Правила оформления исходного кода

ООО "ПК "Дельта"

v1.0, 13.05.2022

Назначение и область применения

Документ определяет набор правил, которыми следует руководствоваться при разработке нового исходного кода.

В первой части документа приведены общие правила оформления исходного кода.

Дополнительно в следующих главах приведены специфичные для каждого языка требования к оформлению исходного кода:

- "Правила оформления исходного кода на языке C";
- "Правила оформления исходного кода на языке SystemVerilog";
- "Правила оформления исходного кода на языке VHDL";
- "Правила оформления исходного кода на языке Python";
- "Правила оформления исходного кода на языке Perl 5".

Следование требованиям настоящего документа является обязательным при работе над новыми программными проектами. При работе над исходным кодом, к которому не применялись положения настоящих правил, следует придерживаться принятых для этого исходного кода правил оформления.

Оглавление

Используемые сокращения

Сокращение	Значение	Сокращение, англ.	Значение, англ.
ИПП	Интерфейс прикладного программирования	API	Application Programming Interface
ИСР	Интегрированная среда разработки	IDE	Integrated Development Environment

Введение

Если положения настоящего документа допускают вариативность оформления тех или иных конструкций, следует придерживаться одного варианта оформления в рамках всего программного проекта.

1. Общие правила оформления исходного кода

1.1. Именование

При именовании любых объектов (тип, переменная, функция, модуль, файл, макрос и т. д.) необходимо следовать следующим правилам:

- Имена должны описывать назначение объекта.
- Экономия горизонтального пространства вторична, важнее, чтобы код был понятен читателю.
- Следует свести к минимуму использование аббревиатур. Допускается использование общеупотребимых, либо однозначно понимаемых в предметной области программного проекта аббревиатур. Допускается использование аббревиатур в случаях, когда она является общим префиксом нескольких идентификаторов.
- Описательность объекта должна быть пропорциональна его области видимости. Например, `n` может быть хорошим именем в пятистрочной функции. Также допустимы общеизвестные сокращения, например, `i` для переменной цикла.

1.2. Комментирование

Исходный код должен быть исчерпывающе документирован. Тем не менее, комментирование тривиальных конструкций не требуется. Исходный код и комментарии должны вместе создавать связную структуру и читаться как единый текст, уточняя, поясняя и иллюстрируя друг друга.

Комментарии должны быть оформлены в виде прозы на русском языке.

Не следует помещать в комментариях имя автора, дату создания или изменения файла, описание внесённых изменений. Учёт изменений должен осуществляться системой управления исходным кодом.

Рекомендуется приводить библиографические ссылки или гиперссылки на материалы, содержащие методы (алгоритмы), реализованные в исходном коде. Для ссылок на публикацию (издание) рекомендуется приводить идентификатор публикации (издания).

Использование не имеющих смысловой нагрузки комментариев исключительно в качестве разделителей между функциями, объявлениями переменных, логически разделёнными блоками кода и т.п., не допускается.

1.3. Представление текста

Для представления содержимого текстовых файлов, в том числе файлов с исходным кодом, должна использоваться кодировка UTF-8. Необходимость использования маркера последовательности байтов (BOM) не регламентируется. Перевод строки должен быть в стиле Unix (`\n`).

1.4. Ширина строк

Ширина строк исходного кода не должна превышать 80 символов.

1.5. Расстановка отступов

Для отступов используются пробелы, использование табуляции запрещено.

Отступы не используются для выравнивания по одному столбцу идентификаторов и операторов, находящихся на нескольких строках. Использование отступов для выравнивания однострочных комментариев, расположенных на соседних строках, по одному столбцу допускается.

1.5.1. Сокращение уровня отступов

Каждый фрагмент исходного кода должен располагаться на как можно меньшем уровне отступов. В противном случае ограничение на ширину строк (см. раздел 1.4) может привести к излишней фрагментации чрезмерно длинных выражений.

1.5.2. Оформление выражений, занимающих несколько строк

Если выражение невозможно разместить на одной строке, оно должно быть разбито на несколько строк. Все строки, кроме первой, должны располагаться на большем на единицу уровне отступов относительно первой строки.

Оператор, по которому производится разбиение, должен размещаться в начале новой строки.

1.6. Расстановка пробелов

1.6.1. Пробелы в списках

Разделённые запятой элементы списков должны быть дополнительно отделены друг от друга одним пробелом. Запятая должна быть прижата к элементу, за которым она следует, пробел должен быть прижат к следующему элементу списка.

1.6.2. Пробелы на концах строк

Пробел или символ табуляции не могут быть последними символами в строке.

1.7. Расстановка пустых строк

Использование более одной пустой строки подряд не допускается. В конце файла должна быть ровно одна пустая строка.

2. Правила оформления исходного кода на языке C

2.1. Оформление исходного кода

В данной части документа приведены правила оформления исходного кода на языке C (стандарты ISO/IEC 9899:1999[c99-std], ISO/IEC 9899:2011[c11-std], ISO/IEC 9899:2018[c18-std]).

Демонстрационная программа, оформленная в соответствии с положениями настоящей части, приведена в приложении A.

2.1.1. Комментирование

В многострочных комментариях в стиле C89, не являющихся комментариями Doxygen, не следует помещать "звёздочку" в начале каждой строки комментария.

2.1.1.1. Комментирование глобальных и локальных констант и переменных

Назначение локальных и глобальных констант и переменных должно быть документировано. Желательно размещение краткого комментария на одной строке с объявлением константы или переменной, однако при необходимости комментарий может занимать несколько строк; во втором случае он должен размещаться непосредственно над объявлением константы или переменной.

2.1.1.2. Комментарии Doxygen

Для построения технической документации следует использовать средство Doxygen[doxygen-offsite]. В коде на C следует использовать стиль комментариев JavaDoc.

Комментарии Doxygen используются для документирования сущностей, которые становятся символами в исполняемом коде. Кроме того, они документируют различные области видимости в коде на C (область файлов, область функций, область блоков внутри функций).

1. Комментирование файлов. Каждый файл, содержащий исходный код, должен начинаться с комментария, кратко описывающего назначение файла. При необходимости, комментарий должен содержать также и развёрнутое описание реализованного в файле алгоритма, формата (протокола) обрабатываемых данных и т.п.
2. Комментирование структур. Каждой структуре должен соответствовать комментарий, описывающий её назначение.
3. Комментирование функций и макроопределений. Комментарии к функциям и макроопределениям должны содержать, по крайней мере, краткое описание символа или макроопределения и описание возвращаемого значения и принимаемых аргументов, если таковые существуют. Для аргументов следует помечать, являются ли передаваемые в них значения входными или выходными по отношению к функции (макроопределению).

