULTS
/local_grant = VCC;% активные низкие выходы %
/request_out=VCC;
```

```
%должны быть равны по умолчанию %
/grant_out = VCC;          % 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 этом файле активные низкие. Altera рекомендует, чтобы Вы выбирали схему именования узлов, ясно указывающую имена активных низких сигналов, например, начальное "n" или слеш (/).

Операторы If Then используются для определения активности модулей, т.е. равен ли сигнал GND. Если сигнал активный, то активизируются уравнения, следующие за соответствующим оператором If Then.

1.3.5.9 Реализация двунаправленных выводов

MAX+PLUS II позволяет конфигурировать выводы I/O как двунаправленные. Двунаправленные выводы можно определить с помощью порта BIDIR, который соединяется с выходом примитива TRI. Сигнал между выводом и примитивом TRI является двунаправленным и может использоваться для управления другой логикой проекта.

Файлы bus_reg2.tdf и bus_reg3.tdf, приведенные ниже, оба реализуют регистр, который фиксирует значение, обнаруженное на тристабильной шине. Также они могут выдавать запомненное значение обратно на шину. Один файл реализует DFF и TRI функции с помощью ссылок на логические функции. Другой файл использует объявления Register и Instance, соответственно, в разделе Variable.

```
SUBDESIGN bus_reg2
    (
    clk : INPUT;
    oe  : INPUT;
    io  : BIDIR;
    )
    VARIABLE
      dff_out : NODE;

BEGIN

  dff_out = DFF(io, clk, ,);
  io = TRI(dff_out, oe);

END;
```

```
SUBDESIGN bus_reg3
  (
clk : INPUT;
oe  : INPUT;
io  : BIDIR;
  )
VARIABLE
   my_dff  : DFF;
     my_tri  : TRI;
BEGIN

 my_dff.d = io;
  my_dff.clk = clk;
my_tri.in = my_dff.q;
 my_tri.oe = oe;
io = my_tri.out;

END;
```

Двунаправленный сигнал io, управляемый примитивом TRI, используется в качестве входа d триггера D (DFF).

Также Вы можете присоединить двунаправленный вывод из TDF файла нижнего уровня к выводу верхнего уровня. Двунаправленный выходной порт подпроекта должен соединяться с двунаправленным выводом с верхнего уровня иерархии. Прототип Function для TDF файла нижнего уровня должен включать двунаправленный вывод в предложении RETURNS. Файл bidir1.tdf, приведенный ниже, включает четыре экземпляра функции bus_reg2, упомянутой выше.

```
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;
```

1.3.5.10 Реализация тристабильных шин

Примитивы TRI, которые управляют портами OUTPUT или BIDIR, имеют вход разрешения выхода (Output Enable), который переводит выход в высокоимпедансное состояние.

Вы можете создать тристабильную шину путем соединения примитивов TRI и портов OUTPUT или BIDIR вместе с помощью узла TRI_STATE_NODE типа. Схема управления должна обеспечивать разрешение не более одного выхода в одно и тоже время.

Файл tri_bus.tdf, приведенный ниже, реализует тристабильную шину, используя узел TRI_STATE_NODE типа, созданный в объявлении Node.

```
SUBDESIGN tri_bus
(
  in[3..1], oe[3..1] : INPUT;
  out1               : OUTPUT;
)

VARIABLE
  tnode : TRI_STATE_NODE;
BEGIN
  tnode = TRI(in1, oe1);
  tnode = TRI(in2, oe2);
  tnode = TRI(in3, oe3);
  out1 = tnode;
END;
```

В этом примере несколько присваиваний узлу tnode, связывают сигналы вместе. Для реализации тристабильной шины требуется тип TRI_STATE_NODE, вместо типа NODE: для типа NODE сигналы связываются вместе с помощью проводного И или проводного ИЛИ, тогда как для типа TRI_STATE_NODE сигналы соединяются с тем же самым узлом. Однако, если только одна переменная присваивается узлу TRI_STATE_NODE, то она трактуется как переменная обычного типа NODE.

1.3.6 Последовательностная логика

Последовательную логику в языке AHDL можно реализовать с помощью конечных автоматов, регистров и защелок или используя библиотеку параметрических модулей (LPM). Конечные автоматы особенно удобны для реализации последовательной логики. Другими примерами являются счетчики и контроллеры.

1.3.6.1 Объявление регистров

Регистры запоминают значения данных и синхронизируют их с помощью сигнала Clock. Вы можете объявить экземпляр регистра с помощью объявления Register в разделе Variable. (Вы можете также реализовать регистр используя ссылки на функции в разделе Logic). AHDL предлагает несколько примитивов регистров, а также поддерживает регистровые LPM функции.

После того как Вы объявили регистр, Вы можете соединить его с другой логикой в TDF файле, используя его порты. Порт экземпляра используется в следующем формате:

<имя экземпляра>.<имя порта>

Файл bur_reg.tdf, приведенный ниже, использует объявление Register для создания байтного регистра, который фиксирует значения входов d на переднем фронте Clock, когда вход 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;
```

Регистры объявляются в разделе Variable как DFFE(D триггер с сигналом разрешения). Первое булево уравнение в разделе Logic соединяет вход clk с портами Clock триггеров ff[7..0].

Второе уравнение соединяет вход загрузки с портами разрешения тактовой частоты. Третье уравнение соединяет входы данных d[7..0] с входными портами триггеров ff[7..0]. И четвертое уравнение соединяет выходы с выходными портами триггеров. Все четыре уравнения оцениваются совместно.

Вы можете также объявить T, JK, и SR триггеры в разделе Variable, а затем использовать в разделе Logic.

Если Вы хотите загрузить регистр на определенном переднем фронте глобального сигнала Clock, Altera рекомендует использовать вход разрешения тактовой частоты одного из DFFE, TFFE, JKFFE, или SRFFE триггеров для управления загрузкой регистра.

Файл lpm_reg.tdf, приведенный ниже, использует ссылку для реализации экземпляра функции lpm_dff, который обладает такой же функциональностью, как и файл bur_reg.tdf.

```
INCLUDE "lpm_dff.inc";
SUBDESIGN lpm_reg
(
  clk, load, d[7..0] : INPUT;
  q[7..0]          : OUTPUT;
)
BEGIN
q[] = lpm_dff (.clock=clk, .enable=load, .data[]=d[])
    WITH (LPM_WIDTH=8)
  RETURNS (.q[]);
END;
```

1.3.6.2 Объявление регистровых выходов

Вы можете объявить регистровые выходы TDF файла путем объявления выходных портов как триггеров в разделе Variable. Файл reg_out.tdf, приведенный ниже, имеет туже самую функциональность, что и файл bur_reg.tdf, но обладает регистровыми выходами.

```
SUBDESIGN reg_out
(
  clk, load, d[7..0] : INPUT;
  q[7..0]          : OUTPUT;
)
VARIABLE
q[7..0] : DFFE; % также объявлены как регистровые %
BEGIN
  q[].clk = clk;
  q[].ena = load;
  q[] = d[];
END;
```

Когда Вы присваиваете значение регистровым выходам в разделе Logic, то значение с d входов направляется в регистр. Выходы регистра не изменяются до тех пор, пока не появится возрастающий фронт сигнала Clock. Для определения тактового входа регистра используйте конструкцию <имя регистра>.clk в разделе Logic. Вы можете реализовать глобальный тактовый сигнал Clock, используя примитив GLOBAL с помощью логической опции **Global Signal** в диалоговом окне **Individual Logic Options**, которое Вы можете открыть из окна **Logic Options** (меню Assign), или с помощью опции **Automatic Global Clock** из диалогового окна **Global Project Logic Synthesis**(меню Assign).

В файле, приведенном ниже, каждый DFFE триггер, объявленный в разделе Variable, запитывает выход с тем же именем, поэтому Вы можете обратиться к выходам q триггеров без использования порта q.

В TDF файле высокого уровня выходные порты синхронизируются с выходными выводами. Когда Вы объявляете одинаковое имя для выходного порта и регистра, присваивания опций probe и logic применяются к выводу, а не регистру (за исключением логической опции Fast I/O). Поэтому, если Вы хотите протестировать регистр или использовать специфические для регистра логические опции, Вы должны по разному назвать регистры и порты.

1.3.6.3 Создание счетчиков

Счетчики можно определить с помощью D триггеров (DFF и DFFE) и операторов If Then или с помощью функции lpm_counter.

Файл ahdlcnt.tdf, приведенный ниже, реализует 16-битный суммирующий счетчик с загрузкой, который можно сбросить в ноль.

```
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 этом файле в разделе Variable объявляется 16 триггеров с имена count0 по count15. Оператор If Then определяет значение, которое загружается в триггеры на возрастающем фронте Clock.

Файл lpm_cnt.tdf, приведенный ниже, использует функцию lpm_counter для реализации той же функциональности, что и файл ahdlcnt.tdf.

```
INCLUDE "lpm_counter.inc";
SUBDESIGN lpm_cnt
(
  clk, load, ena, clr, d[15..0] : INPUT;
  q[15..0]                      : OUTPUT;
)
VARIABLE
  my_cntr: lpm_counter WITH (LPM_WIDTH=16);

BEGIN
  my_cntr.clock  = clk;
  my_cntr.aload  = load;
  my_cntr.cnt_en = ena;
  my_cntr.aclr   = clr;
  my_cntr.data[] = d[];
  q[] = my_cntr.q[];
END;
```

1.3.7 Конечные автоматы

В языке AHDL конечные автоматы реализуются также легко как таблицы истинности и булевы уравнения. Язык структурирован настолько, что Вы можете или сами присвоить значения состояниям или позволить компилятору MAX+PLUS II сделать эту работу за Вас.

Компилятор использует усовершенствованные эвристические алгоритмы автоматического присваивания состояний, которые минимизируют логические ресурсы, требующиеся для реализации конечных автоматов.

От Вас просто требуется нарисовать диаграмму состояний и построить таблицу следующих состояний. Затем компилятор автоматически выполнит следующие функции:

- назначит биты, выбирая или T или D триггер (TFF или DFF) для каждого бита

- присвоит значения состояниям
- применит сложную технику логического синтеза для получения уравнений возбуждения

Для определения конечного автомата на языке AHDL, необходимо включить следующие элементы в TDF файл:

- объявление конечного автомата (раздел Variable)
- булевы уравнения управления (раздел Logic)
- переходы между состояниями в операторе Table или Case (раздел Logic)

Также Вы можете импортировать и экспортировать конечные автоматы между TDF файлами и другими файлами проекта, определяя входные и выходные сигналы как автоматные порты в разделе Subdesign.

1.3.7.1 Реализация конечных автоматов

Вы можете создать конечный автомат, объявив его имя, состояния и, дополнительно, биты конечного автомата в объявлении конечного автомата в разделе Variable.

Файл simple.tdf, приведенный ниже, обладает такой же функциональностью как D триггер (DFF).

```
SUBDESIGN simple
(
  clk, reset, 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 файле simple.tdf конечный автомат с именем ss объявлен в разделе Variable. Состояния автомата определены как s0 и s1, а биты состояния не объявлены.

Переходы конечного автомата определяют условия изменения к новому состоянию. Вы должны условно присвоить состояния в пределах одной поведенческой конструкции для определения переходов конечного автомата. Для этой цели рекомендуются операторы Case или Table. Например, в simple.tdf переходы из каждого состояния определяются в предложениях WHEN оператора Case.

Вы можете также определить выходное значение для состояния с помощью оператора If Then или Case. В операторах Case эти присваивания выполняются в предложениях WHEN. Например, в simple.tdf выход q присваивается GND, когда конечный автомат ss находится в состоянии s0 и VCC, когда автомат находится в состоянии s1.

Выходные значения можно также определить в таблицах истинности как описано в пункте 1.3.7.3Присваивание состояний.

1.3.7.2 Установка сигналов Clock, Reset & Enable

Сигналы Clock, Reset, и Clock Enable управляют триггерами регистра состояний конечного автомата. Эти сигналы определяются с помощью булевых уравнений управления в разделе Logic.

В файле simple.tdf, приведенном ниже, Clock конечного автомата управляется входом clk. Сигнал асинхронного сброса конечного автомата Reset управляется сигналом reset, который является активным высоким. В этом файле проекта объявление входа ena в разделе Subdesign и булева уравнения ss.ena = ena в разделе Logic подсоединяет сигнал Clock Enable.

```
SUBDESIGN simple
(
  clk, reset, ena, d : INPUT;
  q                  : OUTPUT;
)
VARIABLE
  ss: MACHINE WITH STATES (s0, s1);

BEGIN
  ss.clk = clk;
  ss.reset = reset;
  ss.ena = ena;
  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;
```

1.3.7.3 Присваивание состояний

Бит состояния - это выход триггера, который используется конечным автоматом для запоминания однобитного значения. В большинстве случаев Вы должны разрешить компилятору MAX+PLUS II присвоить биты состояния и значения для минимизации требующихся логических ресурсов: логический синтезатор автоматически минимизирует количество необходимых битов состояния, оптимизируя как использование устройства так и производительность.

Однако некоторые конечные автоматы могут работать быстрее, используя значения состояний, которые требуют больше чем минимальное количество битов состояния. Кроме того, Вы можете захотеть, чтобы определенные биты состояния являлись выходами конечного автомата. Для управления этими случаями Вы можете объявить биты конечного автомата и значения в объявлении конечного автомата.

Команда **Global Project Logic Synthesis** (меню Assign) включает опцию **One-Hot State Machine Encoding** (позиционное кодирование состояний), которая автоматически реализует этот тип кодирования для проекта. Кроме того, компилятор автоматически реализует позиционное кодирование для устройств FLEX 6000, FLEX 8000, и FLEX 10K, несмотря на то, включена или нет эта опция. Если Вы точно назначили биты состояния, в добавление к использованию автоматического позиционного кодирования, логика Вашего проекта может быть реализована неэффективно.

Файл stepper.tdf, приведенный ниже, реализует контроллер 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 этом примере выходы phase[3..0], объявленные в разделе Subdesign, также объявлены как биты конечного автомата ss в объявлении конечного автомата. Заметьте, что ccw и cw никогда не должны одновременно равняться 1 в одной и той же таблице. AHDL предполагает, что только одно условие в таблице истинности является истинным в одно и тоже время, следовательно, перекрытие комбинаций битов может привести к непредсказуемым результатам.

1.3.7.4 Конечные автоматы с синхронными выходами

Если выходы конечного автомата зависят только от состояний автомата, Вы можете определить его выходы в предложении WITH STATES объявления конечного автомата.

Файл moore1.tdf, приведенный ниже, реализует автомат Мура на четыре состояния.

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

  TABLE
  % текущее    текущий    следующее  %
  % состояние  вход       состояние %
    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;
```

Этот пример определяет состояния конечного автомата с помощью объявления конечного автомата. Переходы между состояниями определены в таблице переходов, которая реализована с помощью оператора Table. В этом примере автомат ss имеет 4 состояния, но только один бит состояния (z). Компилятор автоматически добавляет другой бит и создает соответствующие присваивания для синтезированной переменной для представления автомата на 4 состояния. Этот автомат требует не менее 2 битов.

Когда значения состояний используются в качестве выходов, как в файле moore1.tdf, проект может использовать несколько логических ячеек, но логические ячейки могут требовать дополнительной логики для управления входами их триггеров. В этом случае модуль логического синтеза компилятора не сможет полностью минимизировать конечный автомат.

Другим способом проектирования конечного автомата с синхронными выходами является опускание присваиваний значений состояниям и точное объявление выходных триггеров. Файл moore2.tdf, приведенный ниже, иллюстрирует этот альтернативный метод.

```
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
% состояние    вход     состояние   выход %
    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;
```

Вместо определения выхода с помощью присваиваний значений состояниям в объявлении конечного автомата, этот пример включает столбец ‘‘ следующий выход ’’, после столбца ‘‘ следующее состояние ’’ в операторе Table. Этот метод использует D триггер (DFF), вызванный с помощью ссылки, для синхронизации выходов с тактовой частотой.

1.3.7.5 Конечные автоматы с асинхронными выходами

AHDL поддерживает реализацию конечных автоматов с асинхронными выходами. Выходы этих типов конечных автоматов могут изменяться при изменении входов, несмотря на переходы сигнала Clock.

Файл mealy.tdf, приведенный ниже, реализует автомат Мили на 4 состояния с 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
%  состояние    вход      выход       состояние %
    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;
```

1.3.7.6 Выход из некорректных состояний

Логика, созданная для конечного автомата компилятором MAX+PLUS II, будет вести себя так, как Вы описали в файле TDF. Тем не менее проекты конечных автоматов, которые точно объявляют биты состояния, и которые не используют позиционного кодирования, часто допускают значения битов состояния, которые не связаны с действительными состояниями. Эти не присвоенные значения называются не корректными состояниями. Проект, который вводит некорректные состояния, например, в результате нарушений времени предустановки и удержания, может приводить к неверным выходам. Хотя Altera рекомендует, чтобы входы конечного автомата удовлетворяли всем временным требованиям, Вы можете заставить конечный автомат принудительно вернуться из некорректного состояния в известное состояние с помощью оператора Case.

Для возвращения из некорректных состояний в проектах не использующих FLEX устройств, или проектов не использующих опцию позиционного кодирования, Вы должны назвать все некорректные состояния автомата. Предложение WHEN OTHERS в операторе Case, которое принуждает выполнить каждый переход из некорректного состояния в известное состояние, применяется только к состояниям, которые объявлены, но не упоминаются в предложении WHEN. Предложение WHEN OTHERS может форсировать принудительные переходы, только если все некорректные состояния объявлены в объявлении конечного автомата.

Для n-битного конечного автомата, существует 2^n возможных состояний. Если Вы объявили n бит Вы должны продолжать добавлять имена фиктивных состояний до тех пор, пока количество состояний не достигнет степени 2. Файл recover.tdf, приведенный ниже, содержит автомат, который может возвращаться из некорректных 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;
```

Этот пример содержит 3 бита: q2, q1, и q0. Следовательно существует 8 состояний. Так как объявлено только 5 состояний, были добавлены 3 фиктивных состояния.