4. Комментирование полей структур. Назначение каждого поля структуры должно быть документировано. Комментарии к полям структур должны удовлетворять тем же требованиям, что и комментарии к константам и переменным (см. раздел 2.1.1.1).

Символы и макроопределения с разной областью видимости должны быть документированы в различных местах. Символы (функции, константы и переменные) и макроопределения, составляющие публичный ИПП, должны быть документированы в подключаемых файлах объявлений. Символы и макроопределения, предназначенные для использования в пределах единицы трансляции или иным образом являющиеся внутренними по отношению к проекту, должны быть документированы в месте определения.

2.1.2. Подключение файлов объявлений

Файлы объявлений должны подключаться только по необходимости, если их игнорирование не позволяет собрать проект, и использование упреждающего объявления невозможно. В противном случае всюду, где возможно, должны использоваться упреждающие объявления.

Все используемые проектом символы должны быть объявлены явно. Современные компиляторы позволяют использовать вызовы стандартной библиотеки (например, `printf()`) без их объявления (подключения файла `stdio.h`). Не следует полагаться на это поведение.

Все директивы `#include` должны располагаться в начале файла, сразу после комментария Doxygen уровня файла(см. раздел 2.1.1.2). Если подключение происходит в файле объявлений, директивы `#include` должны помещаться после защищающего макроопределения (см. раздел 2.3.1). Файлы должны подключаться в следующем порядке:

1. файл `config.h` (имя файла может быть произвольным), создаваемый системой сборки (если необходимо);
2. файл объявлений, соответствующий текущему файлу реализации;
3. другие файлы определений из текущего проекта;
4. файлы определений стандартной библиотеки C;
5. файлы определений операционной системы;
6. файлы определений сторонних библиотек.

Все группы директив подключения файлов должны быть отделены друг от друга одной пустой строкой. Файлы в каждой группе удобно подключать в алфавитном порядке.

Подключение файла определений, соответствующего текущему файлу реализации, показывает, что в подключаемом файле нет скрытых зависимостей, не указанных в этом файле явно. Кроме того, это однозначно показывает, что файл, подключаемый первым, является файлом объявлений для текущего файла реализации.

2.1.3. Расстановка отступов

2.1.3.1. Общее правило

Для отступов в исходном коде используется два пробела.

2.1.3.2. Оформление меток

Метка должна находиться на первом знакоместе в строке. Между именем метки и следующим за ним двоеточием пробел не ставится. Метка должна быть единственным выражением в строке, на которой она находится, кроме случаев, когда непосредственно за ней следует пустое выражение. Пример оформления метки приведён в Листинг 1.

Листинг 1. Пример оформления метки

```
1 void
2 foo()
3 {
4   bar();
5   baz();
6  hurr:
7   durr();
8 }
```

2.1.3.3. Оформление оператора switch

Код в теле оператора `switch`, находящийся после каждой метки `case`, должен начинаться с новой строки и располагаться на большем на единицу уровне отступов относительно метки. Метки должны находиться на уровне отступов, на единицу большем относительно ключевого слова `switch` и следующего за ним условия.

Участки кода, следующие за каждой меткой, могут быть отделены одной пустой строкой от имени следующей метки.

В случае, если фрагмент кода, следующего за меткой, должен находиться во вложенной области видимости, он оформляется в соответствии с общими требованиями (см. раздел 2.1.6) следующим образом:

1. открывающая и закрывающая фигурные скобки должны располагаться в одном столбце и быть единственным символом в строке, помимо предваряющих пробелов, определяющих уровень отступов;
2. ключевое слово `break`, если его использование необходимо, располагается на строке, непосредственно следующей за строкой, на которой располагается закрывающая фигурная скобка.

Использование метки `default` в операторе является обязательным. В случае, если метка `default` теоретически недостижима, под ней помещается комментарий и (или) специфичная для

компилятора конструкция, указывающие на её недостижимость.

В случае, когда после фрагмента кода, следующего за меткой, не требуется прерывать выполнение оператора `switch` (т.н. "fallthrough"), необходимо поместить на следующей строке комментарий, содержащий единственное слово `fallthrough`.

В случае, если фрагмент кода, следующего за меткой, представлен в виде единственного тривиального выражения, это выражение и следующее за ним ключевое слово, изменяющее направление потока управления (`break`, `continue`, `goto`, `return`), при наличии такового, разрешается размещать на одной строке с меткой (см. Листинг 2). В остальных случаях метка должна быть единственным выражением в строке, на которой она находится (см. Листинг 3).

Листинг 2. Пример оформления оператора `switch` с единственными тривиальными выражениями

```
switch (condition1) {
  case foo: a = 4; break;
  case bar: b = c + d; break;
  default: __builtin_unreachable();
}
```

Листинг 3. Пример оформления оператора `switch`

```
switch (condition2) {
  case foo:
    {
      int a = 4;
    }
    break;

  case bar:
    b = c + d;
    // fallthrough

  case baz:
    break;

  default:;
    // Недостижимая ветка
}
```

Рекомендации по формулированию условий: см. раздел 2.3.8.

2.1.4. Расстановка пробелов

2.1.4.1. Пробелы внутри и снаружи круглых скобок

Пробелы перед скобками должны ставиться в операторах и не должны ставиться в вызовах, определениях и объявлениях функций и макросов. Это позволяет отличать несколько вызовов функций друг от друга и их аргументов в вызовах функций или сложных выражениях, которые могут включать в себя также арифметические, логические или битовые операторы.

Пробелы не должны помещаться непосредственно после открывающей и перед закрывающей скобками. Пример расстановки пробелов относительно круглых скобок приведён в Листинг 4.

Листинг 4. Пример расстановки пробелов относительно круглых скобок

```
1 // Корректное форматирование
2 while (true)
3   a = foo(arg1, arg2);
4
5 // Некорректное форматирование
6 while (false)
7 {
8   c = bar (arg3, arg4);
9   e = baz( arg5, arg6 );
10 }
```

2.1.4.2. Пробелы в объявлении указателей

При объявлении указателя символ "звёздочка" должен быть прижат к имени указателя; символ не должен быть прижат к типу указателя или отделяться от имени и типа пробелами. Это улучшает читаемость кода в ситуации, когда переменна и указатель одного типа объявлены на одной строке.

При объявлении константного указателя (в выражениях вида `int * const ptr;`) символ "звёздочка" должен быть отделён пробелом и от типа значения, для которого объявляется указатель, и от ключевого слова `const`. Пример расстановки пробелов в объявлениях указателей приведен в Листинг 5.

Листинг 5. Пример расстановки пробелов в объявлениях указателей

```
1 void
2 foo(struct bar *arg)
3 {
4   /* --------vvvvv ------- указатель */
5   struct baz *var1, var2;
6   /* ---------------^^^^-- значение */
7 }
```

2.1.4.3. Пробелы вокруг арифметических, битовых и логических операторов, операторов присваивания и сравнения

Арифметические, битовые и логические операторы, операторы присваивания и сравнения должны быть отделены одним пробелом с каждой стороны от окружающего их кода. Не следует отделять операторы логического отрицания и дополнения от выражения, к которому они относятся.

2.1.4.4. Пробелы в комментариях

Обозначение начала и конца комментария (`/ и /`, `//` или обозначение комментария Doxygen) должно быть отделено от текста комментария одним пробелом.

2.1.5. Расстановка пустых строк

2.1.5.1. Пустые строки вокруг определений функций

Объявления функций должны быть отделены друг от друга одной пустой строкой. Необходимость использования для разделения более чем одной строки или других, дополнительных разделителей является признаком плохой логической структуризации частей проекта. ИСР предоставляют средства навигации в исходном коде, устраняя потребность в явном визуальном разделении объявлений функций.

2.1.5.2. Пустые строки между логически разделёнными блоками кода

Размещение в теле одной функции логически различных блоков кода может свидетельствовать о необходимости реорганизации этой функции. Тем не менее, такая реорганизация нежелательна.

2.1.6. Расстановка фигурных скобок

Фигурные скобки, ограничивающие тело функции, должны располагаться в одном столбце и быть единственным символом в строке. Фигурные скобки, ограничивающие блок кода в функции (создающие вложенную область видимости), должны располагаться в одном столбце и быть единственным символом в строке, за исключением предваряющих пробелов, определяющих уровень отступов, и за исключением случаев, когда блок кода должен выполняться между выполнением двух логически связанных операций, например, захвата и освобождения мьютекса. В остальных случаях открывающая фигурная скобка должна располагаться на той же строке, что и предваряющий её код, и отделяться от него одним пробелом; закрывающая скобка должна располагаться на новой строке и находиться на том же уровне отступов, что и оператор или начало определения, к которому она относится. Если за закрывающей скобкой не следует ключевое слово `else` или `while` (в операторе `do while`), она должна быть единственным символом в строке за исключением предваряющих пробелов. Предпочтительный и нежелательный варианты расстановки фигурных скобок, ограничивающих тело оператора, показаны в Листинг 6 и Листинг 7 соответственно.

Листинг 6. Предпочтительный вариант расстановки фигурных скобок

```
1 if (some_condition) {
2   some_function_call();
3   x = y;
4 }
```

Листинг 7. Нежелательный вариант расстановки фигурных скобок

```
1 if (some_condition)
2 {
3   some_function_call();
4   x = y;
5 }
```

Также рекомендуется вводить вложенные области видимости для кода, выполняющегося между захватом и освобождением экземпляра какого-либо примитива синхронизации.

2.1.7. Объявление численных констант

В случаях, когда невозможно однозначно определить точный тип численной константы, следует использовать определённые стандартом языка уточняющие суффиксы.

Значение с плавающей запятой, записанное десятичной дробью, должно иметь целую и дробную часть, даже если одна из них равна нулю.

Для обозначения нулевого указателя используется константа `NULL`. Использование численной константы `0` не допускается. Пример объявления численных констант приведён в Листинг 8.

Листинг 8. Пример объявления численных констант

```
 1 int i = 0;
 2 unsigned int ui = 0U;
 3 long int li = 0L;
 4 unsigned long int uli = 0UL;
 5 long long int lli = 0LL;
 6 unsigned long long int ulli = 0ULL;
 7 float f = 0.0F;
 8 double d = 0.0;
 9 long double ld = 0.0L;
10 void *p = NULL;
```

2.1.8. Объявление функций

При объявлении функций, предназначенных для внутреннего использования в единице трансляции, нет необходимости в указании в их прототипах имён принимаемых аргументов.

Объявления таких функций должны быть как можно более общими. Имена аргументов расцениваются как особенности реализации, однако прототип функции не является частью её реализации.

Имена аргументов могут быть приведены для функций, составляющих "публичный API". Это позволяет комментариям Doxygen быть более точными, кроме того, такие объявления функций в этом случае становятся одним из видов документации для пользователей API. Правила оформления комментариев Doxygen: см. раздел 2.1.1.2.

2.2. Разработка с учётом особенностей инструментальных средств

Буквальное следование всем правилам из текущего и следующего подразделов невозможно; допускается отходить от них там, где это имеет смысл.

2.2.1. Работоспособность на поддерживаемых платформах

Должна существовать возможность собрать любое фиксированное состояние исходного кода, извлечённое из репозитория системы управления исходным кодом, любым компилятором, для которого в проекте оговорена поддержка.

Необходимо тестирование разрабатываемых программ на всех платформах (уникальных сочетаниях набора команд процессора и двоичного программного интерфейса вычислительной системы), для которых оговорена возможность их использования. На всех платформах разрабатываемое ПО должно быть полностью работоспособно.

2.2.2. Обработка предупреждений компилятора и средств статического анализа

Предупреждения компилятора указывают на недостатки в исходном коде. Несмотря на то, что код может быть корректным с точки зрения синтаксиса C, наличие предупреждений компилятора говорит о том, что код может вести себя не так, как ожидалось, или не следовать общепринятым практикам программирования.

Невозможно избежать предупреждений ото всех используемых в разработке компиляторов, однако количество этих предупреждений должно быть сведено к минимуму.

Не следует отключать предупреждения компилятора с помощью директив `#pragma` или передаваемых компилятору параметров. При появлении такой необходимости предложение должно быть рассмотрено коллегиально и, в случае его принятия, исключение должно быть добавлено в систему сборки проекта (если применяется для единицы трансляции целиком), либо применяться к минимально необходимому фрагменту кода в единице трансляции.

При текущей работе над проектом необходимо включить как можно больше предупреждений компилятора и, возможно, даже расценивать их как ошибки. Следует помнить, что по мере

развития компиляторов набор предупреждений изменяется.

При текущей работе над проектом необходимо применять средства статического анализа исходного кода, предоставляемые используемыми компиляторами. По возможности следует также использовать сторонние средства анализа. Необходимо применять средства динамического анализа исходного кода для выявления ошибок управления памятью, ошибок межпоточного взаимодействия, случаев неопределённого поведения.

Всегда следует проверять, не вызывает ли предупреждений (за исключением ложных срабатываний) вновь добавленный код, перед регистрацией его в системе управления исходным кодом. Новые предупреждения должны расцениваться как вызванные регрессивными изменениями.