1.3.8 Реализация иерархических проектов

TDF файлы, написанные на языке AHDL, можно смешивать с другими файлами в проектную иерархию. Файлы низкого уровня могут быть или файлами, поставляемыми Altera-ой, или мега и макрофункциями, определенными пользователем.

1.3.8.1 Использование непараметрических функций

MAX+PLUS II включает библиотеки примитивов и непараметрических макрофункций. Все логические функции MAX+PLUS II можно использовать для создания иерархических проектов. Мега и макрофункции автоматически устанавливаются в подкаталогах каталога \maxplus2\max2lib, созданного во время инсталляции. Логика примитивов встроена в AHDL.

Существует два способа использовать (т.е. вставлять экземпляр) непараметрическую функцию в языке AHDL:

- Объявить переменную для функции, т.е. имя экземпляра, в разделе Variable объявления Instance и использовать порты экземпляра функции в разделе Logic.
- Использовать ссылку на логическую функцию в разделе Logic TDF файла.

Объявления Instance обеспечивают именование узлов, которые полезны для ввода присваиваний ресурсов и моделирования проекта. С другой стороны с помощью ссылок на логические функции, имена узлов, основанные на ID номерах, можно менять при изменениях логики проекта.

Входы и выходы мега и макрофункций должны объявляться с помощью оператора прототипа функции (Function Prototype). Прототипы функций не требуются для примитивов. MAX+PLUS II снабжена файлами включения (Include Files), которые содержат прототипы для всех мега и макрофункций MAX+PLUS II в каталогах \maxplus2\max2lib\mega_lpm и \maxplus2\max2inc, соответственно. С помощью оператора Include, Вы можете передавать содержимое Include файла в файл TDF, для объявления прототипа мега или макрофункции MAX+PLUS II.

Файл macro1.tdf, приведенный ниже, демонстрирует 4-битный счетчик, соединенный с дешифратором 4 в 16. Экземпляры этих функций создаются с помощью объявлений Instance в разделе 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;
```

Этот файл использует операторы Include, для импортирования прототипов функций для двух макрофункций: 4count и 16dmux. В разделе Variable переменная counter объявлена как экземпляр функции 4count, а переменная decoder объявлена как экземпляр функции 16dmux. Входные порты функций, в формате <имя экземпляра>.<имя порта>, определены с левой стороны булевых уравнений в разделе Logic, а выходные порты с правой стороны.

Файл macro2.tdf, приведенный ниже, имеет такую же функциональность как и macro1.tdf, но создает экземпляры двух функций с помощью ссылок и узлов q[3..0]:

```
INCLUDE "4count";
INCLUDE "16dmux";
SUBDESIGN macro2
```

```
(
  clk        : INPUT;
  out[15..0]  : OUTPUT;
)
VARIABLE
  q[3..0]     : NODE;
BEGIN
  (q[3..0], ) = 4count (clk, , , , , GND, , , , );
% эквивалент подставляемой ссылки со связью по имени порта  %
%  (q[3..0], ) = 4count (.clk=clk, .dnup=GND);         %

% эквивалент подставляемой ссылки со связью по имени порта %
% и предложением RETURNS, определяющим требуемый выход %
%  q[3..0] = 4count (.clk=clk, .dnup=GND)              %
%               RETURNS (qd, qc, qb, qa);            %

  out[15..0]  = 16dmux (.(d, c, b, a)=q[3..0]);

% эквивалент подставляемой ссылки со связью по положению порта %
% out[15..0]  = 16dmux (q[3..0]); %

END;
```

Прототипы функций 4count.inc и 16dmux.inc приведены ниже:

```
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]);
```

Ссылки на 4count и 16dmux появляются в первом и втором булевых уравнениях в разделе Logic, соответственно. Ссылка на 4count использует связь по положению порта, тогда как ссылка на 16dmux использует связь по имени порта. Входные порты обоих макрофункций определяются с правой стороны ссылки, а выходные порты с левой.

Комментарии демонстрируют эквивалентные ссылки для различных видов связи с портом. В ссылке порты с правой стороны символа равенства (=) можно перечислять с помощью или связи по положению или по имени порта. Порты с левой стороны символа равенства всегда используют связь по положению. При использовании связи по положению важен порядок портов, так как существует соответствие один в один между порядком портов в прототипе функции и портами, определенными в разделе Logic. В ссылке на 4count запятые используются как разделители для портов, которые не соединяются точно.

Предложение RETURNS является дополнительным с ссылке. RETURNS можно использовать для перечисления подмножества выходов функции, которые используются в экземпляре.

Примитивы и макрофункции всегда имеют значения по умолчанию для не подсоединенных входов. Напротив, мегафункции необязательно имеют их.

1.3.8.2 Использование параметрических функций

MAX+PLUS II содержит параметрические мегафункции, а также функции библиотеки параметрических модулей (LPM). Например, параметры используются для определения ширины порта или будет ли блок памяти RAM реализован как синхронный или асинхронный. Параметрические функции могут содержать другие подпроекты, которые в свою очередь могут быть параметрическими или непараметрическими. Параметры можно использовать с некоторыми макрофункциями, которые не являются параметрическими. (Примитивы не могут быть параметрическими). Все логические функции MAX+PLUS II можно использовать для создания иерархических проектов. Мега и макрофункции автоматически устанавливаются в подкаталоги каталога \maxplus2\max2lib, созданного во время инсталляции; логика примитивов встроена в язык AHDL.

Параметрические функции объявляются с помощью ссылки на функцию или объявления Instance таким же образом как для непараметрических функций, но с некоторыми дополнительными шагами:

- Экземпляр логической функции должен содержать в себе предложение WITH, которое основано на предложении WITH в прототипе функции, в котором приводится список параметров, используемых экземпляром. Вы можете использовать предложение WITH для дополнительного присваивания значений параметрам экземпляра, однако, для всех необходимых параметров в функции, параметрическое значение должно прикладываться где-нибудь в пределах проекта. Если сам по себе экземпляр не содержит некоторых или всех значений для требуемых параметров, компилятор ищет их в порядке поиска значений параметров.
- Так как параметрические функции не обязательно имеют начальные значения для не подсоединенных входов, Вы должны убедиться что все необходимые порты подсоединены. С другой стороны, примитивы и макрофункции всегда имеют начальные значения для не подсоединенных входов.

Файл lpm_add1.tdf, приведенный ниже, реализует 8-битный сумматор с помощью ссылки на параметрическую мегафункцию lpm_add_sub.

```
INCLUDE "lpm_add_sub.inc";
SUBDESIGN lpm_add1
(
  a[8..1], b[8..1] : INPUT;
  c[8..1]          : OUTPUT;
  carry_out        : OUTPUT;
)
BEGIN
% Экземпляр мегафункции со связью порта по положению %
  (c[], carry_out, ) = lpm_add_sub(GND, a[], b[], GND,,)
                       WITH (LPM_WIDTH=8,
LPM_REPRESENTATION="unsigned");
%Эквивалентный экземпляр со связью по имени %
--(c[],carry_out,)= lpm_add_sub(.dataa[]=a[],.datab[]=b[],
--                          .cin=GND, .add_sub=GND)
-- WITH (LPM_WIDTH=8,
```

```
LPM_REPRESENTATION="unsigned");
END;
```

Прототип функции для lpm_add_sub приведен ниже:

```
FUNCTION lpm_add_sub(cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-1..0],
add_sub)
  WITH (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ADDERTYPE,
    ONE_INPUT_IS_CONSTANT)
  RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
```

Здесь требуется только параметр LPM_WIDTH и экземпляр функции lpm_add_sub в файле lpm_add1.tdf определяет значения параметров только для LPM_WIDTH и LPM_REPRESENTATION.

Файл lpm_add2.tdf, приведенный ниже, идентичен с lpm_add1.tdf, но реализует 8-битный сумматор с помощью объявления Instance.

```
INCLUDE "lpm_add_sub.inc";
SUBDESIGN lpm_add2
(
  a[8..1], b[8..1] : INPUT;
  c[8..1]          : OUTPUT;
  carry_out        : OUTPUT;
)
VARIABLE
  8bitadder : lpm_add_sub WITH (LPM_WIDTH=8,
              LPM_REPRESENTATION="unsigned");
BEGIN
  8bitadder.cin = GND
  8bitadder.dataa[] = a[]
  8bitadder.datab[] = b[]
  8bitadder.add_sub = GND
  c[] = 8bitadder.result[]
  carry_out = 8bitadder.cout
END;
```

1.3.8.3 Использование заказных мега и макро функций

Вы можете легко создать и использовать заказные мега и макрофункции в TDF файле.

После того как Вы определили логику для заказной функции в файле проекта, необходимо выполнить несколько шагов при использовании функции в других TDF файлах или в других типах файлов проекта.

Чтобы подготовить заказную мега или макрофункцию к использованию в другом файле проекта требуется:

1. Откомпилировать и при необходимости промоделировать файл проекта для обеспечения его правильного функционирования.
2. Если Вы планируете использовать функцию в нескольких проектах, Вы должны назначить каталог для хранения файла проекта в качестве библиотеки пользователя с помощью команды **User Libraries** (меню Options) или сохранения копии файла в существующем каталоге пользовательской библиотеки. Или же сохраните копию файла в каталоге, содержащем проект, который будет использовать заказную функцию.

I. С помощью открытия файла в окне текстового редактора создайте Include файл и символ, который представляет текущий файл:

   A. Выберите команду **Create Default Include File** (меню File) для создания Include файла, который можно использовать в TDF файле верхнего уровня. С помощью оператора Include Вы можете импортировать содержимое Include файла в TDF файл, объявляя прототип мега или макрофункции.

   B. Выберите команду **Choose Create Default Symbol** (меню File) для создания символа, который можно использовать в GDF файле.

После того как Вы подготовили функцию для других файлов проекта, Вы можете создать новый TDF файл и вставить экземпляр функции с помощью объявления экземпляра или подставляемой ссылки. Вы можете использовать заказные функции таким же образом как и функции, поставляемые Altera.

1.3.8.4 Импорт и экспорт конечных автоматов

Вы можете импортировать и экспортировать конечные автоматы между TDF файлами и другими файлами проекта, определяя входные и выходные порты как MACHINE INPUT или MACHINE OUTPUT в разделе Subdesign. Прототип функции, который представляет файл, содержащий конечный автомат, должен указывать, какие входы и выходы принадлежат конечному автомату с помощью предварения имен сигналов ключевым словом MACHINE.

Типы портов MACHINE INPUT и MACHINE OUTPUT нельзя использовать в файле проекта верхнего уровня. Хотя высокоуровневый файл с этими портами полностью не компилируется, Вы можете использовать команду **Project Save & Check** (меню File) для проверки его синтаксиса и команду **Create Default Include File** (меню File) для создания Include файла, который представляет текущий файл.

Вы можете переименовать конечный автомат с помощью временного имени, вводя объявление псевдоимени автомата в раздел Variable. Вы можете использовать это псевдоимя в файле, где создан этот автомат или в файле, который использует порт MACHINE INPUT для импорта конечного автомата. Затем Вы можете применить это имя вместо исходного имени автомата.

Файл ss_def.tdf, приведенный ниже, определяет и экспортирует конечный автомат ss с помощью порта ss_out.

```
        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;
```

Файл ss_use.tdf, приведенный ниже, импортирует конечный автомат с помощью порта ss_in.

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

Файл top1.tdf, приведенный ниже, использует ссылки для вставки экземпляров функций ss_def и ss_use. Прототипы функций для ss_def и ss_use содержат ключевые слова MACHINE, которые указывают какие входы и выходы являются автоматными.

```
FUNCTION ss_def (clk, reset, count)
         RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in)
         RETURNS (out);
```

```
        SUBDESIGN top1
        (
                sys_clk, /reset, hold : INPUT;
                sync_out            : OUTPUT;
        )
VARIABLE
        ss_ref: MACHINE; %объявление псевдоимени автомата  %
BEGIN
        ss_ref = ss_def(sys_clk, !/reset, !hold);
        sync_out = ss_use(ss_ref);
END;
```

Внешний конечный автомат можно также реализовать в TDF файле верхнего уровня с помощью объявления экземпляра в разделе Variable. Файл top2.tdf, приведенный ниже, имеет такую же функциональность, как и top1.tdf, но использует объявления экземпляров, вместо ссылок.

```
FUNCTION ss_def (clk, reset, count)
         RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in)
         RETURNS (out);

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;
```

1.3.9 Реализация LCELL & SOFT примитивов

Вы можете ограничить размер (масштаб) логического синтеза путем изменения переменных NODE на SOFT и LCELL примитивы. NODE переменные и LCELL примитивы обеспечивают наибольшее управление всем логическим синтезом. SOFT примитивы не предусмотрены для управления всем логическим синтезом.

NODE переменные, объявленные с помощью объявления Node в разделе Variable налагают несколько ограничений на логический синтез. Во время синтеза логический синтезатор заменяет каждый экземпляр переменной NODE логикой, которая представляет переменную. Затем он минимизирует логику для подгонки в одну логическую ячейку. Обычно этот метод дает самую большую скорость, но может приводить к слишком сложной логике.

SOFT буферы обеспечивают больший контроль по использованию ресурсов, чем NODE переменные. Логический синтезатор выбирает, когда заместить экземпляры SOFT примитивов с помощью LCELL примитивов. SOFT буферы могут помочь в исключении слишком сложной логики и упрощении подгонки проекта, но могут увеличить использование логических ячеек и уменьшить быстродействие.

LCELL примитивы обеспечивают наибольшее управление. Логический синтезатор минимизирует всю логику, которая управляет LCELL примитивом, так что она занимает только одну логическую ячейку. LCELL примитивы всегда реализуются в логической ячейке и никогда не удаляются из проекту даже если они запитываются одним входом. В последнем случае Вы можете использовать SOFT примитив вместо LCELL примитива, который будет удаляться во время логического синтеза.

MAX+PLUS II обеспечивает несколько логических опций, которые автоматически вставляют или удаляют SOFT и LCELL буферы в соответствующих местах проекта.

Следующая иллюстрация демонстрирует два варианта TDF файла: один реализуется с помощью NODE переменных, а другой с SOFT примитивами. В nodevar переменная odd_parity объявлена как NODE и затем ей присвоено значение булева выражения d0 $ d1 $ ... $ d8. В softbuf компилятор замещает некоторые SOFT примитивы на LCELL примитивы во время обработки для улучшения использования устройства.

TDF с NODE переменными:

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

TDF с SOFT примитивами:

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

1.3.10 Реализация RAM & ROM

MAX+PLUS II (и AHDL) снабжены несколькими LPM и мегафункциями, которые позволяют Вам реализовать RAM и ROM в устройствах MAX+PLUS II. Универсальная, масштабируемая природа каждой из этих функций гарантирует, что Вы можете использовать их для реализации любых поддерживаемых типов RAM или ROM в MAX+PLUS II.

Altera не рекомендует создавать заказные логические функции для реализации памяти. Вы должны использовать поставляемые Altera функции во всех случаях, где Вы хотите реализовать RAM или ROM.

Можно использовать следующие мегафункции для реализации RAM и ROM в MAX+PLUS II:

Имя	Описание
lpm_ram_dq	Синхронная или асинхронная память с раздельными портами ввода вывода
lpm_ram_io	Синхронная или асинхронная память с единственным портов I/O
lpm_rom	Синхронная или асинхронная память только для считывания
csdpram	Двухпортовая память
csfifo	Буфер FIFO

В этих LPM функциях параметры используются для определения ширины входных и выходных данных; количество запоминаемых слов; регистровые или нет входы данных, адреса, управления и выхода; должен ли включаться файл начального содержимого памяти для блока RAM и т.д.

1.3.11 Использование итеративно-генерируемой логики

Когда Вы хотите использовать несколько схожих блоков логики, Вы можете использовать оператор For Generate для итеративно-генерируемой логики.

Файл iter_add.tdf, приведенный ниже, демонстрирует пример итеративного создания логики:

```
CONSTANT NUM_OF_ADDERS = 8;
SUBDESIGN iter_add
(
a[NUM_OF_ADDERS..1], [NUM_OF_ADDERS..1],
cin                          : INPUT;
c[NUM_OF_ADDERS..1], cout : OUTPUT;
)
VARIABLE
sum[NUM_OF_ADDERS..1], carryout[(NUM_OF_ADDERS+1)..1] : NODE;
BEGIN
```

```
  carryout[1] = cin;
  FOR i IN 1 TO NUM_OF_ADDERS GENERATE
  sum[i] = a[i] $ b[i] $ carryout[i];    % Полный суматор %
  carryout[i+1] = a[i] & b[i] # carryout[i] & (a[i] $ b[i]);
END GENERATE;
  cout = carryout[NUM_OF_ADDERS+1];
  c[] = sum[];
        END;
```

В iter_add.tdf оператор For Generate используется для присваивания значений полным сумматорам. Выходной перенос carryout генерируется вместе с каждым полным сумматором.

Оператор If Generate особенно полезен с оператором For Generate, который раздельно управляет специальными случаями, например, в первом и последнем каскадах многокаскадного умножителя.

1.3.12 Использование условно-генерируемой логики

Вы можете создать логику условно с помощью оператора If Generate, если, например, хотите реализовать различное поведение в зависимости от значения арифметического выражения. Оператор If Generate приводит список последовательностей операторов поведения, которые активизируются после положительной оценки одного или более арифметических выражений.

Файл condlog1.tdf, приведенный ниже, использует оператор If Generate для реализации различного поведения выхода output_b на основании текущего семейства устройств.