2.3. Сопровождаемость и читаемость исходного кода

2.3.1. Оформление подключаемых файлов объявлений

Содержимое каждого файла объявлений должно быть обрамлено уникальным (в рамках проекта) защищающим макроопределением. Схема именования защищающих макроопределений должна быть предсказуемой и постоянной в пределах всего программного проекта. Пример обрамления подключаемого файла представлен в Листинг 9.

Листинг 9. Пример обрамления подключаемого файла

```
1 #ifndef FILENAME_H
2 #define FILENAME_H
3
4 /* Содержимое файла */
5
6 #endif
```

Объявление в подключаемых файлах следующих сущностей допустимо во всех случаях:

1. структуры данных;
2. объединения;
3. функции.

В подключаемых файлах не могут быть размещены:

1. подключения других файлов объявлений, которые могут быть помещены в соответствующие файлы реализации (см. раздел 2.1.2);
2. определения функций, не являющихся статическими встраиваемыми.

Допустимость объявления в подключаемых файлах следующих сущностей должна рассматриваться для каждого случая в отдельности:

1. перечисления;
2. константы;
3. определения типов данных, заданные с использованием ключевого слова `typedef`;
4. директивы препроцессора и макросы.

2.3.2. Использование предикатных функций вместо предикатных циклов

Циклы, предназначенные для вычисления предикатов, предпочтительно вынести в отдельные функции. Это уменьшит общий уровень вложенности исходного кода, сделает код вычисления пригодным для последующего использования, вынудит присвоить ему имя и описать его работу в комментарии.

2.3.3. Повторное использование небольших фрагментов кода

Для повторного использования небольших фрагментов кода часто применяются макроопределения. Всюду, где это возможно, вместо них следует использовать встраиваемые функции.

2.3.4. Константность переменных и аргументов функций

Любое значение, не изменяющееся во всей области его видимости, должно быть объявлено константным. Это правило должно быть применено к передаваемым в функции указателям и, при необходимости, значениям, на которые они ссылаются.

2.3.5. Экспорт символов

Следует воздерживаться от явного или неявного экспорта символов, предназначенных только для использования внутри одной единицы трансляции, равно как и их публичного объявления.

Предназначенные для внутреннего использования функции должны быть объявлены с модификатором `static`. Отсутствие такового модификатора должно расцениваться как явное публичное объявление функции в составе ИПП разрабатываемого программного проекта, влекущее за собой возможность вызывать функцию из других единиц трансляции и исполняемых файлов, скомпонованных с содержащим функцию исполняемым объектом.

При именовании подлежащих экспорту символов следует использовать префикс, единый в рамках проекта и однозначно указывающий на принадлежность символа проекту, например, `phoenix_init()`, `phoenix_shutdown()`. Данное требование обусловлено, в первую очередь, необходимостью предотвращения коллизий символов, импортированных из различных исполняемых объектов, во время компоновки или исполнения.

2.3.6. Раннее перенаправление потока управления

При написании нового программного кода следует учитывать, как много состояний и

предыдущих решений нужно будет помнить при последующем чтении этого кода. Сравнительно большой фрагмент кода со сложными условными операторами и ключевыми словами, изменяющими направление потока управления, может быть сложным для восприятия, и восстановление описываемого им алгоритма может требовать значительных усилий. Кроме того, такой код сложно комментировать из-за высокого уровня вложенности одной части конструкций и подверженности часты изменениям - другой. Из-за ограничения на длину строки и высокого уровня вложенности некоторые конструкции могут быть разорваны на большое число коротких строк. При чтении такого фрагмента возникнет необходимость запоминания нескольких контекстов и своевременного переключения между ними.

Как можно более ранний выход из тела цикла или условного оператора или возврат из функции позволяет уменьшить общий уровень вложенности исходного кода; упрощает описание инвариантов; показывает читающему код разработчику, что в определённом месте рассматриваемого фрагмента не будет ещё ветви условного оператора, для которой также пришлось бы запоминать контекст.

Не следует использовать ветвление с помощью ключевых слов `else` или `else if` после ключевых слов `return`, `break`, `continue`, `goto`, изменяющих направление потока управления.

2.3.7. Программирование циклов

2.3.7.1. Вычисление условий выхода из цикла

Следует избегать вычислений в операторе цикла. Современные компиляторы C, в общем случае, способны оптимизировать такие конструкции и размещать вычисление непосредственно перед выполнением оператора, однако полагаться на такое поведение нельзя. Более того, подобные конструкции могут навести читающего код разработчика на мысль, что условие выхода из цикла изменяется на каждой итерации, когда в действительности это не так; в результате весь фрагмент кода может быть понят неверно.

Если условие выхода из цикла не должно меняться во время выполнения цикла, следует вычислить его один раз перед выполнением оператора цикла. В противном случае следует явно указать в комментарии, что вычисление условия выхода из цикла на каждой итерации выполняется намеренно. В общем случае рекомендуется реорганизовать фрагмент кода, содержащий такой цикл, таким образом, чтобы необходимость вычисления условия выхода непосредственно в операторе цикла не возникала. Пример вычисления значения в операторе цикла, которого можно избежать, приведён в Листинг 10.

Листинг 10. Пример вычисления значения в операторе цикла, которого можно избежать

```
#define NELEMS(a) (sizeof(a) / sizeof(a[0]))

bool
test_something(void);

void
do_something(void);

size_t
nzeroes(int *a, size_t s)
{
  size_t r = 0;
  for (size_t i = 0; i < s; ++i)
    if (0 == a[i])
      ++r;
  return r;
}

void
loop_wrapper(void)
{
  int a[] = { 45, 0, 0, 23, 59 };
  for (size_t i = 0; i < nzeroes(a, NELEMS(a)); ++i)
    if (test_something())
      do_something();
}
```

2.3.7.2. Выход из цикла с переходом через несколько уровней вложенности

При необходимости передать поток управления из вложенного цикла на несколько уровней выше предпочтительнее использовать немедленный выход из функции или оператор `goto`, чем выстраивать во внешних циклах сложную систему проверки значений флагов-индикаторов, указывающих на необходимость выхода из цикла. В данном случае использование оператора `goto` действительно повышает читаемость и надёжность программного кода.

2.3.8. Программирование условий

Следует избегать использования присваивания в условном операторе, по крайней мере, на верхнем уровне.

При проверке на равенство с константой в условном операторе константу удобно располагать слева от оператора сравнения. Это прервёт компиляцию в случае, если вместо оператора сравнения по ошибке будет записан оператор присваивания.

При проверке некоторого значения на равенство `NULL`, не `NULL`, `true` или `false` следует явно указывать значение, с которым производится сравнение, если его тип отличен от `bool`. В противном случае при проверке возвращённого функцией значения невозможно однозначно определить, какому типу принадлежит это значение, не обращаясь к прототипу функции. Данное требование показано в Листинг 11.

Листинг 11. Пример раннего возврата потока управления из функции:

```
void
foo(bool a, int b)
{
  if (a || 0 != b)
    return;
  /* ... */
  if (!bar() || 0 == baz() || NULL == buz())
    /* bar() возвращает булево значение, baz() - целочисленное,
       buz() - указатель */
    return;
}
```

3. Правила оформления исходного кода на языке SystemVerilog

3.1. Оформление исходного кода

Исходный код должен разрабатываться с использованием стандарта IEEE Std 1800-2017[sv-std]. Допускается в случаях необходимости использовать более ранние стандарты SystemVerilog.

Демонстрационная программа, оформленная в соответствии с положениями настоящей части, приведена в приложении В.

3.1.1. Расстановка отступов

Значение одного отступа равно четырем пробелам.

Исходный код, заключённый между следующими парами ключевых слов, должен располагаться на большем на единицу уровне отступов:

- `begin/end`;
- `package/endpackage`;
- `class/endclass`;
- `function/endfunction`;
- `task/endtask`;
- `case/endcase`;

3.1.2. Расстановка пробелов

3.1.2.1. Пробелы вокруг операторов

Операторы присваивания (`assignment operator`), операторы условия (`conditional expression`), бинарные операторы (`binary operator`), потоковые операторы (`stream operator`) должны быть отделены одним пробелом с каждой стороны от окружающего их кода.

Унарные операторы (`unary operator`) и операторы инкремента/декремента (`increment, decrement operator`) не следует отделять от выражения, к которому они относятся.

3.1.2.2. Оформление пробелов при объявлении вектора и массива

При объявлении вектора или массива допускается не ставить пробелы вокруг арифметических операторов.

Пробелы должны ставиться вокруг упакованных размерностей. Вокруг неупакованных размерностей пробелы не ставятся. Пробелы между размерностями не ставятся.

Листинг 12. Пример объявления вектора и массива

```
/* упакованная размерность --------- */
/* ---vvvvvvvvvvv------------------- */
logic [WIDTH-1:0] foo_0[0:DEPTH-1];
/* -------------------^^^^^^^^^^^-- */
/* ------- неупакованная размерность */
logic [WIDTH - 1:0] foo_1[0:DEPTH - 1];
logic [WIDTH - 1 : 0] foo_2[0 : DEPTH - 1];
logic [WIDTH-1 -: 0] foo_3[0:DEPTH-1];
logic [WIDTH_1-1:0][WIDTH_2-1:0] foo_4[0:DEPTH_1-1][0:DEPTH_2-1].
```

Примечание: подробнее об упакованных и неупакованных массивах смотри раздел 7.4 IEEE Std 1800-2017[sv-std].

3.1.2.3. Оформление метки

При оформлении метки блока кода вокруг `:` ставятся пробелы.

Листинг 13. Пример оформлении метки блока кода

```
begin : foo
end : foo
```

3.1.2.4. Оформление метки в операторе case

При оформлении метки в операторе `case` между меткой и `:` пробел не ставится.

Листинг 14. Пример объявления метки в операторе `case`

```
case ({sel, data})
    2'b00: begin
        im <= -SYM_IM_60;
        re <= SYM_RE_60;
    end
    2'b01: begin
        im <= SYM_IM_60;
        re <= SYM_RE_60;
    end
    2'b10: begin
        im <= -SYM_IM_90;
        re <= SYM_RE_90;
    end
    2'b11: begin
        im <= SYM_IM_90;
        re <= SYM_RE_90;
    end
endcase
```

3.1.2.5. Оформление вызова функции, задачи и макроса

При вызове функции, задачи или макроса между именем функции и открывающей скобкой пробелов ставится не должно.

Листинг 15. Пример вызова функции

```
foo(variable1);
```

3.1.2.6. Оформление пробелов вокруг ключевых слов

Пробелы должны ставиться до и после ключевых слов.

Пробелы не ставятся в следующих случаях:

- перед ключевыми словами, которые следуют сразу за открывающей группой, например, открывающей скобкой;
- перед ключевым словом в начале строки;
- после ключевого слова в конце строки.

Листинг 16. Пример оформления пробелов вокруг ключевых слов

```
if (condition1) begin
    ...
end

always_ff @(posedge clk) begin
    ...
end
```

3.1.3. Оформление скобок

Несмотря на то, что в SystemVerilog существует приоритет выполнения операторов, рекомендуется ставить скобки для однозначного определения порядка выполнения операций.

3.1.3.1. Оформление скобок в тернарных выражениях

Вложенные тернарные выражения должны быть заключены в скобки.

Листинг 17. Пример оформления вложенных тернарных выражений

```
assign foo = condition_1 ? (condition_2 ? x : y) : z;
```

3.1.4. Объявление сигналов

Сигналы должны быть объявлены до их использования. Неявные объявления сигналов запрещены.

3.2. Именование

3.2.1. Константы

При объявлении константы её тип должен быть указан явно.

В случае использования одной и той же константы в пределах проекта рекомендуется выделить её в пакет. Константы, используемые в пределах одного модуля, объявляются в данном модуле.

Имена констант должны быть написаны заглавными буквами с подчеркиванием.

3.2.2. Параметры

При объявлении параметра его тип должен быть указан явно.

Для параметризации модулей, интерфейсов и т.д. должно использоваться ключевое слово

`parameter`, для объявления локальной константы и производных параметров — `localparam`.

`` `define `` и `defparam` никогда не должны использоваться для параметризации модуля.

Имена параметров должны быть написаны заглавными буквами с подчеркиванием.

Листинг 18. Пример оформления параметров модуля

```
module fft4 #(
    parameter int IN_DATA_W = 16,
    parameter int DIR = 1,

    localparam int DATA_N = 4,
    localparam int STAGE1_W = IN_DATA_W + 1,
    localparam int OUT_DATA_W = STAGE1_W + 1
)(
    input logic clk,
    input logic rst,
    input logic ival,
    input logic [IN_DATA_W-1:0] idata_re[0:DATA_N-1],
    input logic [IN_DATA_W-1:0] idata_im[0:DATA_N-1],

    output logic oval,
    output logic [OUT_DATA_W-1:0] odata_re[0:DATA_N-1],
    output logic [OUT_DATA_W-1:0] odata_im[0:DATA_N-1]
);
```

3.2.3. Макросы

Имена макросов должны быть написаны заглавными буквами с символом подчеркивания.

При глобальном определении макроса рекомендуется помещать его в заголовочный файл, содержащий все макросы проекта.