```
PARAMETERS (DEVICE_FAMILY);
SUBDESIGN condlog1
(
  input_a : INPUT;
  output_b : OUTPUT;
)
BEGIN
  IF DEVICE_FAMILY == "FLEX8K" GENERATE
```

```
    output_b = input_a;
  ELSE GENERATE
    output_b = LCELL(input_a);
  END GENERATE;
END;
```

Оператор If Generate особенно полезен с оператором For Generate, который управляет специальными случаями различно.

MAX+PLUS II включает предопределенный параметр DEVICE_FAMILY, как показано в примере выше и предварительно вычисляемую функцию USED, которую можно использовать в арифметических выражениях. Параметр DEVICE_FAMILY можно использовать для проверки текущего семейства устройств для проекта, заданного с помощью команды **Device** (меню Assign). Функцию USED можно использовать для проверки того, использовался ли дополнительный порт в текущем экземпляре.

Вы можете найти многочисленные примеры операторов If Generate в TDF файлах, которые реализуют LPM функции в MAX+PLUS II. Эти файлы размещаются в подкаталоге mega_lpm каталога max2lib.

1.3.13 Использование оператора Assert

Вы можете использовать оператор Assert для проверки действительности любого произвольного выражения, которое использует параметры, числа, вычисляемые функции или использует или не использует статус порта. Вы могли бы, например, использовать оператор Assert для определения того, попадает ли значение дополнительного параметра в диапазон, определяемый значением второго параметра.

Когда Вы используете оператор Assert с условиями, Вы приводите список приемлемых значений для устанавливаемых условий. Если значение не допустимо, активизируется оператор и выдается сообщение. Если Вы используете оператор Assert без условий, то оператор активизируется всегда.

Компилятор вычисляет каждое условие только один раз после того, как модуль экстрактора списка связей (Netlist Extractor) разрешил все значения параметров. Оператор не может зависеть от значения сигнала, который реализуется в устройстве. Например, если оператор Assert помещается после оператора If Then вида IF a = VCC THEN c = d, то условие оператора Assert не может зависеть от значения a.

Файл condlog2.tdf, приведенный ниже, имеет такую же функциональность как и condlog1.tdf, но использует операторы Assert в разделе Logic для сообщения какая логика сгенерирована оператором If Generate.

```
PARAMETERS (DEVICE_FAMILY);
SUBDESIGN condlog2
(
  input_a : INPUT;
  output_b : OUTPUT;
)
BEGIN
  IF DEVICE_FAMILY == "FLEX8000" GENERATE
    output_b = input_a;
ASSERT
      REPORT "Компиляция для семейства FLEX8000"
      SEVERITY INFO;
  ELSE GENERATE
    output_b = LCELL(input_a);
    ASSERT (DEVICE_FAMILY == "FLEX10K")
      REPORT "Компиляция для семейства %", DEVICE_FAMILY;
  END GENERATE;
END;
```

1.4 Элементы

1.4.1 Зарезервированные слова

Зарезервированные ключевые слова используются для управления операторами AHDL, а также для предопределенных констант GND и VCC.

Зарезервированные ключевые слова отличаются от зарезервированных идентификаторов тем, что ключевые слова можно использовать как символьные имена при заключении их в одиночные кавычки ('), в то время как зарезервированные идентификаторы нельзя. Как те, так и другие можно свободно использовать в комментариях.

Altera рекомендует вводить все ключевые слова с заглавных букв для удобства чтения.

Для получения контекстно-зависимой помощи по ключевому слову сначала убедитесь, что TDF файл сохранен с расширением .tdf. Затем откройте файл в окне текстового редактора и нажмите Shift+F1 и щелкните кнопкой 1 на нем или выберите кнопку контекстно-зависимой помощи на панели инструментов.

Ниже приведен список всех зарезервированных ключевых слов:

AND	FUNCTION	OUTPUT
ASSERT	GENERATE	PARAMETERS
BEGIN	GND	REPORT
BIDIR	HELP_ID	RETURNS
BITS	IF	SEGMENTS
BURIED	INCLUDE	SEVERITY
CASE	INPUT	STATES
CLIQUE	IS	SUBDESIGN
CONNECTED_PINS	LOG2	TABLE
CONSTANT	MACHINE	THEN
DEFAULTS	MOD	TITLE
DEFINE	NAND	TO
DESIGN	NODE	TRI_STATE_NODE
DEVICE	NOR	VARIABLE
DIV	NOT	VCC
ELSE	OF	WHEN
ELSIF	OPTIONS	WITH
END	OR	XNOR
FOR	OTHERS	XOR

1.4.2 Зарезервированные идентификаторы

Ниже приведен список всех зарезервированных идентификаторов:

CARRY	JKFFE	SRFFE
CASCADE	JKFF	SRFF
CEIL	LATCH	TFFE
DFFE	LCELL	TFF
DFF	MCELL	TRI
EXP	MEMORY	USED
FLOOR	OPENDRN	WIRE
GLOBAL	SOFT	X

1.4.3 Символы

Символы ниже имеют в языке AHDL предопределенные значения. Этот список включает символы, которые используются в качестве операторов и компараторов в булевых выражениях и как операторы в арифметических выражениях.

Символ	Функция
_ (подчеркивание)	Идентификаторы, описанные пользователем и используемые как допустимые символы в символьных именах.
- (тире)	
/ (прямой слеш)	
-- (два тире)	Начинает однострочный комментарий в VHDL стиле
% (процент)	Ограничивает комментарий в AHDL стиле
( ) (круглые скобки)	Ограничивают и определяют последовательные имена шин. Например, шина (a, b, c) состоит из узлов a, b, и c. Ограничивают имена выводов в разделах

		Subdesign и операторах прототипов функций. Дополнительно, ограничивает входы и выходы таблиц истинности в операторах Truth Table. Заключают биты и состояния объявлений State Machine. Ограничивают операции наивысшего приоритета в булевых и арифметических выражениях. Ограничивают определения параметров в операторах Parameters, объявлениях Instance и параметрические имена в операторах Function Prototype и в подставляемых ссылках. Дополнительно, ограничивают условие в операторе Assert. Ограничивают аргументы оценочных функций в операторах Define.
[ ]	(скобки)	Ограничивают диапазон шины
'…'	(кавычки)	Ограничивают символьные имена
"…"	(двойные кавычки)	Ограничивают строки в операторах Title, Parameters, Assert. Ограничивают имена файлов в операторах Include. Ограничивают цифры в недесятичных числах
.	(точка)	Отделяет символьные имена переменных логической функции от имен портов. Отделяет расширения от имен файлов.
..	(эллипс)	Отделяет старший бит от младшего.
;	(точка с запятой)	Оканчивает операторы и разделы AHDL.
,	(запятая)	Отделяет символьные имена от типов в объявлениях.
=	(равно)	Присваивает входам значения по умолчанию

	GND и VCC в разделе Subdesign. Присваивает значения опциям в операторе Options. Присваивает значения по умолчанию параметрам в операторе Parameters или в подставляемой ссылке. Присваивает значения состояниям конечного автомата. Присваивает значения булевым уравнениям. Соединяет сигнал с портом в подставляемой ссылке, которая использует соединение по имени порта.
=> (стрелка)	Отделяет входы от выходов в операторах Truth Table. Отделяет WHEN предложения от булевых выражений в операторах Case.
+ (плюс)	Оператор сложения
- (минус)	Оператор вычитания
== (два знака равенства)	Оператор эквивалентности строк или чисел
! (восклицательный знак)	Оператор НЕ
!= (знак восклицание равно)	Оператор неравенства
> (больше чем)	Компаратор больше чем
>= (больше или равно)	Компаратор больше чем или равно
< (меньше чем)	Компаратор меньше чем
<= (меньше или равно)	Компаратор меньше чем или равно
& (амперсант)	Оператор И
!& (восклицание амперсант)	Оператор И-НЕ
$ (знак доллара)	Оператор Исключающее - ИЛИ
!$ (восклицание	Оператор Исключающее - ИЛИ - НЕ

доллар)	
# (знак фунта)	Оператор ИЛИ
!# (восклицание фунт)	Оператор ИЛИ-НЕ
? (вопрос)	Тернарный оператор. Он использует следующий формат: <выражение 1> ? < выражение 2> : < выражение 3> Если первое выражение не ноль (истина), то вычисляется второе выражение и результат возвращается тернарному выражению. В противном случае возвращается значение третьего выражения.

1.4.4 Строковые и символьные имена

В AHDL существует три типа имен:

A. Символьные имена являются идентификаторами, описываемыми пользователем. Они используются для объявления следующих частей TDF:

1. Внутренних и внешних узлов и шин
2. Констант
3. Переменных конечных автоматов, битов состояний и имен состояний
4. Экземпляров
5. Параметров
6. Сегментов памяти
7. Оценочных функций
8. Именованных операторов

B. Имена подпроектов - это имена, которые пользователь определил для файлов проекта более низкого уровня. Имя подпроекта должно совпадать с именем TDF файла.

C. Имена портов - это символьные имена, идентифицирующие входы или выходы логической функции.

Компилятор генерирует имена содержащие символ тильда (~), которые могут появляться в файле подгонки (Fit File) проекта. При использовании обратной аннотации эти имена появятся и в ACF файле проекта. Символ тильда зарезервирован только для имен, генерируемых компилятором и Вы не можете использовать его в Ваших собственных именах выводов, узлов и шин.

Для трех типов имен доступны два вида представления: с использованием кавычек и без них. Строковые имена заключаются в одиночные кавычки ('), а символьные имена без них.

Когда Вы создаете символ представления TDF файла, который содержит строковые имена портов, кавычки не включаются в его символ представления входов и выходов (pinstub).

1.4.5 Шины

Символьные имена и порты одного и того же типа можно объявить и использовать как шины в булевых выражениях и уравнениях.

Шина, которая может содержать до 256 членов (или битов), рассматривается как коллекция узлов и действует как одно целое.

Одиночные узлы и константы GND и VCC можно дублировать для создания шин.

Шины можно объявить с помощью следующих трех способов:

1. Имя шины состоит из символьного имени или имени порта, за которым следует указание поддиапазона, заключенного в скобки, т.е. a[4..1]. Имя вместе с самым длинным числом в диапазоне может содержать до 32 символов. Например,

Имя q[MAX..0] правильно, если константа MAX была описана выше в операторе Constant.

После определения шины скобки [] являются коротким способом описания всего диапазона. Например,

a [4..1] можно указать как a[].
b [6..0][3..2] можно указать как b[][].

2. Имя шины состоит из символьного имени или имени порта, за которым следует указание поддиапазонов, заключенных в скобки, т.е. d[6..0][2..0]. Имя вместе с самым длинным числом в диапазоне может содержать до 32 символов. К индивидуальному узлу в шине можно обратиться как name[y][z] или namey_z, где y и z числа в диапазоне шины.
3. Последовательное имя шины состоит из списка символьных имен, портов или чисел, разделенных запятыми и заключенных в скобки, например, (a, b, c).

Эта нотация полезна для определения имен портов. Например,

Входные порты переменной reg типа DFF можно записать как reg.(d, clk, clrn, prn).

Ниже приведены две совокупности примеров, демонстрирующие две шины, описанные с помощью различной нотации:

b[5..0]
(b5, b4, b3, b2, b1, b0)
b[]

b[log2(256)..1+2-1]
b[2^8..3 mod 1]
b[2*8..8 div 2]

1.4.5.1 Диапазоны и поддиапазоны шин

Диапазоны в именах шин могут состоять из чисел или арифметических выражений, разделенных двумя точками (..) и заключенных в скобки []. Например,

a[4..1]	шина с членами a4, a3, a2, и a1.
d[B"10"..B"00"]	шина с членами d2, d1, и d0.
b[2*2..2-1]	шина с членами b4, b3, b2, и b1. Ограничителями диапазона являются арифметические выражения.
q[MAX..0]	допустимая шина, если константа MAX была описана в операторе Constant.
c[MIN(a,b)..0]	допустимая шина, если оцениваемая функция MIN была описана в операторе Define.
t[WIDTH-1..0]	допустимая шина, если параметр WIDTH был описан в операторе Parameters.

Не зависимо от того является ли ограничитель диапазона числом или арифметическим выражением компилятор разделяет и интерпретирует ограничители как десятичные значения (целые числа).

Поддиапазоны содержат подмножество узлов, определенных в объявлении шины и могут описываться рядом способов. Запятые можно использовать как заменители только в шинах с левой стороны булева уравнения или подставляемой ссылки. Например,

Если Вы объявили шину c[5..1], то Вы можете использовать следующие поддиапазоны этой шины:

c[3..1]

c[4..2]
c4
c[5]
(c2, , c4)

В поддиапазоне (c2, , c4), запятая используется для сохранения места не назначенному члену шины.

Диапазоны обычно приводятся в убывающем порядке. Для указания диапазонов в возрастающем порядке или как в убывающем так и в возрастающем порядке Вы должны определить опцию BIT0 с помощью оператора Options для предотвращения выдачи предупреждающих сообщений компилятором. В шинах с двумя диапазонами эта опция воздействует на оба диапазона.

1.4.6 Числа в AHDL

Вы можете использовать десятичные, двоичные, восьмеричные и шестнадцатеричные числа в любых сочетаниях. Синтаксис для каждого основания показывается ниже.

Основание:	**Значения:**
Десятичное	<последовательность цифр от 0 до 9>
Двоичное	B"<последовательность 0-ей, 1-ц и X-ов>" (где X = "безразличное состояние")
Восьмеричное	O"<последовательность цифр от 0 до 7>" или Q"< последовательность цифр от 0 до 7>"
Шестнадцатеричное	X"< последовательность цифр от 0 до 9, A до F>" H"< последовательность цифр от 0 до 9, A до F >"

К числам применяются следующие правила:

1. Компилятор MAX+PLUS II всегда интерпретирует числа в булевых выражениях как группы двоичных цифр; числа в диапазонах шин как десятичные значения.
2. Числа нельзя присваивать одиночным узлам в булевых уравнениях. Вместо этого используйте VCC и GND.

1.4.7 Арифметические выражения

Арифметические выражения можно использовать для определения оцениваемых функций в операторах Define, констант в операторах Constant, значений параметров в операторах Parameters и в качестве ограничителей диапазонов шин. Например,

Диапазон, определенный с помощью арифметического выражения:

```
SUBDESIGN foo
(
 a[4..2+1-3+8] : INPUT;
)
```

Константа, определенная с помощью арифметического выражения:

```
CONSTANT foo = 1 + 2 DIV 3 + LOG2(256);
```

Оцениваемая функция, определенная с помощью арифметического выражения:

```
DEFINE MIN(a,b) = ((a < b) ? a : b);
```

Арифметические операторы и компараторы используются в этих выражениях для выполнения основных арифметических и сравнительных операций с числами в них. В арифметических выражениях используются следующие операторы и компараторы:

Оператор/ компаратор:	Пример:	Описание:	Приоритет:
+ (унарный)	+1	положительный	1
- (унарный)	-1	отрицательный	1
!	!a	NOT	1
^	a ^ 2	степень	1
MOD	4 MOD 2	модуль	2
DIV	4 DIV 2	деление	2
*	a * 2	умножение	2
LOG2	LOG2(4-3)	логарифм по основанию 2	2
+	1+1	сложение	3
-	1-1	вычитание	3
== (числовой)	5 == 5	числовое равенство	4
== (строковый)	"a" == "b"	строковое равенство	4
!=	5 != 4	не равно	4
>	5 > 4	больше чем	4
>=	5 >= 5	больше чем или равно	4
<	a < b+2	меньше чем	4
<=	a <= b+2	меньше чем или равно	4
&	a & b	AND	5
AND	a AND b		
!&	1 !& 0	NAND	5
NAND	1 NAND 0		
$	1 $ 1	XOR	6
XOR	1 XOR 1		
!$	1 !$ 1	XNOR	6
XNOR	1 XNOR 1		
#	a # b	OR	7
OR	a OR b		
!#	a !# b	NOR	7

NOR	a NOR b		
?	(5<4) ? 3:4	тернарный	8

Ко всем арифметическим выражениям применяются следующие правила:

1. Арифметические выражения должны давать не отрицательные числа.
2. Когда результат LOG2 не целый, он автоматически округляется до следующего целого. Например, LOG2(257) = 9.

Арифметические операторы, поддерживаемые в арифметических выражениях, являются надмножеством арифметических операторов, поддерживаемых в булевых выражениях, которые описываются в 1.4.9.2.

1.4.8 Встроенные оценочные функции

В AHDL встроены следующие предопределенные оценочные функции, которые не нужно определять в операторах Define:

- USED, которую можно использовать для контроля того, использовался ли порт, например, в операторе If Generate или Parameters. USED принимает имя порта в качестве входа и возвращает значение FALSE, если порт не используется.
- CEIL, которая возвращает наименьшее целое число большее вещественного числа. Хотя эта операция применима ко всем арифметическим выражениям, она имеет смысл только для LOG2 и DIV, в которых результат может быть вещественным.
- FLOOR, которая возвращает наибольшее целое число меньшее вещественного числа. Хотя эта операция применима ко всем арифметическим выражениям, она имеет смысл только для LOG2 и DIV, в которых результат может быть вещественным.

Примеры

CEIL(LOG2(255)) = 8
FLOOR(LOG2(255)) = 7
Использованный статус протестирован в операторе Assert:
USED(aconst) == # 0 USED(AVALUE)

1.4.9 Булевы выражения

Булевы выражения состоят из операндов, разделенных логическими и арифметическими операторами и компараторами и дополнительно сгруппированы с помощью круглых скобок. Выражения используются в булевых уравнениях также как и в других операторах таких как Case и If Then.

Булево выражение может быть одним из следующих:

1. Операнд
   Например, a, b[5..1], 7, VCC

2. Подставляемая ссылка на логическую функцию
   Например, out[15..0] = 16dmux(q[3..0]);

3. Префиксный оператор (! или -), применяемый к булеву выражению
   Например, !c

4. Два булевых выражения, разделенных бинарным оператором
   Например, d1 $ d3

5. Булево выражение, заключенное в круглые скобки
   Например, (!foo & bar)

Вы можете именовать булевы операторы и компараторы в файлах AHDL для облегчения ввода присваиваний ресурсов и для интерпретации раздела Equations в файле отчета. За дополнительной информацией обратитесь к 1.3.5.2

1.4.9.1 Логические операторы

В булевых выражениях можно использовать следующие логические операторы:

Оператор:	Пример:	Описание:
!	!tob	дополнение до 1
NOT	NOT tob	
&	bread & butter	И
AND	bread AND butter	
!&	a[3..1] !& b[5..3]	И-НЕ
NAND	a[3..1] NAND b[5..3]	
#	trick # treat	ИЛИ
OR	trick OR treat	
!#	c[8..5] !# d[7..4]	ИЛИ-НЕ
NOR	c[8..5] NOR d[7..4]	
$	foo $ bar	Исключающее ИЛИ
XOR	foo XOR bar	
!$	x2 !$ x4	Исключающее ИЛИ-НЕ
XNOR	x2 XNOR x4	

Каждый оператор представляет двухвходовый логический вентиль, за исключением оператора NOT (!), который является префиксом инвертирования одного узла. Вы можете использовать или имя или символ для представления логического оператора.

Выражения, которые используют эти операторы, интерпретируются различно в зависимости от того, являются ли операнды одиночными узлами, шинами или числами.

Вы можете позволить компилятору заменить И операторы и все компараторы в булевых выражениях на lpm_add_sub и lpm_compare функции, включая логическую опцию **Use LPM for AHDL Operators**.

1.4.9.1.1 Булевы операторы, использующие NOT

Оператор НЕ является префиксом инвертора. Поведение оператора НЕ зависит от операнда, на который он воздействует.

С оператором НЕ можно использовать три типа операндов:

1. Если операнд - одиночный узел, GND, или VCC, выполняется одиночная инверсия. Например, !a означает, что сигнал проходит через инвертор.
2. Если операнд - группа узлов, то каждый член группы проходит через инвертор. Например, шина !a[4..1] интерпретируется как (!a4, !a3, !a2, !a1).
3. Если операнд - число, он трактуется как двоичное число и каждый его бит инвертируется. Например, !9 интерпретируется как !B"1001", то есть B"0110".

1.4.9.1.2 Булевы операторы, использующие AND, NAND, OR, NOR, XOR, и XNOR

С бинарными операторами существует пять сочетаний операндов. Каждое из этих сочетаний интерпретируется различно:

1. Если оба операнда - одиночные узлы или константы GND и VCC, оператор выполняет логическую операцию над двумя элементами. Например, (a & b).
2. Если оба операнда - группы узлов, оператор действует на соответствующие узлы каждой группы, выполняя побитовые операции между группами. Группы должны иметь одинаковый размер. Например, (a, b, c) # (d, e, f) интерпретируется как (a # d, b # e, c # f).
3. Если один операнд - одиночный узел, GND, или VCC, а другой группа узлов, одиночный узел или константа дублируется для создания группы такого же размера как другой оператор. Затем выражение трактуется как групповая операция. Например, a & b[4..1] интерпретируется как (a & b4, a & b3, a & b2, a & b1).

4. Если оба операнда - числа, то более короткое число расширяется с учетом знака для согласования с размером другого числа и трактуется затем как групповая операция. Например, в выражении (3 # 8), 3 и 8 преобразуются в двоичные числа B"0011" и B"1000", соответственно. Результатом будет B"1011".
5. Если один операнд - число, а другой узел или группа узлов, то число разделяется на биты для согласования с размером группы и выражение рассматривается как групповая операция. Например, в выражении (a, b, c) & 1, 1 преобразуется к B"001" и выражение становится (a, b, c) & (0, 0, 1). Результатом будет (a & 0, b & 0, c & 1).

Выражение, которое использует VCC как операнд, интерпретируется в зависимости от выражения, которое использует 1 как операнд. Например, в первом выражении, 1 - число в знакорасширенном формате . Во втором выражении, узел VCC дублируется . Затем каждое выражение трактуется как групповая операция.

(a, b, c) & 1 = (0, 0, c)
(a, b, c) & VCC = (a, b, c)

1.4.9.2 Арифметические операторы в булевых выражения

Арифметические операторы используются для арифметических операций сложения и вычитания над числами и шинами в булевых выражениях. В них используются следующие операторы:

Оператор:	Пример:	Описание:
+ (унарный)	+1	плюс
- (унарный)	-a[4..1]	минус
+	count[7..0] + delta[7..0]	сложение
-	rightmost_x[] - leftmost_x[]	вычитание

К бинарным операторам применимы следующие правила:

- Операции выполняются между двумя операндами, которые должны быть шинами или числами.
- Если оба операнда - шины, то они должны иметь один размер.
- Если оба операнда числа, более короткое число расширяется до размеров другого операнда.
- Если один оператор - число, а другой группа узлов, то число усекается или расширяется для согласования размеров операндов. Если отбрасываются любые значимые биты, то компилятор MAX+PLUS II выдает сообщение об ошибке.

⇒ Когда Вы складываете две шины вместе с правой стороны булева уравнения с помощью оператора +, Вы можете поместить 0 с левой стороны группы для расширения ширины шины. Этот метод обеспечивает добавление дополнительного бита данных с левой стороны уравнения, который можно использовать как сигнал выходного переноса. Например, шины count[7..0] и delta[7..0] дополняются нулями для обеспечения информацией сигнала cout:

(cout, answer[7..0]) = (0, count[7..0]) + (0, delta[7..0])

1.4.9.3 Компараторы

Для сравнения одиночных узлов или шин используются два типа компараторов: логические и арифметические. В булевых выражениях можно использовать следующие компараторы.

Компаратор:	Пример:	Описание
== (логический)	addr[19..4] ==	равно

	B"B800"	
!= (логический)	b1 != b3	не равно
< (арифметический)	fame[] < power[]	меньше чем
<= (арифметический)	money[] <= power[]	меньше чем или равно
> (арифметический)	love[] > money[]	больше чем
>= (арифметический)	delta[] >= 0	больше чем или равно

Логические компараторы могут сравнивать одиночные узлы, шины и числа без неопределенных (X) значений. При сравнении шин или чисел, шины должны иметь одинаковый размер. Компилятор MAX+PLUS II выполняет побитовое сравнение шин, возвращая VCC, когда сравнение истинно, и GND, когда сравнение ложно.

Арифметические компараторы могут сравнивать только шины и числа; шины должны иметь одинаковый размер. Компилятор выполняет беззнаковое сравнение значений шин, т.е., каждая шина интерпретируется как положительное двоичное число и сравнивается с другой шиной.

1.4.9.4 Приоритеты булевых операторов и компараторов

Операнды, разделенные логическими и арифметическими операторами и компараторами вычисляются в соответствии с правилами приоритетов, приведенными ниже (приоритет 1 - наивысший). Операции одинакового приоритета оцениваются слева направо. С помощью скобок () можно менять порядок вычислений.

Приоритет:	Оператор/Компаратор:
1	- (минус)
1	! (НЕ)
2	+ (сложение)

2	- (вычитание)
3	== (равно)
3	!= (не равно)
3	< (меньше чем)
3	<= (меньше чем или равно)
3	> (больше чем)
3	>= (больше чем или равно)
4	& (И)
4	!& (И-НЕ)
5	$ (Исключающее ИЛИ)
5	!$ (Исключающее ИЛИ-НЕ)
6	# (ИЛИ)
6	!# (ИЛИ-НЕ)

1.4.10 Логические функции

1.4.10.1 Мегафункции/LPM

MAX+PLUS II предлагает большое разнообразие *мегафункций*, включая *LPM* функции а также *параметризуемые функции*. Ниже приводится список мегафункций.

Вентили	
lpm_and	*lpm_inv*
lpm_bustri	*lpm_mux*
lpm_clshift	*lpm_or*
lpm_constant	*lpm_xor*
lpm_decode	*mux*
busmux	

Арифметические компоненты	
lpm_abs	*lpm_counter*
lpm_add_sub	*lpm_mult*
lpm_compare	

Запоминающие компоненты	
csfifo	*lpm_ram_dq*
csdpram	*lpm_ram_io*
lpm_ff	*lpm_rom*
lpm_latch	*lpm_dff*
lpm_shiftreg	*lpm_tff*

Другие функции	
clklock	*pll*
ntsc	

Функции *Мегаядра*	
a16450	*a8255*
a6402	*fft*
a6850	*rgb2ycrcb*
a8237	*ycrcb2rgb*
a8251	

Мегафункция - сложный или высокоуровневый строительный блок, который можно использовать совместно с примитивами вентилей и триггеров и/или с макрофункциями старого типа в файлах проекта.

Altera поставляет библиотеку мегафункций, включая функции из библиотеки параметризуемых модулей (LPM) версии 2.1.0, в директории \maxplus2\max2lib\mega_lpm, созданной во время инсталляции.

Для просмотра файла, содержащего логику мегафункции, укажите символ мегафункции в графическом редакторе или ее имя в текстовом редакторе и выберите **Hierarchy Down** (меню File).

Библиотека параметризуемых функций (LPM) - технологически-независимая библиотека логических функций, параметризуемая для достижения масштабируемости и адаптируемости. Altera реализовала параметризуемые модули (называемые также параметризуемые функции) из LPM в версии 2.1.0, которые предлагают архитектурно-независимый ввод проекта для всех, поддерживаемых MAX+PLUS II устройств. Компилятор включает встроенную поддержку компиляции LPM для функций, используемых во входных файлах (схемном, AHDL, VHDL, и EDIF).

Параметризуемая функция - логическая функция, использующая параметры для достижения масштабируемости, адаптируемости и эффективной реализации в кремнии.

Мегафункции Мегаядра - предварительно проверенные HDL файлы для сложных функций системного уровня, которые можно приобрести у Altera. Они оптимизированы под архитектуры FLEX 10K, FLEX 8000, FLEX 6000, MAX 9000, и MAX 7000 устройств. Мегафункции Мегаядра состоят из нескольких файлов. Файл для последующего синтеза используется для реализации проекта (подгонки) в заданном устройстве. Кроме этого прилагаются VHDL или Verilog HDL функциональные модели для проектирования и отладки со стандартными EDA средствами моделирования.

Altera поставляет библиотеку мегафункций, включая любые приобретаемые мегафункции Мегаядра в директории \maxplus2\max2lib\mega_lpm, созданной во время инсталляции.

Если Ваш код доступа для мегафункции Мегаядра содержит разрешение просмотра источника файла проекта, Вы можете просмотреть его, указывая символ мегафункции в графическом редакторе или имя в текстовом редакторе и выбирая **Hierarchy Down** (меню File).

Ниже приводится описание наиболее часто применяемых мегафункций. Полные сведения по всем мегафункциям можно найти в системе помощи (меню Help, команда Megafunctions/LPM).

lpm_and (вентиль И)

Altera рекомендует использовать примитивы вентилей И или их операторы вместо lpm_and для более легкой реализации и улучшения времени компиляции. Тем не менее lpm_and могут быть полезны при необходимости иметь параметризуемые входы.

Прототип функции

```
FUNCTION lpm_and
        (data[LPM_SIZE-1..0][LPM_WIDTH-1..0])
  WITH (LPM_WIDTH, LPM_SIZE)
  RETURNS (result[LPM_WIDTH-1..0])
```

Порты :

ВХОДЫ

Имя порта	**Необходим**	**Описание**	**Комментарии**
data[][]	Да	Вход данных в вентиль И	Размер порта LPM_SIZE x LPM_WIDTH

ВЫХОДЫ

Имя порта	**Необходим**	**Описание**	**Комментарии**
result[]	Да	Побитовое И.	Размер порта LPM_WIDTH.

Параметры :

Параметр	Тип	Необходим	Описание
LPM_WIDTH	Целый	Да	Ширина портов data[][] и result[]. Количество AND вентилей.
LPM_SIZE	Целый	Да	Количество входов в каждый AND вентиль. Количество входных шин.

Каждый вентиль И имеет следующую функцию:

Входы	Выходы
data[LPM_SIZE-1]_[LPM_WIDTH-1]	result[LPM_WIDTH-1]
0XXX…	0
X0XX…	0
XX0X…	0
…	…
1111…	1

Используемый ресурс:

Простые вентили lpm_and используют приблизительно одну логическую ячейку на вентиль.

1.4.10.2 Макрофункции

MAX+PLUS II предлагает свыше 300 макрофункций.

Имена шинных макрофункций оканчиваются на букву B. Они функционально идентичны с соответствующими не шинными макрофункциями, но имеют сгруппированные входные и/или выходные выводы.

Для просмотра схемы или AHDL файла, который содержит логику макрофункции, укажите символ макрофункции в графическом редакторе или имя макрофункции в текстовом редакторе и выберите Hierarchy Down (меню File).

Категории макрофункций:

Сумматоры	Защелки
АЛУ	Умножители
Буферы	Мультиплексоры
Компараторы	Генераторы четности
Конвертеры	Быстрые умножители
Счетчики	Регистры
Декодеры	Сдвиговые регистры
Цифровые фильтры	Регистры хранения
EDAC	SSI функции
Шифраторы	Элементы ввода/вывода
Делители частоты	

1.4.10.3 Примитивы

MAX+PLUS II обеспечивает большое многообразие примитивных функций для разработки схем. Так как AHDL и VHDL логические операторы, порты и некоторые операторы замещают примитивы в AHDL и VHDL файлах, то примитивы являются подмножеством их, доступных для GDF файлов, как показано ниже.

Примитив - один из основных функциональных блоков, применяющийся для проектирования схем с помощью программы MAX+PLUS II. Примитивы используются в графических файлах (.gdf), текстовых файлах (.tdf), и VHDL файлах (.vhd).

Символы примитивов для графического редактора поставляются в директории \maxplus2\max2lib\prim, созданной во время инсталляции.

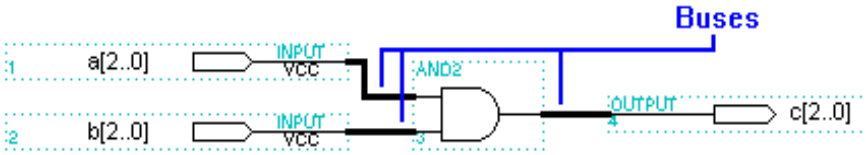
Прототипы функций встроены в программу MAX+PLUS II.

Примитивные массивы

Примитивный массив - это примитив, который представляет несколько идентичных примитивов. Вы можете использовать примитивные массивы для создания более компактных GDF или OrCAD Schematic файлов путем ввода единственного примитива, который экстрактор списков связей компилятора переведет в несколько примитивов.

Вы можете создать примитивный массив двумя способами:

- Если все порты символа (*pinstub*) примитива соединяются с шинами, состоящими из n членов, примитив переводится в массив n индивидуальных примитивов. Каждый индивидуальный узел шины соединяется с соответствующим портом символа каждого индивидуального примитива в массиве. Например,



В этом примере примитивный массив создается при соединении трех шин A[0..2], B[0..2], и C[0..2] с двумя выводами INPUT, выводом OUTPUT и вентилем AND2.

Во время обработки компилятор переводит этот примитивный массив в 6 выводов INPUT, 3 вывода OUTPUT и 3 вентиля AND2 следующим образом:

Один AND2 вентиль соединяется с узлами A0, B0, и C0.

Один AND2 вентиль соединяется с узлами A1, B1, и C1.

Один AND2 вентиль соединяется с узлами A2, B2, и C2.

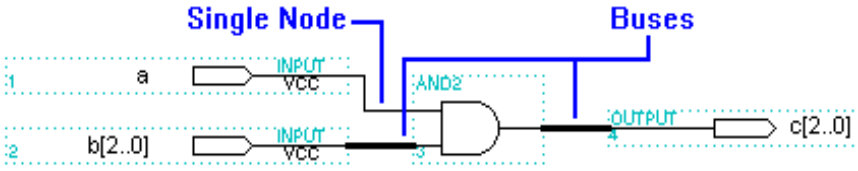
Входные выводы A0, A1, и A2 соединяются с узлами A0, A1, и A2, соответственно.

Входные выводы B0, B1, и B2 соединяются с узлами B0, B1, и B2, соответственно.

Выходные выводы C0, C1, и C2 соединяются с узлами C0, C1, и C2, соответственно.

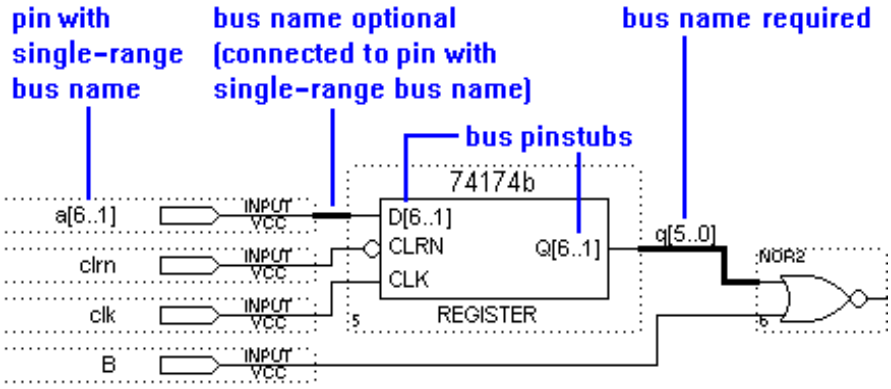
Примитивы выводов INPUT, INPUTC, OUTPUT, OUTPUTC, BIDIR, или BIDIRC, которым даны шинные имена переводятся в массив примитивов. Вы не можете использовать перечислимое имя шины для обозначения примитива вывода.

- Если некоторые порты символа примитива соединяются с шинами с n членами, а некоторые с одиночными узлами, примитив переводится в массив n примитивов. В этом случае каждый индивидуальный узел шины соединяется с соответствующим портов символа каждого примитива в массиве, а каждый узел, который не является частью шины, соединяется с тем же самым портом символа каждого примитива. Например,



Вы должны обозначить все узлы и шины, которые используются для создания примитивного массива, за исключением следующих случаев, где имена узлов и шин не обязательны:

- Одиночный узел, который соединяется с примитивным массивом.
- Шинный проводник, который соединяется с примитивным массивом, если не меньше одного сегмента сети, которая содержит этот проводник, явно обозначено перед любыми точками соединения или, если сеть соединяется с выводом с шинным именем. Например,



Вы не можете вводить присваивания для *проб* и ресурсов символов примитивов, которые используются для создания примитивных массивов. Обратитесь в раздел Принципы работы с присваиваниями.

Проба - уникальное имя, связанное с любым узлом, например, вход или выход примитива, мегафункции или макрофункции, которое можно использовать вместо полного иерархического имени узла в любом месте MAX+PLUS II. Таким образом пробное имя обеспечивает быструю идентификацию узла.

Не используемые входы примитивов, мегафункций и макрофункций

К не подсоединенным входным портам примитива, мегафункции, символов макрофункций и экземпляров применяются следующие правила.

- Не используемые входы примитивов триггеров имеют следующие значения по умолчанию:

  CLRN: VCC (неактивный)
  PRN: VCC (неактивный)
  ENA: VCC (активный)

- Требуются входы data и Clock в триггеры и входы data и ENA в защелки.
- Неиспользуемый вход OE буфера TRI по умолчанию имеет значение VCC (активный).
- В файлах графического редактора неиспользуемые входы логических примитивов должны соединяться с VCC или GND.
- Логические уровни по умолчанию для неиспользуемых входов макрофункций документированы в Помощи для каждой макрофункции.
- В отличии от макрофункций, мегафункции могут не иметь значений по умолчанию для входов в некоторые порты и неудачное соединение таких портов приведет к выдачи компилятором сообщений об ошибке.

Примитивы буферов

CARRY	OPNDRN
CASCADE	SOFT
EXP	TRI
GLOBAL (SCLK)	WIRE (только GDF)
LCELL (MCELL)	

Примитивы триггеров и защелок

DFF	SRFF
DFFE	SRFFE
JKFF	TFF
JKFFE	TFFE
LATCH	

Примитивы/Порты входов и выходов

BIDIR или INOUT	BIDIRC (только GDF)
INPUT или IN	INPUTC (только GDF)
OUTPUT или OUT	OUTPUTC(толькоGDF)

Логические примитивы

AND	NOR
BAND (только GDF)	NOT
BNAND (только GDF)	OR
BNOR (только GDF)	VCC (только GDF)
BOR (только GDF)	XNOR
GND (только GDF)	XOR
NAND	

Другие примитивы (только GDF)

CONSTANT
PARAM
Title Block

Прототипы функций для примитивов в TDF файлах не нужны. Однако Вы можете переопределить порядок вызова входов примитива, вводя оператор Function Prototype в Ваш TDF.

1.4.11 Порты

Порт - это вход или выход логической функции. Порт может находится в двух местах:

- Порт, который является входом или выходом текущего файла, объявляется в разделе Subdesign.
- Порт, который является входом или выходом экземпляра примитива или файла разработки более низкого уровня, используется в разделе Logic.

Порты текущего файла

Порт, который является входом или выходом текущего файла объявляется в следующем формате в разделе Subdesign:

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

Доступны следующие типы портов:

INPUT	MACHINE INPUT
OUTPUT	MACHINE OUTPUT
BIDIR	

Когда текстовой файл проекта является старшим в иерархии, имя порта синонимично с именем вывода. Дополнительное значение порта по умолчанию, которое может быть или VCC или GND, можно определить для типов портов INPUT и BIDIR. Это значение используется только если слева порт не подсоединен, когда экземпляр TDF применяется в файле разработки более высокого уровня.

Например:

```
SUBDESIGN top
(
foo, bar, clk1, clk2, c[4..0][6..0]      : INPUT = VCC;
a0, a1, a2, a3, a4            : OUTPUT;
b[7..0]                                  : BIDIR;
)
```

Вы можете импортировать и экспортировать конечные автоматы между TDF и другими файлами разработки, описывая входы и выходы как MACHINE INPUT или MACHINE OUTPUT в разделе Subdesign. Прототип функции, который представляет файл, должен указывать, какие порты принадлежат конечному автомату. MACHINE INPUT и MACHINE OUTPUT можно использовать только в файлах более низкого уровня в иерархии проекта.

Порты экземпляров

Порт, который является входом или выходом экземпляра логической функции присоединяется в разделе Logic. Для соединения логической функции с другими частями TDF, Вы вставляете экземпляр функции с помощью подставляемой ссылки, объявления Instance или конечного автомата с помощью State Machine и затем используете порты функции в разделе Logic.

Если Вы используете подставляемую ссылку с присваиванием по положению порта для создания экземпляра логической функции, важен порядок портов, а не имена. Порядок портов определяется в прототипе функции.

Если Вы используете объявление Instance или подставляемую ссылку со связью по имени для создания экземпляра логической функции, важны имена портов, а не их порядок.

В следующем примере D триггер объявляется как переменная reg в разделе Variable, а затем используется в разделе Logic:

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

Имена портов используются в следующем формате в разделе Logic:

<имя экземпляра>.<имя порта>

<имя экземпляра> - это имя функции, данное пользователем. <имя порта> идентично с именем порта, который объявляется как вход или выход файла в разделе Subdesign TDF файла более низкого уровня или имя вывода в файле разработки другого типа. <имя порта> синонимично с именем порта символа (pinstub), который представляет экземпляр файла разработки в GDF.

Все функции, поставляемые Altera, имеют предопределенные имена портов (pinstub), которые показываются в прототипе функции. Наиболее используемые имена портов примитивов показаны в следующей таблице:

Имя порта	Описание
.q	Выход триггера или защелки
.d	Вход данных триггера или защелки
.t	Вход T триггера
.j	J вход JK триггера
.k	K вход JK триггера
.s	Вход установки SR триггера

.r Вход очистки SR триггера

.clk Тактовый вход триггера

.ena Вход разрешения тактирования триггера, разрешения фиксации защелки разрешения конечного автомата

.prn Активный низкий вход предустановки триггера

.clrn Активный низкий вход очистки триггера

.reset Активный высокий вход сброса конечного автомата

.oe Вход разрешения выхода TRI примитива

.in Первичный вход CARRY, CASCADE, EXP, TRI, OPNDRN, SOFT, GLOBAL, и LCELL примитивов

.out Выход TRI, OPNDRN, SOFT, GLOBAL, и LCELL примитивов

1.5 Структура проекта

Этот раздел описывает структуру проекта на языке AHDL. Разделы и операторы языка AHDL описываются в том порядке, в котором они следуют в текстовом файле проекта (TDF - Text Design File).

- Обзор
- Оператор Title
- Оператор Parameters
- Оператор Include
- Оператор Constant
- Оператор Define
- Оператор Function Prototype
- Оператор Options
- Оператор Assert
- Раздел Subdesign
- Раздел Variable

Раздел Logic

1.5.1 Обзор

Текстовый файл проекта на языке AHDL должен содержать, как минимум, два раздела: Subdesign и Logic. Все остальные разделы и операторы являются необязательными. В предлагаемом к ознакомлению разделе ‘Структура проекта’ информация об операторах и разделах языка AHDL дается в том порядке, в котором они следуют в текстовом файле проекта (TDF - Text Design File).

1.5.2 Оператор Title

Оператор Title позволяет внести в текстовый файл проекта комментарий, который в дальнейшем будет помещен в файл отчета (Report File),генерируемый компилятором. Следующий пример демонстрирует использование оператора Title:

```
TITLE "Display Controller";
```

При использовании оператора Title необходимо соблюдать следующие правила:

- Оператор Title начинается с ключевого слова TITLE, за которым следует текстовая строка -заголовок, заключенная в двойные кавычки. Оператор заканчивается символом ‘;’ (точка с запятой).
- Если оператор Title используется в текстовом файле проекта, то использованный заголовок помещается в начало файла отчета (Report File). В показанном выше примере, заголовок Display Controller помещается в файл отчета.
- Заголовок может содержать до 255 символов, кроме того в нем не должны использоваться символы конца строки (end-of-line) и конца файла (end-of-file). Для использования кавычек в заголовке необходимо использовать пары двойных кавычек. Пример:

```
TITLE """EPM5130"" Display Controller";
```

- В одном текстовом файле проекта может использоваться не более одного оператора Title.
- Оператор Title должен быть расположен за пределами других разделов языка AHDL.

1.5.3 Оператор Parameters

Оператор Parameters позволяет определять один и более параметров, управляющих *экземпляром (an instance)* параметрической мега- или макрофункции. Следующий пример демонстрирует использование оператора Parameters:

```
PARAMETERS
(
  FILENAME = "myfile.mif", -- optional default value follows "=" sign
  WIDTH,
  AD_WIDTH = 8,
  NUMWORDS = 2^AD_WIDTH
);
```

При использовании оператора Parameters необходимо соблюдать следующие правила:

- Оператор Parameters начинается с ключевого слова PARAMETERS, за которым следует список из одного или более параметров и необязательных значений по умолчанию. Весь список заключается в круглые скобки.
- Параметры в списке отделяются друг от друга запятыми; имена параметров отделяются от необязательных значений по умолчанию символом (=). В примере, показанном выше, только параметр WIDTH не имеет предопределенного значения.
- Имена параметров могут представлять собой либо имена, определенные пользователем, либо имена, предопределенные фирмой Altera .
- Значения параметров могут представлять собой текстовые строки, заключенные в двойные кавычки. В том случае, если значения параметров не заключены в двойные кавычки, компилятор пытается интерпретировать их как арифметические выражения; если это не удается, они интерпретируются как строки.
- Оператор Parameters заканчивается символом (;).
- После того, как параметр был определен, он может использоваться во всем текстовом файле проекта.

- Параметр может быть использован лишь после того, как он был определен.
- Имена параметров должны быть уникальными.
- Имя параметра не должно содержать пробелов. Для разделения слов и лучшего восприятия необходимо пользоваться символом подчеркивания.
- Оператор Parameters может использоваться произвольное количество раз в рамках одного текстового файла проекта.
- Оператор Parameters должен быть расположен за пределами других разделов языка AHDL.
- Параметры, используемые для определения других параметров, должны быть определены ранее.
- Использование круговых ссылок недопустимо. Следующий пример демонстрирует использование недопустимой круговой ссылки:

```
PARAMETERS
(
  FOO = BAR;
  BAR = FOO;
);
```

На этапе компиляции текстового файла проекта, компилятор осуществляет поиск значений параметров в следующей последовательности:

1. Производится анализ *экземпляра (an instance)* логической функции. Например, в текстовом файле проекта, в *объекте (an instance),* созданном путем *объявления объекта (Instance Declaration)* или *подставляемой ссылкой (in-line reference),* можно определить те параметры, которые будут использоваться, а также в необязательном порядке определить их значения. В графическом файле проекта (GDF - Graphic Design File) можно выбрать символ и, используя команду **Edit Ports/Parameters** из меню **Symbol,** присвоить значения параметров для этого объекта.

2. Производится анализ *экземпляра* логической функции более высокого уровня иерархии. Значения параметров *экземпляра* логической функции более высокого уровня иерархии распространяются на подфункции данной логической функции, если *экземпляры* этих логических подфункций не имеют своих значений для данных параметров.
3. Производится анализ глобальных значений параметров проекта по умолчанию, определенных командой **Global Project Parameters** из меню **Assign.** Эти значения хранятся в файле установок и конфигурации (Assignment&Configuration file - .acf) проекта.
4. Просматриваются необязательные значения по умолчанию, указываемые в разделе Parameters текстового файла проекта (TDF), или с помощью примитива PARAM в графическом файле проекта, описывающем логическую функцию. Эти значения по умолчанию используются только в том файле, в котором они приводятся и не распространяются на подпроекты, входящие в данный проект.

1.5.4 Оператор Include

Оператор Include позволяет импортировать текст из файла с расширением .inc в текущий файл. Следующий пример демонстрирует использование оператора Include:

INCLUDE "const.inc";

Оператор Include имеет следующие характеристики:

- Оператор Include начинается с ключевого слова INCLUDE, за которым следует имя подключаемого .inc-файла, заключенного в двойные кавычки.
- Если явно не указывать расширение подключаемого файла, то компилятор по умолчанию предполагает, что файл имеет расширение .inc.
- Оператор Include заканчивается символом (;).
- На этапе компиляции осуществляется замена оператора Include содержимым .inc-файла. В примере, показанном выше, файл **const.inc** заменяет текст INCLUDE “const.inc”;

Оператор Include часто используется для подключения прототипов функций для файлов более низкого уровня иерархии по отношению к данному текстовому файлу проекта (TDF). Для использования мега- и макрофункций необходимо сначала определить их логику функционирования в соответствующем файле проекта. Затем необходимо использовать оператор Function Prototype для определения портов функции. В качестве альтернативного варианта, можно использовать оператор Include для подключения прототипа функции, хранящегося в соответствующем файле с расширением .inc. Затем можно осуществить *объявление объекта (Instance Declaration)* или *подставляемую ссылку (in-line reference)* для экземпляра логической функции.

Можно автоматически создать файл с расширением .inc, содержащий прототип функции для текущего файла проекта, с помощью команды **Create Default Include File** меню File.

На этапе компиляции текстового файла проекта, компилятор осуществляет поиск файлов с расширением .inc в следующей последовательности:

1. Сначала осуществляется поиск в директории данного проекта
2. Просматриваются пользовательские библиотеки указанные командой **User Libraries** меню **Options.**
3. Просматриваются директории **\maxplus2\max2lib\mega_lpm** и **\maxplus2\max2inc**, созданные во время инсталляции.

После изменений, внесенных в текстовый файл проекта (TDF), в котором осуществляется подключение файлов с расширением .inc, можно использовать команду **Project Save&Check** меню **File** или осуществить полную перекомпиляцию проекта для обновления дерева иерархии проекта, выводимого в окне отображения иерархии проекта.

При использовании оператора Include необходимо соблюдать следующие правила:

- Имя файла, приведенного в операторе Include, не должно содержать пути.

- В программном обеспечении рабочих станций имена файлов контекстно-зависимы. В документации MAX+PLUSII имена файлов могут приводиться как с использованием прописных, так и строчных букв. Однако в случае использования оператора Include имена файлов должны в точности повторять их оригинальные имена. Названия макро- и мегафункций поставляемых фирмой Altera целиком состоят из строчных букв.
- Оператор Include должен быть расположен за пределами других разделов языка AHDL.
- Оператор Include может использоваться произвольное количество раз в рамках одного текстового файла проекта (TDF).

Файлы с расширением .inc должны удовлетворять следующим соглашениям:

- Полные имена этих файлов должны иметь расширение .inc.
- Файлы с расширением .inc могут содержать лишь следующие операторы:
  - Function Prototype
  - Define
  - Parameters
  - Constant

Вложенность при использовании файлов с расширением .inc недопустима.

Файлы с расширением .inc не должны содержать секцию Subdesign.

1.5.5 Оператор Constant

Оператор Constant позволяет ввести в применение информативное символическое имя для числа или арифметического выражения. Следующие примеры демонстрируют использование оператора Constant:

```
CONSTANT UPPER_LIMIT = 130;
CONSTANT BAR = 1 + 2 DIV 3 + LOG2(256);
CONSTANT FOO = 1;
CONSTANT FOO_PLUS_ONE = FOO + 1;
```

Оператор Constant имеет следующие характеристики:

- Оператор Constant начинается с ключевого слова CONSTANT, за которым следует символическое имя, затем символ (=) и далее число (при необходимости, включая его основание) или арифметическое выражение.
- Оператор Constant заканчивается символом (;).
- После того, как константа была определена, она может быть использована в пределах всего текстового файла проекта (TDF). В примере, приведенном выше, в разделе Logic можно использовать константу UPPER_LIMIT для представления десятичного числа 130.
- Константы могут быть определены посредством арифметических выражений. В эти арифметические выражения могут входить константы определенные ранее.
- Компилятор вычисляет арифметические выражения, используемые в операторе Constant и упрощает их до числовых значений. При этом не производится генерация логических схем.

При использовании оператора Constant необходимо соблюдать следующие правила:

- Константа может быть использована лишь после того, как она определена.
- Имена констант должны быть уникальными.
- Имя константы не должно содержать пробелов. Для разделения слов в имени константы и улучшения восприятия имен констант следует пользоваться символом подчеркивания.
- Оператор Constant может использоваться произвольное количество раз в рамках одного текстового файла проекта.
- Оператор Constant должен быть расположен за пределами других разделов языка AHDL.
- Константы, используемые для определения других констант, должны быть определены ранее.
- Использование циклических ссылок недопустимо. Следующий пример демонстрирует использование недопустимой циклической ссылки:

```
CONSTANT FOO = BAR;
CONSTANT BAR = FOO;
```

1.5.6 Оператор Define

Оператор Define позволяет определить *оценочную функцию (evaluated function)*, представляющую собой математическую функцию, возвращающую значение, вычисленное на основе необязательных входных аргументов.

В следующем примере описывается оценочная функция MAX, предопределяющая существование по меньшей мере одного порта в разделе Subdesign:

```
DEFINE MAX(a,b) = (a > b) ? a : b;
SUBDESIGN
(
  dataa[MAX(WIDTH,0)..0]: INPUT;
  datab[MAX(WIDTH,0)..0]: OUTPUT;
)
BEGIN
  datab[] = dataa[];
END;
```

Оператор Define имеет следующие характеристики:

- Оператор Define начинается с ключевого слова DEFINE, за которым следует символическое имя и список из одного или более аргументов, заключенных в круглые скобки.
- Аргументы отделяются друг от друга запятыми. Символ (=) отделяет список аргументов от арифметического выражения

  ⇒ При отсутствии аргументов оценочная функция эквивалентна константе.

  ⇒ Компилятор производит вычисления арифметических выражений приведенных в операторе Define и упрощает их до числовых значений. Генерации логических схем при этом не производится.

- Оператор заканчивается символом (;).

- Один раз определенная оценочная функция может использоваться затем в пределах всего текстового файла проекта (TDF).
- Для определения оценочных функций могут использоваться ранее определенные оценочные функции. Например, приведенная ниже оценочная функция MIN_ARRAY_BOUND вычисляется на основе значения оценочной функции MAX:

```
DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE MIN_ARRAY_BOUND(x) = MAX(0, x) + 1;
```

При использовании оператора Define необходимо соблюдать следующие правила:

- Оценочная функция может быть использована только после того как она была определена.
- Имена оценочных функций должны быть уникальными.
- Имена оценочных функций не должны содержать пробелов. Для разделения слов в имени оценочной функции и улучшения ее восприятия следует пользоваться символом подчеркивания.
- Оператор Define может использоваться произвольное количество раз в рамках одного текстового файла проекта.
- Оператор Define должен быть расположен за пределами других разделов языка AHDL.

1.5.7 Оператор Function Prototype.

Операторы Function Prototype имеют ту же функцию, что и символы в графических файлах проектов. И те и другие представляют собой краткое описание функции, описывая ее имя, а также входные, выходные и двунаправленные порты. Для функций, импортирующих и экспортирующих конечные автоматы, могут использоваться порты автомата.

Входные порты мега- и макрофункций не имеют значений по умолчанию, как это имеет место в файлах графического редактора MAX+PLUSII. Поэтому входные значения неиспользуемых портов должны быть указаны явно. Кроме того в секции Subdesign могут быть указаны значения по умолчанию для двунаправленных портов. Заметим, что для выходных портов нельзя определить значения по умолчанию.

Перед созданием *объекта (an instance)* мега- или макрофункции необходимо убедиться в существовании соответствующего ей файла проекта, описывающего ее логическое функционирование. Затем с помощью оператора Function Prototype описываются порты функции и создается *экземпляр (an instance)* логической функции путем *объявления объекта (Instance Declaration)* или *подставляемой ссылки (in-line reference),*

Следующие примеры демонстрируют использование операторов Function Prototype. Первый пример демонстрирует описание параметризируемой функции, а второй - не параметризируемой функции:

```
FUNCTION lpm_add_sub (cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-1..0],
add_sub)
  WITH (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ADDERTYPE,
    ONE_INPUT_IS_CONSTANT)
  RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
FUNCTION compare (a[3..0], b[3..0])
        RETURNS (less, equal, greater);
```

Оператор Function Prototype имеет следующие характеристики:

- За ключевым словом FUNCTION следует имя функции. В примерах показанных выше использованы имена функций lpm_add_sub и compare.
- За именем функции следует список входных портов. В первом примере, показанном выше, входными портами являются cin, dataa[LPM_WIDTH-1..0] и datab[LPM_WIDTH-1..0]; во втором примере входными портами являются a3,a2,a1,a0,b3,b2,b1 и b0.

- В параметризируемой функции за списком параметров следует ключевое слово WIDTH и список имен параметров. Список заключен в круглые скобки; имена отделены друг от друга запятыми.
- За списком выходных и двунаправленных портов функции следует ключевое слово RETURNS.В первом примере, показанном выше, выходными портами являются result[LPM_WIDTH-1..0], count и overflow ;во втором примере - less, equal и greater.
- Список входных и выходных портов заключается в круглые скобки. Имена отделяются друг от друга запятыми.
- При импортировании и экспортировании конечных автоматов, используемый файлом оператор Function Prototype должен использовать автоматный порт (определяемый ключевым словом MACHINE) для указания того, какие входы и выходы являются конечными автоматами. Пример:

```
FUNCTION ss_def (clock, reset, count)
  RETURNS (MACHINE ss_out);
```

- Оператор Function Prototype заканчивается символом (;).
- Оператор Function Prototype должен быть расположен за пределами других разделов языка AHDL и кроме того он должен располагаться до *экземпляра* логической функции созданной путем *объявления объекта* или *подставляемой ссылки (in-line reference)*.

Для *экземпляра примитива* также следует использовать механизм *объявления объекта (Instance Declaration)* или *подставляемую ссылку (in-line reference)*. Однако, в отличие от мега- и макрофункций логика функционирования примитива предопределена, таким образом нет необходимости определять логику функционирования примитива в отдельном файле проекта. Кроме того нет необходимости использовать оператор Function Prototype, за исключением тех случаев, когда нужно изменить порядок следования портов примитива.

Следующий пример демонстрирует прототип функции, существующий по умолчанию для примитива JKFF:

```
FUNCTION JKFF (j, k, clk, clrn, prn)
```

```
    RETURNS (q);
```

Данный пример показывает модифицированный прототип функции для примитива JKFF:

```
FUNCTION JKFF (k, j, clk, clrn, prn)
    RETURNS (q);
```

Альтернативой использования оператора Function Prototype в файле проекта является применение оператора Include для подключения файлов с расширением .inc, содержащих прототипы используемых функций. Кроме того MAX+PLUSII имеет в своем составе команду **Create Default Include File** в меню File, которая автоматически создает файл с расширением .inc, содержащий прототип функции для текущего файла проекта.

Прототипы функций для всех мега- и макрофункций хранятся в файлах с расширением .inc в директориях **\maxplus2\max2lib\mega_lpm** и **\maxplus2\max2inc** соответственно. Контекстно-зависимая подсказка для всех поставляемых фирмой Altera мега-, макрофункций и примитивов, выводит содержимое соответствующих им прототипов функций.

1.5.8 Оператор Options

Оператор Options предназначен для определения значения опции BIT0, указывающего в отношении группы, является ли бит с наименьшим номером битом с наибольшим весом (MSB - Most Significant Bit), битом с наименьшим весом (LSB - Least Significant Bit) или с весом, зависящим от места расположения индекса данного бита при описании группы. Использование данной опции позволяет избежать генерации сообщений предупредительного характера, если бит с наименьшим номером в группе используется не в качестве бита с наименьшим весом, что предполагается по умолчанию. При описании группы с размерностью, определенной диапазоном чисел, левое число представленного диапазона (заметим, что оно может быть и наименьшим и наибольшим в данном диапазоне) всегда представляет собой индекс бита с наибольшим весом (MSB - Most Significant Bit); соответственно правое число представленного диапазона (заметим также, что оно может быть и наименьшим и наибольшим числом указанного диапазона) всегда представляет собой индекс бита с наименьшим весом (LSB - Least Significant Bit). Если упомянутый диапазон чисел представлен в возрастающем порядке и при этом не установлена опция BIT0=MSB, то будет сгенерировано предупреждающее сообщение .Если использована опция BIT0=MSB и упомянутый диапазон представлен в убывающем порядке, то также будет сформировано предупреждающее сообщение. При установке опции BIT0=ANY можно определять размерности групп диапазонами чисел, представленных как в возрастающем, так и в убывающем порядке без генерации предупреждающих сообщений.

Оператор Options начинается ключевым словом OPTIONS, за которым следует опция BIT0 и ее установка. Оператор Options заканчивается символом (;).

Следующий пример демонстрирует использование оператора Options:

```
OPTIONS BIT0 = MSB;
```

В данном примере бит с наименьшим номером в группе определен как бит, имеющий наибольший вес (MSB).Другими возможными вариантами являются LSB - наименьший вес и ANY - вес, зависящий от расположения бита с наименьшим номером при описании группы.

Оператор Options расположенный в начале текстового файла проекта производит установку порядка следования битов в группах, действительную в пределах всего файла проекта. Если текущий файл проекта является файлом проекта верхнего уровня иерархии, то установка в операторе Options действительна в отношении всех подпроектов, входящих в данный проект верхнего уровня. Если текущий файл проекта не является файлом проекта верхнего уровня, то действие установки оператора Options распространяется только на данный файл проекта.

1.5.9 Оператор Assert

Оператор Assert позволяет проверять действительность выражений арбитражного характера, в которых используются параметры, числа, оценочные функции, а также статусные состояния портов (используется порт или не используется)

Следующий пример демонстрирует использование оператора Assert:

```
ASSERT (WIDTH > 0)
        REPORT          "Ширина (%) должна быть положительным целым"
WIDTH
        SEVERITY                ERROR
        HELP_ID                 INTVALUE;
-- for internal Altera use only
```

Оператор Assert имеет следующие характеристики:

За ключевым словом ASSERT следует арифметическое выражение, в необязательном порядке заключенное в круглые скобки. Когда выражение принимает значение ‘ложь’, строка - сообщение, следующая за ключевым словом REPORT, выводится в текстовом процессоре .При отсутствии условного выражения строка сообщения выводится безусловно.

За ключевым словом REPORT следует строка сообщения и необязательные параметры, представленные переменными. Строка сообщения заключается в двойные кавычки и может содержать символы % , которые замещаются значениями соответствующих переменных. Если ключевое слово REPORT не используется и при этом значение выражения арбитражного характера принимает значение ‘ложь’, то в текстовом процессоре выдается следующее сообщение:

<severity>: Line <line number>, File <filename>: Assertion failed

Необязательные переменные, включаемые в сообщение состоят из одного или более параметров, оценочных функций или арифметических выражений. Переменные, включаемые в сообщение отделяются друг от друга запятыми. Значения переменных подставляются в порядке появления в сообщении символов % .В примере, показанном выше, значение переменной WIDTH заменяет символ % в строке сообщения.

За необязательным ключевым словом SEVERITY следует уровень строгости ERROR, WARNING или INFO. По умолчанию предполагается уровень строгости ERROR.

Ключевое слово HELP_ID и строка - подсказка поддерживается в некоторых поставляемых фирмой Altera функциях и зарезервировано для внутреннего использования фирмой Altera.

Оператор Assert заканчивается символом (;).

Оператор Assert может использоваться внутри раздела Logic или за пределами других разделов языка AHDL.

1.5.10 Раздел Subdesign

Раздел Subdesign определяет входные, выходные и двунаправленные порты данного проекта.

Следующий пример демонстрирует использование раздела Subdesign:

```
SUBDESIGN top
(
        foo, bar, clk1, clk2          : INPUT = VCC;
        a0, a1, a2, a3, a4: OUTPUT;
        b[7..0]                       : BIDIR;
```

)

Раздел Subdesign имеет следующие характеристики:

- За ключевым словом SUBDESIGN следует имя подпроекта. Имя подпроекта должно совпадать с именем текстового файла проекта. В данном примере подпроект имеет имя top.
- Список сигналов заключается в круглые скобки.
- Сигналы представляются символическими именами с определением их типа (например, INPUT)
- Имена сигналов отделяются друг от друга запятыми .За именами следует двоеточие, далее тип сигналов и символ (;).
- Возможными типами портов являются : INPUT, OUTPUT, BIDIR, MACHINE INPUT или MACHINE OUTPUT.В примере, показанном выше, сигналы foo, bar, clk1 и clk2, а сигналы a0, a1, a2, a3 и a4 являются выходами. Шина b[7..0] является двунаправленной.
- Ключевые слова MACHINE INPUT и MACHINE OUTPUT используются для импорта и экспорта конечных автоматов между текстовыми файлами проектов и другими файлами проектов. Однако типы портов MACHINE INPUT и MACHINE OUTPUT не могут использоваться в текстовых файлах проектов верхнего уровня.
- После указания типа порта в необязательном порядке можно указать значение по умолчанию GND или VCC (в противном случае значений по умолчанию не предусматривается).В примере приведенном выше VCC является значением, присвоенным по умолчанию для входных сигналов в том случае, если они не используются в файле более высокого уровня иерархии (присвоения, осуществляемые в файле более высокого уровня иерархии, имеют больший приоритет)

В файле высшего уровня иерархии порты, имеющие тип INPUT, OUTPUT или BIDIR являются выводами устройства. В файлах более низких уровней иерархии все типы портов являются точками входа и выхода данного файла, но не устройства в целом.

1.5.11 Раздел Variable

Необязательный раздел Variable используется для описания и/или генерации переменных, используемых в разделе Logic. Переменные языка AHDL сходны с переменными, используемыми в языках высокого уровня; они используются для определения внутренней логики.

Следующий пример демонстрирует использование раздела Variable:

```
VARIABLE
        a, b, c    : NODE;
        temp               : halfadd;
        ts_node : TRI_STATE_NODE;
        IF DEVICE_FAMILY == "FLEX8000" GENERATE
                8kadder : flex_adder;
                d, e               : NODE;
        ELSE GENERATE
                7kadder : pterm_adder;
                f, g               : NODE;
        END GENERATE;
```

Раздел Variable может включать следующие операторы и конструкции:

- Описание объектов.
- Описание узлов.
- Описание регистров.
- Описание конечных автоматов.
- Описание псевдоимен конечных автоматов.

⇒ Раздел Variable может также содержать операторы If Generate, которые могут быть использованы для генерирования объектов, узлов, регистров, конечных автоматов, и псевдоимен конечных автоматов.

⇒ Раздел Variable имеет следующие характеристики:

- Раздел начинается с ключевого слова VARIABLE.

- Определенные пользователем символические имена переменных отделяются друг от друга запятыми, а от соответствующего им типа символом двоеточия. Допустимыми типами переменных являются: NODE, TRI_STATE_NODE, <*primitive*>, <*megafunction*>, <*macrofunction*> или <*state machine declaration*>. В примере, показанном выше, внутренними переменными являются a, b и c, имеющие тип NODE; temp является экземпляром макрофункции halfadd; и tsnode является объектом типа TRI_STATE_NODE.
- Каждая строка определения переменных заканчивается символом (;).

⇒ В файле с расширением .fit для текущего проекта могут иметь место имена, сгенерированные компилятором и имеющие в своем составе знак тильда (~). Если производится обратная аннотация присоединений, осуществленных в файле с расширением .fit, то эти имена появятся в файле установок и конфигурации (.acf).Символ тильды зарезервирован исключительно для имен генерируемых компилятором; использовать их для обозначения выводов, узлов и групп (шин) запрещено.

1.5.11.1 Описание объектов.

Каждое использование или реализация конкретной логической функции может быть произведено как объявлением переменной в разделе описания переменных, так и процедурой реализации объекта. После указанного объявления входные и выходные порты каждой логической функции можно использовать также как и порты проектируемого текстового файла проекта.

При необходимости реализации объекта мега- или макрофункции надо убедиться в существовании соответствующего ей файла с описанием ее логического функционирования. Затем используется оператор Function Prototype для описания портов и параметров функции и производится реализация функции посредством подставляемой ссылки или объявления объекта.

Для экземпляра примитива также используется подставляемая ссылка или объявления объекта. Однако, в отличие от мега- и макрофункций, логика функционирования примитива предопределена, поэтому нет необходимости определять логику функционирования примитива в отдельном текстовом файле проекта. В большинстве случаев нет необходимости использовать оператор Function Prototype.

При использовании процедуры объявления объекта в разделе описания переменных производится описание переменной типа *<primitive>, <megafunction>* или *<macrofunction>*. Для параметрических мега- и макрофункций объявление включает список параметров, используемых объектом и в необязательном порядке, значения этих параметров. После определения переменной порты объекта функции можно использовать с применением следующего формата:

<имя экземпляра>.<имя порта>

Например, если необходимо использовать в данном файле проекта функции adder и compare, нужно выполнить следующие описания экземпляров в разделе описания переменных:

```
VARIABLE
        comp  : compare;
        adder : lpm_add_sub WITH (LPM_WIDTH = 8);
```

Переменные comp и adder являются объектами функций compare и lpm_add_sub, имеющих следующие входы и выходы:

```
a[3..0], b[3..0]          : INPUT;-- входы компаратора
less, equal, greater      :OUTPUT;--выходы компаратора

a[8..1], b[8..1]          : INPUT; -- входы сумматора
sum[8..1]                 : OUTPUT;--выходы сумматора
```

Таким образом, в секции Logic можно использовать следующие порты переменных comp и adder:

comp.a[], comp.b[], comp.less, comp.equal, comp.greater
adder.dataa[], adder.datab[], adder.result[]

Эти порты могут использоваться в любом операторе также как и узлы.

Поскольку все примитивы имеют только один выход можно использовать имя примитива без указания имени его выходного порта (например, без .q или .out) в правой части выражений. Аналогично, если примитив имеет лишь один вход (т.е. все примитивы за исключением примитивов JKFF, JKFFE, SRFF и SRFFE), то можно использовать имя примитива без указания имени его входного порта в левой части выражений (т.е., без .d, .t или .in).

На этапе компиляции компилятор осуществляет поиск значений параметров мега- и макрофункций в порядке, описанном в разделе “Оператор Parameters”.

1.5.11.2 Описание узлов.

AHDL поддерживает два типа узлов : NODE и TRI_STATE_NODE.

Оба типа являются типами переменных общего назначения, используемых для хранения значений сигналов, которые не были описаны ни в разделе Subsection ни в разделе описания переменных. Таким образом, переменные обоих типов могут использоваться как с левой, так и с правой стороны выражения.

И NODE и TRI_STATE_NODE схожи с типами портов INPUT, OUTPUT и BIDIR, описываемых в разделе Subsection, в том, что и те и другие представляют проводники, по которым распространяются сигналы.

⇒ В файле с расширением .fit для текущего проекта могут иметь место имена, сгенерированные компилятором и имеющие в своем составе знак тильда (~). Если производится обратная аннотация присоединений, осуществленных в файле с расширением .fit, то эти имена появятся в файле установок и конфигурации (.acf). Символ тильды зарезервирован исключительно для имен генерируемых компилятором; использовать их для обозначения выводов, узлов и групп (шин) запрещено.

Следующий пример демонстрирует процедуру определения узла:

```
SUBDESIGN node_ex
(
        a, oe       : INPUT;
        b           : OUTPUT;
        c           : BIDIR;
)
VARIABLE
        b : NODE;
        t : TRI_STATE_NODE;
BEGIN
        b = a;
        out = b   % следовательно out = a %
        t = TRI(a, oe);
        t = c;     % t есть шина c и a %
END;
```

NODE и TRI_STATE_NODE отличаются тем, что многократное присваивание значений этим объектам дает различные результаты:

- Многократные присваивания узлам типа NODE объединяют сигналы в соответствии с функцией монтажное И или монтажное ИЛИ. Значения переменных по умолчанию, определенные в операторах Default детерминируют поведение: значение по умолчанию VCC предопределяет выполнение функции монтажное И над несколькими значениями, присваиваемыми к данному узлу; соответственно значение по умолчанию GND предопределяет выполнение функции монтажное ИЛИ.
- Если только одна переменная присвоена узлу типа TRI_STATE_NODE , то он ведет себя также как и узел типа NODE .

Следующие примитивы и сигналы могут быть подключены к узлам типа TRI_STATE_NODE:

- Примитивы TRI.
- Порты типа INPUT файла проекта с файлами проекта более высокого уровня иерархии.
- Порты типа OUTPUT и BIDIR файла проекта с файлом проекта более низкого уровня иерархии.
- Порты типа BIDIR данного файла проекта.
- Другие узлы типа TRI_STATE_NODE данного файла проекта.

1.5.11.3 Объявление регистров.

Объявление регистров используется для определения регистров, включая D, T, JK и SR триггеры (DFF, DFFE, TFF, TFFE, JKFF, JKFFE, SRFF и SRFFE) и защелки (LATCH). Следующий пример демонстрирует описание регистра:

```
VARIABLE
        ff : TFF;
```

Именем объекта, представляющего собой T - триггер, является ff. После данного объявления можно использовать входной и выходной порты объекта ff с использованием следующего формата:

ff.t
ff.clk
ff.clrn
ff.prn
ff.q

Поскольку все примитивы имеют только один выход можно использовать имя примитива без указания имени его выходного порта (например, без .q или .out) в правой части выражений. Аналогично, если примитив имеет лишь один вход (т.е. все примитивы за исключением примитивов JKFF, JKFFE, SRFF и SRFFE), то можно использовать имя примитива без указания имени его входного порта в левой части выражений (т.е., без .d, .t или .in).

Например, прототип функции для примитива DFF имеет вид : FUNCTION DFF(d, clk, clr, prn) RETURNS (q); . В следующем текстовом файле проекта выражение a = b эквивалентно a.d = b.q:

```
VARIABLE
        a, b : DFF;
BEGIN
        a = b;
END;
```

1.5.11.4 Объявление конечных автоматов.

Конечный автомат создается определением его имени, состояний и в необязательном порядке его битами в разделе описания переменных.

Следующий пример демонстрирует описание конечного автомата:

```
VARIABLE
        ss :    MACHINE
                        OF BITS (q1, q2, q3)
                        WITH STATES (
                                s1 = B"000",
```

```
s2 = B"010",
s3 = B"111");
```

Имя конечного автомата в данном примере ss. Биты состояний q1, q2 и q3 являются выходами регистров данного автомата. Состояниями данного конечного автомата являются s1, s2 и s3, каждому из которых присвоено числовое значение представленное битами q1, q2 и q3.

Процедура объявления конечного автомата имеет следующие характеристики:

- Конечный автомат имеет символическое имя. В примере, показанном выше, именем конечного автомата является ss.
- За именем конечного автомата следует двоеточие и далее ключевое слово MACHINE.
- Определение конечного автомата должно включать список состояний, а также может включать имена битов состояний.
- Необязательное указание имен битов состояний производится с использованием ключевого слова OF BITS, за которым следует список имен битов, отделенных друг от друга запятыми ;список должен быть заключен в круглые скобки. В примере, показанном выше, определены имена битов состояний q1, q2 и q3.
- Состояния определяются ключевыми словами WITH STATES, за которым следует список имен состояний отделенных друг от друга запятыми ;этот список также должен быть заключен в круглые скобки. В примере, показанном выше определены имена состояний s1, s2 и s3.
- Первое состояние указанное в списке состояний за ключевыми словами WITH STATES является состоянием Reset для конечного автомата.
- В необязательном порядке именам состояний могут быть присвоены числовые значения, следующие за знаком (=) после соответствующего имени состояния. В примере, показанном выше, состоянию с именем s1 присвоено числовое значение B”000”, состоянию с именем s2 присвоено числовое значение B”001” и s3 присвоено значение B”010”.

- Предусмотрена возможность определения псевдонима имени конечного автомата, объявленного в данном текстовом файле проекта или импортируемого из другого файла.
- Символ (;) заканчивает конструкцию определения конечного автомата.

⇒ Каждое состояние конечного автомата представляется уникальным набором значений на выходах триггеров, хранящих состояния конечного автомата. Количество состояний связано с количеством битов состояний конечного автомата следующим образом:

<количество состояний> = 2^<количество битов состояний>

1.5.11.5 Объявления псевдоимен конечных автоматов.

Используя процедуру объявления псевдоимени конечного автомата в разделе описания переменных, можно описать псевдоимя для данного, описанного или импортированного из другого файла конечного автомата. После указанной процедуры можно пользоваться псевдоименем конечного автомата наравне с его исходным. Например:

```
FUNCTION ss_def (clock, reset, count)
        RETURNS (MACHINE ss_out);
VARIABLE
        ss : MACHINE;
BEGIN
        ss = ss_def (sys_clk, reset, !hold);
        IF ss == s0 THEN
        ELSIF ss == s1 THEN
END;
```

Процедура объявления псевдоимени конечного автомата имеет следующие характеристики:

- Псевдоним представляет собой символическое имя. За псевдонимом следует символ двоеточия и далее ключевое слово MACHINE. В примере, показанном выше, символическое имя ss является псевдонимом конечного автомата.
- Предусмотрена возможность импортирования и экспортирования конечных автоматов между текстовыми файлами проектов, а также другими файлами проектов путем определения входных и выходных портов с использованием ключевых слов MACHINE INPUT или MACHINE OUTPUT в разделе Subdesign.
- При импортировании и экспортировании конечных автоматов прототип функции, представляющий файл описания конечного автомата должен определять какие входы и выходы являются конечными автоматами. В примере, показанном выше, ss_out является именем конечного автомата.
- Декларация псевдонима конечного автомата заканчивается символом (;).

⇒ Порты типов MACHINE INPUT и MACHINE OUTPUT не могут использоваться в файлах проектов верхнего уровня.

1.5.12 Раздел Logic

Раздел Logic определяет логическое функционирование текстового файла проекта (TDF) и является собственно его телом. В этом разделе могут быть многократно использованы следующие операторы и разделы:

- Булевские выражения.
- Управляющие булевские выражения.
- Оператор Case.
- Оператор Defaults.
- Оператор If Then.
- Оператор If Generate
- Оператор If Generate
- Оператор таблицы истинности

⇒ Раздел Logic может также содержать оператор Assert.

Раздел Logic заключается в ключевые слова BEGIN и END. За ключевым словом END следует символ (;), заканчивающий раздел. Если используется оператор Defaults, то он должен предшествовать всем другим операторам в этом разделе.

AHDL является параллельным языком. Компилятор анализирует поведенческую модель, описанную в разделе Logic, параллельно. Выражения, осуществляющие множественные присваивания объекту, имеющему тип NODE или переменной, объединяются в соответствии с функцией монтажное ИЛИ.

1.5.12.1 *Булевские выражения*.

Булевские выражения используются в разделе Logic текстового файла проекта на языке AHDL для представления соединений узлов, входных и выходных потоков сигналов через входные и выходные выводы, примитивы, макро- и мегафункции и конечные автоматы.

Следующий пример демонстрирует сложное булевское выражение:

```
a[] = ((c[] & -B"001101") + e[6..1]) # (p, q, r, s, t, v);
```

Левая часть выражения может быть символическим именем, именем порта или именем группы. Для инвертирования выражения в левой части выражения можно пользоваться операцией NOT (!). Правая часть равенства представлена булевским выражением, вычисляемым в порядке, описанном в разделе “Приоритеты булевских операторов и операций отношения”.

Символ эквивалентности (=) используется в булевских выражениях для индикации того, что результат булевского выражения, представленного в правой части, является источником сигнала для символического объекта или группы в левой части. Символ (=) отличается от символа (==), используемого как компаратор.

В примере, показанном выше, булевское выражение в правой части равенства вычисляется в соответствии со следующими правилами:

1. Двоичное число B”001101” меняет знак и принимает вид B”110011”. Унарная операция (-) имеет наивысший приоритет.

2. B”110011” объединяется по И с группой c[]. Эта операция имеет второй уровень приоритета, потому что она заключена в круглые скобки.
3. Результат групповой операции, проведенной на втором шаге, прибавляется к группе e[6..1].
4. Результат, полученный на третьем шаге, объединяется по ИЛИ с группой (p, q, r, s, t, v). Это выражение имеет наименьший уровень приоритета.

Результат операции присваивается группе a[ ].

Для корректного выполнения операций, показанных выше, необходимо, чтобы количество бит в группе в левой части выражения было равно или делилось нацело на число бит в группе в правой части выражения. Биты в левой части выражения отображаются на соответствующие биты в правой части выражения по порядку.

В отношении булевских выражений используются следующие правила:

- Множественные присваивания, осуществляемые в отношении переменной объединяются в соответствии с монтажным ИЛИ (#), исключая тот случай, когда значением по умолчанию для этой переменной является VCC.
- Узлы в левой части булевского выражения однозначно соответствуют узлам в правой части .
- Если значение одиночного узла, VCC или GND присваиваются группе, то значение узла или константы копируется до размерности группы . Например, (a, b) = e эквивалентно a = e и b = e.
- Если и левая и правая части выражения представляют собой группы одинакового размера, то каждый член группы, расположенной в правой части, соответствует тому члену группы в левой части, который расположен на той же позиции .Например, (a, b) = (c, d) эквивалентно a = c и b = d.

⇒ При сложении двух групп в правой части булевского выражения с использованием операции (+) можно добавить символ “0” слева каждой группы для знакового расширения. Этот метод может быть использован для получения дополнительного бита сигнала переноса в группе, расположенной в левой части выражения. В следующем примере группы count[7..0] и delta[7..0] представлены в знакорасширенном формате для получения значения бита переноса, обозначенного символическим именем cout в левой части выражения:

(cout, answer[7..0]) = (0, count[7..0]) + (0, delta[7..0])

- Если в левой и правой частях булевского выражения расположены группы разных размерностей, то количество бит в группе слева должно быть равно или делиться нацело на количество бит в правой части выражения. Биты в левой части выражения отображаются на биты в правой части выражения по порядку. Следующая запись является корректной:

  a[4..1] = b[2..1]

  В данном выражении биты отображаются в следующем порядке:

  a4 = b2

  a3 = b1

  a2 = b2

  a1 = b1
- Группа узлов или чисел не может быть присвоена одиночному узлу.
- Если число в правой части выражения присваивается группе, расположенной в левой части выражения, то число усекается или расширяется путем распространения знака до соответствия размеру группы в левой части. Если при этом происходит усечение значащих битов, то компилятор выдает сообщение об ошибке. Каждый член в правой части выражения присваивается соответствующему члену в левой части выражения по порядку. Например, (a, b) = 1 эквивалентно a = 0; b =1;
- Запятые могут использоваться для резервирования места под неупомянутые элементы группы в булевских выражениях Следующий пример демонстрирует использование запятых для резервирования места под отсутствующие элементы группы (a, b, c, d) :

(a, , c,) = B"1011";

В данном примере элементам a и c присваивается значение “1”.

- Каждое выражение заканчивается символом (;).

1.5.12.2 Управляющие булевские выражения.

Управляющие булевские выражения используются в разделе Logic для определения значений сигналов Clock, Reset и Clock Enable в конечных автоматах.

Следующий пример демонстрирует использование управляющих булевских выражений:

```
ss.clk = clk1;
ss.reset = a & b;
ss.ena = clk1ena;
```

Управляющие булевские выражения имеют следующие характеристики:

- Значения сигналов Clock, Reset и Clock Enable для всякого конечного автомата могут быть определены с использованием следующего формата: *<имя конечного автомата>.<имя порта>.*В примере, показанном выше, значения этих входов определены для конечного автомата с именем ss.
- Имя конечного автомата, определенное на этапе его объявления, может быть использовано в управляющих булевских выражениях.
- Тактирующему сигналу *<имя конечного автомата>*.clk должно быть присвоено значение.
- Если в качестве начального значения конечного автомата выбрано ненулевое значение, то должен использоваться сигнал начальной установки *<имя конечного автомата>.reset;* в противном случае использование этого сигнала необязательно.
- Использовать тактирующий сигнал *<имя конечного автомата>.ena* необязательно.
- Каждое выражение заканчивается символом (;).

1.5.12.3 Оператор Case.

Оператор Case определяет список альтернативных вариантов, которые могут быть активизированы в зависимости от значения переменной, группы или выражения, следующего за ключевым словом CASE.

Следующий пример демонстрирует использование оператора Case:

```
CASE f[].q IS
        WHEN H"00" =>
                addr[] = 0;
                s = a & b;
        WHEN H"01" =>
                count[].d = count[].q + 1;
        WHEN H"02", H"03", H"04" =>
                f[3..0].d = addr[4..1];
                WHEN OTHERS =>
                        f[].d = f[].q;
        END CASE;
```

Оператор Case имеет следующие характеристики:

- Булевское выражение, группа или конечный автомат располагаются между ключевыми словами CASE и IS (в примере, показанном выше, это f[ ].q).
- Оператор Case завершается ключевыми словами END CASE за которыми следует символ (;).
- Телом оператора Case является список из одного или более неповторяющихся альтернативных вариантов, следующих за ключевым словом WHEN. Каждому альтернативному варианту предшествует ключевое слово WHEN.
- Каждый альтернативный вариант представляет собой одно или более отделенных друг от друга запятыми значений констант, за которыми следует символ (=>).

- Если значение булевского выражения, стоящего за ключевым словом CASE, соответствует какому - либо альтернативному варианту, то все операторы, следующие за соответствующим символом (=>) активизируются. В примере, приведенном выше, если f[ ].q равно h”01”, то активизируется булевское выражение count[ ].d = count[ ].q + 1.
- Если значение булевского выражения, стоящего за ключевым словом CASE не равно ни одному из альтернативных вариантов, то активизируется альтернативный вариант, стоящий за ключевыми словами WHEN OTHERS. В примере, показанном выше, если значение f[ ].q не равно H”00”, H’01” или H”CF”, то активизируется выражение f[ ].d = f[].q.
- Оператор Defaults определяет значение по умолчанию для тех случаев, когда ключевые слова WHEN OTHERS не используются.
- Если оператор Case используется для определения переходов конечного автомата, то ключевые слова WHEN OTHERS не могут использоваться для выхода из недопустимых состояний. Если состояния конечного автомата определяются n -мерным кодом и при этом автомат имеет 2^n состояний, то использование ключевых слов WHEN OTHERS является допустимым.
- Каждый альтернативный вариант должен заканчиваться символом (;).

1.5.12.4 Оператор Defaults.

Оператор Defaults позволяет определять значения по умолчанию, применяемые в таблицах истинности, а также в операторах If Then и Case. Поскольку активно- высокие сигналы автоматически имеют значения по умолчанию GND, то оператор Default необходим лишь в случае использования активно-низких сигналов.

⇒ Не следует путать значения по умолчанию, присваиваемые переменным со значениями по умолчанию, присваиваемыми портам в разделе Subdesign.

Следующий пример демонстрирует использование оператора Defaults:

```
BEGIN
        DEFAULTS
```

```
        a = VCC;
    END DEFAULTS;