При локальном определении макроса он используется только рамках одного локального файла. Он должен быть явно отменён после использования, чтобы избежать загрязнения глобального пространства имен макросов. Чтобы указать, что макрос предназначен только для использования в локальной области, имя макроса должно начинаться с одного подчеркивания.

Листинг 19. Пример определения локального макроса

```
`define _MAKE_THING(_x) \
    thing i_thing_##_x (.clk(clk), .i(i##_x), .o(o##_x));
`_MAKE_THING(a)
`_MAKE_THING(b)
`_MAKE_THING(c)
`undef _MAKE_THING
```

3.2.4. Суффиксы

Суффиксы используются для уточнения при описании объектов. В следующей таблице перечислены суффиксы и их значения.

Суффикс(-ы)	Описание
_n	Активный низкий сигнал
_n, _p	Дифференциальная пара, активный низкий и высокий уровни
_d	Конвейерная версия сигналов

3.2.5. Перечисления

Имена перечислений должны быть написаны строчными буквами с подчеркиваниями. Имена значений перечисления должны быть написаны заглавными буквами с символом подчеркивания.

Листинг 20. Пример оформления перечислений

```
enum logic [1:0] {
    READ = 2'b01,
    WRITE = 2'b10
} state;
```

3.2.6. Имена сигналов

Под сигналом понимается сеть, регистр, порт, переменная.

Имена сигналов должны быть написаны строчными буквами с подчеркиваниями. Исключение составляют порты модулей, которые непосредственно подключаются к выводам микросхемы. Они должны быть написаны заглавными буквами с подчеркиваниями.

Листинг 21. Пример оформления модуля, порты которого подключаются к выводам микросхемы

```
module ultra_long_fft_test_on_fmc132p #(
    localparam int PCIE_LINK_W = 8, // Разрядность сигнальных пар PCI-e
    localparam int LED_N = 4 // Количество светодиодов на FMC132p
)(
    input logic PCIE_CLK_P,
    input logic PCIE_CLK_N,
    input logic PCIE_RST_N,
    input logic [PCIE_LINK_W-1:0] PCIE_RX_P,
    input logic [PCIE_LINK_W-1:0] PCIE_RX_N,

    output logic [PCIE_LINK_W-1:0] PCIE_TX_P,
    output logic [PCIE_LINK_W-1:0] PCIE_TX_N,
    output logic FPGA_LED[0:LED_N-1]
);
```

3.2.6.1. Префиксы

Необходимо использовать общие префиксы, чтобы идентифицировать группы сигналов, объединенных по функциональному назначению.

3.2.6.2. Именование сигналов тактирования

Сигналы тактирования должны соответствовать следующему виду:

- `[*_]clk`,
- `[*_]clkin`,
- `[*_]clock[_*]`,
- `[*_]aclk`,
- `[*_]aclkin`.

Примечание: `[*_]` и `[_*]` необязательны, а `*` соответствует любому тексту. В именах не учитывается регистр.

3.2.6.3. Именование дифференциальных сигналов тактирования

Дифференциальные сигналы тактирования должны соответствовать следующему виду:

- `[*_]clk_p`,
- `[*_]clk_n`.

Примечание: `[*_]` и `[_*]` необязательны, а `*` соответствует любому тексту. В именах не учитывается регистр.

3.2.6.4. Именование сигналов сброса

Сигналы сброса должны соответствовать следующему виду:

- `[*_]aresetn`,
- `[*_]axi_resetn`,
- `[*_]reset[_*]`,
- `[*_]resetin`,
- `[*_]resetn`,
- `[*_]rst`,
- `[*_]rst_n`,
- `[*_]rstin`,
- `[*_]rstn`.

Примечание: `[*_]` и `[_*]` необязательны, а `*` соответствует любому тексту. Также обратите внимание, что в именах не учитывается регистр.

3.2.6.5. Именование сигналов AXI

Имена интерфейсов AXI должны состоять из имени интерфейса, за которым следует имя сигнала AXI. Для интерфейса можно использовать любое имя, если оно согласовано для каждого порта.

Таким образом, интерфейсы AXI4, AXI4-Lite должны иметь следующий вид `<interface_name>_axi_<signal_name>`, а интерфейсы AXI4-Stream - `<interface_name>_axis_<signal_name>`.

Примечание: В именах не учитывается регистр.

Список сигналов интерфейса AXI4 и AXI4-Lite приведен в AMBA AXI and ACE Protocol Specification[axi-spec].

Список сигналов интерфейса AXI4-Stream приведен в AMBA AXI-Stream Protocol Specification[axis-spec].

3.3. Структура модуля

Общий порядок описания модуля должен соответствовать следующему порядку:

1. заголовок модуля с описанием параметров портов;
2. секция описания локальных параметров;
3. секция описания функций, задач, классов;
4. секция описания сигналов;

5. секция постоянных назначений;
6. секция экземпляров включённых модулей;
7. секция процедурных блоков.

После каждой секции следует пустая строка.

Отклонения от данной структуры допускаются в двух случаях:

1. описание тривиальных постоянных назначений входным сигналам, связанных с экземпляром модуля, допускается размещать сразу за описанием соответствующего экземпляра модуля;
2. процесс, формирующий элементарные входные сигналы для модуля, как то сигналы адреса для памяти, сигналы записи/чтения и т.д., допускается размещать также сразу после описания экземпляра модуля.

3.3.1. Объявление модуля

При объявлении модуля должен использования стиль Verilog-2001. Стиль Verilog-95 запрещен.

Объявление порта в модуле должно полностью объявлять его имя, тип и направление.

Открывающая скобка должна быть в той же строке, что и объявление модуля, а первый порт должен быть объявлен в следующей строке.

Закрывающая скобка должна быть на отдельной строке в нулевом столбце.

Отступ для объявления модуля следует стандартному правилу отступа.

При объявлении портов следует придерживаться следующей последовательности:

1. порты тактирования;
2. порты сброса;
3. входные порты;
4. выходные порты.

Входные и выходные порты разделяются одной пустой строкой.

Допускает разделять порты не по входу и выходу, а по назначению, например, на ведущий интерфейс и ведомый.

Листинг 22. Пример описания модуля

```
module nco #(
    parameter int PHASE_WIDTH = 14,
    parameter int OUTPUT_WIDTH = 18,
    parameter int DEPTH = 14,
    parameter string ACC_USE_DSP = "yes",
    parameter string ROM_STYLE = "block"
)(
    input logic clk,
    input logic rst,
    input logic ival,
    input logic signed [PHASE_WIDTH-1:0] iphase_inc, // Инкремент фазы
    input logic signed [PHASE_WIDTH-1:0] iphase_offset, // Сдвиг фазы

    output logic oval,
    output logic signed [OUTPUT_WIDTH-1:0] osin, // Значение синуса
    output logic signed [OUTPUT_WIDTH-1:0] ocos // Значение косинуса
);
```

3.4. Оформление операторов

3.4.1. Оформление оператора `begin`/`end`

Ключевое слово `begin` должно находиться в той же строке, что и предыдущее ключевое слово, и завершает строку. `end` должен начинать новую строку.

3.4.2. Оформление оператора условий

В операторе условия ключевое слово `else` должно начинаться с новой строки.

3.4.3. Оформление оператора `case`

Код в теле оператора `case`, находящийся после каждой метки, должен начинаться с новой строки и располагаться на большем на единицу уровне отступов относительно метки.

Участки кода, следующие за каждой меткой, могу быть отделены одной пустой строкой от имени следующей метки.

4. Правила оформления исходного кода на языке VHDL-93/200X

Единица измерения отступа равна 2 или 4 пробелам.

4.1. Комментирование

4.1.1. Общие положения

Исходный код должен быть исчерпывающе документирован. Исходный код и комментарии должны вместе создавать связную структуру и читаться как единый текст, уточняя, поясняя и иллюстрируя друг друга.

Комментарии должны быть оформлены в виде прозы на русском языке: в них должны соблюдаться структура предложений, правила орфографии и пунктуации, грамматика, правила использования прописных и строчных букв, аббревиатур, и т.д.

4.1.1.1. Комментирование глобальных и локальных параметров, регистров

Назначение локальных и глобальных параметров должно быть документировано. Желательно размещение краткого комментария на одной строке с объявлением параметра, однако при необходимости комментарий может занимать несколько строк; во втором случае он должен размещаться непосредственно над объявлением параметра или регистра.

4.1.2. Общая структура описания модуля и пакета

Общий порядок описания модуля должен соответствовать следующему порядку:

1. блок описания зависимостей;
2. блок описания сущности (`entity`);
    a. декларативная часть описания сущности;
        i. секция описания наследуемых параметров модуля (`generic`);
        ii. секция описания интерфейса (`port`);
        iii. секция описания констант;
        iv. секция объявления локальных функций и процедур;
    b. исполняемая часть описания сущности;
3. блок описания архитектуры;
    a. декларативная часть описания архитектуры;
        i. секция описания глобальных констант;
        ii. секция описания используемых модулей;

iii. секция описания локальных функций/процедур;

iv. секция объявления локальных сигналов модуля;

b. исполняемая часть описания архитектуры;

i. секция описания подключений сигналов;

ii. секция создания и описания экземпляров используемых модулей;

Общий порядок описания пакета должен соответствовать следующему порядку:

1. блок описания зависимостей;
2. блок объявления пакета (`package`);
   a. секция описания параметров блока (`generic`);
   b. секция объявления констант;
   c. секция объявления функций/процедур;
3. блок описания пакета (`package body`);
   a. секция присвоения значений константам;
   b. секция описания функций/процедур;

4.1.3. Блок описания зависимостей

В блоке описания зависимостей присутствует описание используемых библиотек и компонентов библиотек. Для всех строк описания зависимостей отступ равен нулю.

Листинг 23. Пример оформления включения библиотек

```
1 library LibName;
```

Листинг 24. Пример оформления компонентов библиотек

```
1 use LibName.ComponentName.all;
2 use LibName.PackageName.all;
```

Блок описания зависимостей отделяется от предыдущих и последующих записей двумя пустыми строками.

4.1.4. Блок описания сущности

Блок описания сущности состоит из декларативной и исполняемой секции. Между декларативной и исполняемой секциями вставляется пустая строка.

Листинг 25. Пример оформления описания сущности

```
1 entity EntityName is
2     -- declarative part
3 begin
4     -- executable part
5 end EntityName;.
```

Блок описания сущности отделяется от блока описания архитектуры пустой строкой. Отступ строк открытия и закрытия блока описания блока, а также ключевое слово `begin` имеют отступ 0. Для всех внутренних строк минимальный отступ равен 1.

4.1.5. Декларативная секция описания сущности

Декларативная секция сущности состоит из секций описания параметров модуля, секции описания интерфейса модуля, секции объявления констант модуля, секции объявления функций/процедур.

Между секциями описания параметров и интерфейса модуля пустая строка не вставляется. Две вышеперечисленные секции отделяются от последующих пустой строкой. Константы объявляются по одной на каждой строке, без пустых строк, с комментарием к каждой константе. Секция объявления констант, как и объявление каждой функции/процедуры, отделяется пустой строкой.

Листинг 26. Пример оформления декларативной секции описания сущности

```
generic (
    ...
);
Port (
    ...
);

-- constants

-- function_1

...

-- function_N

...

-- procedure_1

...

-- procedure_N
```

Отступ строк объявления констант имеют отступ 1. Открывающие и закрывающие строки для секций описания параметров/портов и объявления функций и процедур имеют отступ 1. Все внешние секции перечисленных блоков имеют минимальный отступ 2.

4.1.6. Секция описания параметров сущности

Заголовок секции описания параметров модуля и открывающая скобка находятся на одной строке. Описание каждого параметра находится на отдельной строке. Внутри блока описания параметров пустые строки допускаются для разделения разных логических групп параметров, при этом необходимо комментирование логической группы.

Листинг 27. Пример оформления секции параметров модуля

```
generic (
    gGenericName_1: type [:= DefaultValue];
    ...
    gGenericName_N: type [:= DefaultValue];
);
```

Открывающая и закрывающая строки имеют отступ 1. Все внутренние строки имеют отступ 2.

4.1.7. Секция описания интерфейса

Заголовок секции описания интерфейса модуля и открывающая скобка находятся на одной строке. Описание каждого порта находится на отдельной строке. При несложной структуре портов блока сначала располагается список входных портов, потом выходных. Списки входных и выходных портов разделяются пустой строкой.

При сложной структуре портов (к примеру наличии нескольких обособленных интерфейсов) для улучшения читаемости следует логически связанные входные/выходные порты группировать в блоки. Блоки рекомендуется разделять пустой строкой.

Листинг 28. Примеры оформления секции описания интерфейса модуля

```
port (
    iInputPortName_1: type [:= DefaultValue];
    ...
    iInputPortName_N: type [:= DefaultValue];

    oInputPortName_1: type [:= DefaultValue];
    ...
    oInputPortName_N: type [:= DefaultValue];
);

port (
    iIface1_input: type [:= DefaultValue];
    iIface1_output: type [:= DefaultValue];

    oIface2_input: type [:= DefaultValue];
    oIface2_output: type [:= DefaultValue];
);
```

Открывающая и закрывающая строки имеют отступ 1. Все внутренние строки имеют отступ 2.