    IF y & z THEN
    a = GND;        % a активный низкий %
    END IF;
END;
```

Оператор Defaults имеет следующие характеристики:

- Значения по умолчанию заключаются в ключевые слова DEFAULTS и END DEFAULTS. Оператор заканчивается символом (;).
- Тело оператора Defaults состоит из одного или более логических выражений, присваиваемых константам или переменным. В примере, показанном выше, значение по умолчанию VCC присваивается переменной a.
- Каждое выражение заканчивается символом (;).
- Оператор Default активизируется в том случае, когда какая-либо переменная, включенная в список оператора Default в каком-либо из операторов, оказывается неопределенной. В примере, показанном выше, переменная *a* оказывается неопределенной, если *y* и *z* имеют значения логического нуля; таким образом активизируется выражение (*a* = VCC) в операторе Default.

При использовании оператора Default необходимо соблюдать следующие правила:

- В разделе Logic допускается использовать не более одного оператора Default и кроме того при его использовании он должен располагаться сразу за ключевым словом BEGIN.
- Если в операторе Default в отношении одной и той же переменной производятся многократные присваивания, то все присваивания за исключением последней игнорируются.

- Оператор Default не может использоваться для присваивания значения X (безразлично) переменным.
- Многократные присваивания значений узлу, имеющему тип NODE, объединяются в соответствии с функцией логическое ИЛИ, за исключением того случая, когда значением по умолчанию для этой переменной является VCC. Следующий пример текстового файла проекта (TDF) иллюстрирует значения по умолчанию для двух переменных: *a* с значением по умолчанию GND и *bn* с значением по умолчанию VCC:

```
BEGIN
        DEFAULTS
                a = GND;
                bn = VCC;
        END DEFAULTS;

        IF c1 THEN
                a = a1;
                bn = b1n;
        END IF;

        IF c2 THEN
                a = a2;
                bn = b2n;
        END IF;
END;
```

Этот пример эквивалентен следующему выражению:

```
a = c1 & a1 # c2 & a2;
bn = (!c1 # b1n) & (!c2 # b2n);
```

- Переменные, имеющие активно низкий уровень и участвующие в многократных присваиваниях, должны иметь значение по умолчанию VCC. В следующем примере reg[].clrn имеет значение по умолчанию VCC:

```
SUBDESIGN 5bcount
(
        d[5..1]         : INPUT;
        clk             : INPUT;
        clr             : INPUT;
        sys_reset       : INPUT;
        enable          : INPUT;
        load            : INPUT;
        q[5..1]         : OUTPUT;
)
VARIABLE
        reg[5..1]       : DFF;
BEGIN
        DEFAULTS
                reg[].clrn = VCC;
        END DEFAULTS;

        reg[].clk = clk;
        q[]     = reg[];

        IF sys_reset # clr THEN
                reg[].clrn = GND;
        END IF;

        !reg[].prn  = (load & d[]) & !clr;
        !reg[].clrn = load & !d[];
        reg[] = reg[] + (0, enable);
END;
```

1.5.12.5 Оператор If Then.

Оператор If Then содержит список операторов, выполняемых в том случае, если булевское выражение, расположенное между ключевыми словами IF и THEN, принимает истинное значение .

Следующий пример демонстрирует использование оператора If Then:

```
IF a[] == b[] THEN
        c[8..1] = H "77";
        addr[3..1] = f[3..1].q;
        f[].d = addr[] + 1;
ELSIF g3 $ g4 THEN
        f[].d = addr[];
ELSE
        d = VCC;
END IF;
```

Оператор If Then имеет следующие характеристики:

- Между ключевыми словами IF и THEN располагается булевское выражение, в зависимости от значения которого выполняется или не выполняется список операторов, располагающийся за ключевым словом THEN. Каждый оператор в этом списке оканчивается символом (;).
- Между ключевыми словами ELSEIF и THEN располагается дополнительное булевское выражение а за ключевым словом THEN также располагается список операторов, выполняемых в зависимости от значения булевского выражения. Эти необязательные ключевые слова и операторы могут повторяться многократно.
- Оператор(ы), следующий за ключевым словом THEN, активизируется в том случае, если соответствующее ему булевское выражение принимает истинное значение. При этом последующие конструкции ELSEIF THEN игнорируются.

- Ключевое слово ELSE, за которым следует один или более операторов, схоже по своему значению с ключевыми словами WHEN OTHERS в операторе Case. Если ни одно из булевских выражений не приняло истинное значение, то выполняются операторы, следующие за ключевым словом ELSE. В примере, показанном выше, если ни одно из булевских выражений не приняло истинного значения, то выполняется оператор d = VCC. Использование ключевого слова ELSE не является обязательным.
- Значения булевских выражений, следующих за ключевыми словами IF и ELSEIF оцениваются последовательно.
- Оператор If Then заканчивается ключевыми словами END IF за которыми следует символ (;).

Оператор If Then может генерировать логические схемы, которые слишком сложны для компилятора. Если оператор If Then содержит сложные булевские выражения, то учет инверсии каждого из этих выражений вероятно приведет к еще более сложным булевским выражениям. Например, если a и b сложные выражения, то инверсия этих выражений может быть еще более сложной.

Оператор If:	Интерпретация компилятором:
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;

⇒ В отличие от операторов If Then, которые могут оценивать лишь значения булевских выражений, операторы If Generate могут оценивать значения наборов арифметических выражений. Основное различие между операторами If Then и If Generate состоит в том, что в первом случае значение булевского выражения оценивается аппаратным способом (в кремнии), а во втором случае значение набора арифметических выражений оценивается на этапе компиляции.

1.5.12.6 Оператор If Generate

Оператор If Generate содержит список операторов, активизирующийся в случае положительного результата оценки арифметического выражения.

Следующий пример демонстрирует использование оператора If Generate:

```
IF DEVICE_FAMILY == "FLEX8K" GENERATE
        c[] = 8kadder(a[], b[], cin);
ELSE GENERATE
        c[] = otheradder(a[], b[], cin);
END GENERATE;
```

Оператор If Generate имеет следующие характеристики:

- Между ключевыми словами If Generate заключается арифметическое выражение, значение которого подвергается оценке. За ключевым словом GENERATE следует список операторов, каждый из которых заканчивается символом (;). Операторы активизируются в том случае, если арифметическое выражение принимает истинное значение.
- За ключевыми словами ELSE GENERATE следует один или более операторов, которые активизируются в случае, если арифметическое выражение принимает ложное значение.

- Оператор If Generate заканчивается ключевыми словами END GENERATE, за которыми следует символ (;).
- Оператор If Generate может использоваться в разделе Logic и в разделе Variable.

⇒ В отличие от операторов If Then, которые могут оценивать лишь значения булевских выражений, операторы If Generate могут оценивать значения наборов арифметических выражений. Основное различие между операторами If Then и If Generate состоит в том, что в первом случае значение булевского выражения оценивается аппаратным способом (в кремнии), а во втором случае значение набора арифметических выражений оценивается на этапе компиляции.

⇒ Оператор If Generate особенно часто используется с операторами For Generate, что позволяет различным образом обрабатывать особые ситуации, например, младший значащий бит в многокаскадном умножителе. Этот оператор может также использоваться для тестирования значений параметров, как показано в последнем примере.

1.5.12.7 Оператор For Generate.

Следующий пример показывает использование итерационного оператора For Generate:

```
CONSTANT NUM_OF_ADDERS = 8;

SUBDESIGN 4gentst
(
a[NUM_OF_ADDERS..1], b[NUM_OF_ADDERS..1],
 cin                                : INPUT;
c[NUM_OF_ADDERS..1], cout           : OUTPUT;
)
VARIABLE
        carry_out[(NUM_OF_ADDERS+1)..1] : NODE;
BEGIN
```

```
carry_out[1] = cin;
        FOR i IN 1 TO NUM_OF_ADDERS GENERATE
                c[i] = a[i] $ b[i] $ carry_out[i];                % Полный сумматор
%
carry_out[i+1] = a[i] & b[i] # carry_out[i] & (a[i] $ b[i]);
        END GENERATE;
cout = carry_out[NUM_OF_ADDERS+1];
END;
```

Оператор For Generate имеет следующие характеристики:

- Между ключевыми словами FOR и GENERATE заключаются следующие параметры:

    1. Временная переменная, представляющая собой символическое имя. Эта переменная используется лишь в пределах оператора For Generate и заканчивает свое существование после того, как компилятор обработает этот оператор. В примере, показанном выше такой переменной является переменная *i*. Это имя не может использоваться в качестве имени константы, параметра или узла в пределах данного проекта.
    2. За ключевым словом IN следует диапазон, ограниченный двумя арифметическими выражениями. Арифметические выражения разделяются между собой ключевым словом TO. В примере, показанном выше арифметическими выражениями являются 1 и NUM_OF_ADDRESS. Границы диапазона могут содержать выражения, состоящие только из констант и параметров; использование переменных при этом недопустимо.
- За ключевым словом GENERATE следует один или более логических операторов, каждый из которых заканчивается символом (;).
- Оператор If Generate заканчивается ключевыми словами END GENERATE, за которыми следует символ (;).

1.5.12.8 Подставляемая ссылка для реализации логической функции (In-Line Logic Function Reference).

Подставляемая ссылка для реализации логической функции представляет собой булевское выражение. Это быстрый способ для реализации логической функции, требующий лишь одну строку в разделе Logic и не требующий объявления переменной.

При необходимости реализации объекта мегафункции или макрофункции нужно убедиться, что логика ее функционирования описана в соответствующем файле проекта. Затем с помощью оператора Function Prototype декларируется прототип функции и далее реализуется объект функции посредством подставляемой ссылки или путем объявления объекта.

Для реализации объекта примитива также используется подставляемая ссылка или производится описание в разделе объявления объектов. Однако в отличие от мега - и макрофункций логика функционирования примитивов предопределена, то есть нет необходимости определять логику функционирования примитива в отдельном файле проекта. В большинстве случаев нет необходимости использовать оператор Function Prototype для определения прототипа функции.

Следующие примеры демонстрируют прототипы функций *compare* и *lpm_add _sub*. Функция compare имеет входные порты a[3..0] и b[3..0], а также выходные порты less, equal, greater; функция lpm_add_sub имеет входные порты dataa[LPM_WIDTH-1..0], cin и add_sub, а также выходные порты result[LPM_WIDTH-1..0], cout и overflow.

```
FUNCTION compare (a[3..0], b[3..0])
        RETURNS (less, equal, greater);

FUNCTION lpm_add_sub (cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-1..0],
add_sub)
        WITH (LPM_WIDTH, LPM_REPRESENTATION)
        RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
```

Подставляемые ссылки (in-line logic function references) для функций *compare* и *lpm_add_sub* указываются в правой части показанного ниже выражения:

```
(clockwise, , counterclockwise) = compare(position[], target[]);
sum[] = lpm_add_sub (.datab[] = b[], .dataa[] = a[])
  WITH (LPM_WIDTH = 8)
  RETURNS (.result[]);
```

Подставляемая ссылка для логической функции имеет следующие характеристики:

- За именем функции справа от символа равенства (=) следует заключенный в круглые скобки список сигналов, содержащий символические имена, десятичные числа или группы разделенные между собой запятыми. Все эти компоненты описывают входные порты функции.
- В списке сигналов имена портов могут иметь позиционное соответствие, либо соответствие по имени:
    - В примере, показанном выше и демонстрирующем использование функции compare, имена переменных position[] и target[] имеют позиционное соответствие портам a[3..0] и b[3..0]. При использовании позиционного соответствия можно применять запятые для резервирования места под выходы, не подсоединяемые к конкретным переменным. В функции compare выход equal не подключается к переменным, поэтому необходимо использовать запятую для резервирования его места в правой части выражения.
    - В примере, показанном выше и демонстрирующем использование функции lpm_add_sub, входы .datab[] и .dataa[] соединяются соответственно с переменными b[] и a[] путем установления соответствия по имени. Соответствие между переменными и портами устанавливается посредством использования символа (=).

1. Имена портов должны иметь следующий формат .<имя порта> как в правой, так и в левой части подставляемой ссылки, использующей способ установления соответствия портов переменным по имени.

2. Установление соответствия портов переменным по имени возможно лишь в правой части подставляемой ссылки. В левой части подставляемой ссылки всегда используется позиционное соответствие.

- В параметризируемой функции, за ключевым словом WITH и списком имен параметров следует список входных портов. Этот список заключается в круглые скобки, а имена параметров разделяются запятыми. Декларируются лишь те параметры, которые используются объектом; значения параметров отделяются от имен параметров посредством символа равенства. В примере, показанном выше и демонстрирующим использование функции lpm_add_sub, параметру LPM_WIDTH присвоено значение *8*. Если какому-либо параметру не присвоено никакого значения, то компилятор осуществляет поиск значений для этих параметров в том порядке, который описан в разделе “Оператор Parameters”.
- В левой части подставляемой ссылки выходы функции ставятся в соответствие переменным. В примере, показанном выше и демонстрирующем использование функции *compare* выходы *less* и *greater* поставлены в соответствие переменным *clockwise* и *counterclockwise* с использованием позиционного соответствия. Подобным же образом в примере для функции lpm_add_sub выходы *result[]* поставлены в соответствие группе *sum[]* с использованием позиционного соответствия.
- Значения переменных, которые определены где-либо в разделе Logic, являются значениями связанными с соответствующими им входами и выходами. В примере, показанном выше для функции *compare*, значения *position[]* и *target[]* являются значениями, подаваемыми на соответствующие входы функции compare. Значения выходных портов less и greater связаны с clockwise и counterwise, соответственно. Эти переменные могут быть использованы в других выражениях раздела Logic.

1.5.12.9 Оператор Truth Table.

Оператор Truth Table используется для определения комбинационной логики или для определения поведения автоматов. В таблицах истинности, используемых в AHDL каждая строка таблицы состоит из комбинации входных значений и соответствующих этой комбинации выходных значений. Эти выходные значения могут использоваться как обратные связи для определения переходов автоматов из одного состояния в другое, а также его выходов.

Следующий пример демонстрирует использование оператора Truth Table:

```
TABLE
a0,       f[4..1].q   => f[4..1].d,       control;

0,        B"0000"  =>  B"0001",      1;
0,        B"0100"  =>  B"0010",      0;
1,        B"0XXX"  =>  B"0100",      0;
X,        B"1111"  =>  B"0101",      1;
END TABLE;
```

Оператор Truth Table имеет следующие характеристики:

- Заголовок таблицы истинности состоит из ключевого слова TABLE, за которым следует разделенный запятыми список входов, символ (=>) и разделенный запятыми список выходов таблицы. Заголовок таблицы истинности заканчивается символом (;).
- Входы таблицы истинности являются булевскими выражениями; выходы являются переменными. В примере, показанном выше, входными сигналами являются *a0* и *f[4..1].q*; выходными сигналами являются *f[4..1]* и *control*.

Тело таблицы истинности состоит из одного или более компонентов, каждый из которых представляет одну или более строку и заканчивается символом (;).

Каждый компонент состоит из разделенного запятыми списка входов и разделенного запятыми списка выходов. Входы и выходы разделены символом (=>).

Каждый сигнал имеет однозначное соответствие с значениями в каждом компоненте тела таблицы истинности. Таким образом, первый компонент в примере, показанном выше, определяет, что когда *a0* имеет значение *0*, а *f[4..1].q* имеет значение *B"0000",* то *f[4..1].d* примет значение *B"0001",* а сигнал *control* примет значение 1.

Входные и выходные значения могут быть числами, предопределенными константами VCC и GND, символическими константами (т.е. символическими именами, используемыми как константы) или группами чисел или констант. Входные значения могут также иметь значение X (безразличное состояние).

Входные и выходные значения соответствуют входам и выходам, названия которых указаны в заголовке таблицы.

Описание таблицы истинности заканчивается ключевыми словами END TABLE, за которыми следует символ (;).

В отношении описания таблицы истинности необходимо соблюдать следующие правила:

- Имена, используемые в заголовке таблицы истинности должны представлять собой либо одиночные узлы, либо группы.
- Нет необходимости оговаривать в таблице истинности все возможные комбинации входных сигналов. Можно использовать символ "X" для определения того, что выходное значение не зависит от входного. Следующий пример определяет, что, если a0 имеет высокий уровень и f4 имеет низкий уровень, то логические уровни остальных входов не имеют значения. Таким образом, можно указать лишь общую часть нескольких комбинаций входных сигналов, а для всех остальных использовать символ "X":

```
TABLE
a0,     f[4..1].q  =>  f[4..1].d,     control;

0,      B"0000"  =>  B"0001",     1;
0,      B"0100"  =>  B"0010",     0;
1,      B"0XXX"  =>  B"0100",     0;
```

```
X,        B"1111" => B"0101",      1;
END TABLE;
```

- Количество разделенных запятыми элементов таблицы истинности должно в точности соответствовать количеству элементов в заголовке таблицы истинности. В противном случае в отношении выходных сигналов используются значения по умолчанию.
- При использовании символа “X” для определения нескольких комбинаций значений входных сигналов необходимо внимательно следить за тем, чтобы определяемое таким образом подмножество комбинаций не перекрывалось ни с каким другим подмножеством в пределах данной таблицы истинности. В противном случае возможны непредсказуемые результаты.

1.6 Синтаксис

1.7 Стилизация

1.8 Золотые правила

1.9 Контекстно-зависимая помощь

Для получения контекстно-зависимой помощи, Вы сперва должны сохранить Ваш текстовой файл с расширением .**tdf**.

Контекстно-зависимая помощь доступна для следующих элементов:

- Арифметические операторы (в булевых выражениях)
- Арифметические операторы (в арифметических выражениях)
- Компараторы
- Логические операторы
- Мегафункции
- Макрофункции
- Примитивы
- Зарезервированные идентификаторы
- Зарезервированные ключевые слова
- Символы