4.1.8. Секция объявления констант сущности

Параметры объявляются общим списком. Использование пустой строки допускается для разделения групп констант.

Листинг 29. Пример оформления секции констант модуля

```
1 -- constants group 1
2 cConstantName_1: type := DefaultValue;
3 ...
4 cConstantName_N: type := DefaultValue;
5
6 -- constants group 2
7 cConstantName_M: type := DefaultValue;
```

Отступ всех строк данной секции равен 1.

4.1.9. Секция объявления функций/процедур

Функции и процедуры используются для свертки кода. Они объявляются общим списком и разделяются пустой строкой. Открывающая скобка находится на строчке с заголовком функции/процедуры. Список входных значений начинается со следующей строки с отступом относительно заголовка, равным 1. Закрывающая скобка находится на отдельной строке.

Листинг 30. Пример оформления секции параметров модуля

```
1 function FunctionName is (
2     [signal] InputName_1: type;
3     [signal] InputName_2: type
4 ) return type;
5
6 procedure ProcedureName (
7     [signal] InputName_1: type;
8     [signal] InputName_2: type
9 );
```

Открывающая и закрывающая строки имеют отступ 1. Все внутренние строки имеют отступ 2.

4.1.10. Исполняемая секция сущности

Исполняемая секция сущности может содержать различные операции по работе с входными сигналами, параметрами и константами. Операции группируются по однородным спискам.

4.1.11. Блок описания архитектуры

Блок описания архитектуры состоит из декларативной и исполняемой части. Ключевое слово `begin` отделяется от обеих частей пустой строкой. Декларативная и исполняемая секции отделяются от сервисных ключевых строк пустой строкой.

Листинг 31. Пример оформления блока описания архитектуры

```
1 architecture ArchitectureName of EntityName is
2     -- декларативная секция архитектуры
3 begin
4     -- исполняемая секция архитектуры
5 end ArchitectureName;
```

Открывающая и закрывающая строки, а также ключевое слово begin имеют отступ 0. Все внутренние строки имеют минимальный отступ 1.

4.1.12. Декларативная часть описания архитектуры

Декларативная часть описания архитектуры может включать секции глобальных констант, описания используемых модулей (компонента), описания локальных функций/процедур, объявление внутренних сигналов модуля.

Константы, относящиеся ко всему модулю, следует выносить в отдельный блок сразу в начале декларативной части. К данному типу констант можно отнести константы, от которых зависит условная компиляция модуля, а также объявление разрядности большого набора сигналов (адреса, данных, и т.д.).

Если объявление глобальной константы в начале декларативной части невозможно по стандарту (например, ей присваивается значение, вычисляемое функцией), то возможно размещение такой константы после блока, от которого она зависит. Но в данном случае рекомендуется вынести часть кода в отдельный пакет, так как согласно стандарту для пакета VHDL действуют правила оформления, отличные от правил оформления архитектуры.

Константы, относящиеся к отдельным сигналам, следует размещать рядом с их объявлением в секции объявления сигналов. К таким константам можно отнести пределы счета счетчиков, отдельные значения, использующиеся при обработке только одного сигнала или малой логически связанной группы сигналов.

Оформление описания используемых модулей совпадает с описанием сущности. Описание прототипов блоков разделяются пустой строкой.

Листинг 32. Пример описания используемого блока

```
component ComponentName is
    generic (
        gGenericName_1: type [:= DefaultValue];
        ...
        gGenericName_N: type [:= DefaultValue]
    );
    port (
        iInputSignal_1: type [:= DefaultValue];
        ...
        iInputSignal_N: type [:= DefaultValue];

        oOutputSignal_1: type [:= DefaultValue];
        ...
        oOutputSignal_M: type [:= DefaultValue]
    );
end component;
```

Открывающая и закрывающая строки имеют отступ 1. Ключевые слова `generic/port` и соответствующие им закрывающие скобки имеют отступ 2. Все вложенные в них строки имеют минимальный отступ 3.

Описание заголовка функций/процедур совпадает их объявлением. Описание логики работы функций/процедур.

Листинг 33. Пример оформления функций и процедур:

```
function FunctionName (
    [signal] InputName: type;
    [signal] InputName: type
) return type is
begin
    -- function logic
    Return [Value];
end FunctionName;

procedure ProcedureName (
    [signal] InputName: type;
    [signal] InputName: type
) is
begin
    -- procedure logic
end ProcedureName;
```

Открывающая и закрывающая строки, закрывающая скобка интерфейса, а также ключевое слово `begin` имеют отступ 1. Все внутренние строки имеют отступ 2.

Секция объявления внутренних сигналов представляет собой список. Объявление каждого сигнала располагается на отдельных строчках. Пустые строки используются для разделения групп сигналов.

Листинг 34. Пример секции объявления сигналов

```
1 signal sSignalName_1: type [:= DefaultValue];
2 ...
3 signal sSignalName_N: type [:= DefaultValue];
```

Все строки имеют отступ 1.

4.1.13. Исполняемая часть архитектуры

Исполняемая часть архитектуры может включать секцию описания подключений сигналов и секцию создания и описания экземпляров используемых модулей. Обе секции отделяются пустой строкой.

Пример оформления исполняемой части архитектуры:

Открывающая и закрывающая строки имеют отступ 0. Все внутренние строки имеют отступ 1.

4.1.14. Секция описания подключения сигналов

Секция описания подключений сигналов может включать постоянные подключения сигналов, использования математических и логических функций языка, а также структуры `generate`, `process`, `loop`.

Листинг 35. Пример оформления секции описания подключения сигналов

```
signal_1 <= signal_2;
...

Signal_2 <= signal_3 + signal_4;
...

Label1:
for i in 0 to n generate

    signal_3 (n-1 downto 0) <= signal_5 (n*2-1 downto n);
    signal_4 (n-1 downto 0) <= signal_5 (n*2-1 downto n);

end generate;

process (wClk)
    variable vVariableName: type [:= DefaultName];
begin

    if iClk'event and iClk = '1' then

        label2: for j in 0 to m loop

            signal_5 (j) <= -signal_6 (j);

        end loop;

    end if;

end process;
```

Все строки имеют базовый отступ 1. Для многострочных блоков правила установки отступа определяются соответствующими разделами данного документа.

4.1.15. Общая структура описания пакета

Оформление секций пакета совпадает с оформлением секций модуля. Различие заключается в ключевых словах открытия и закрытия блоков пакета.

Листинг 36. Пример оформления пакета (общая схема)

```
1  package PackageName is
2
3      -- constants declaration
4
5      -- functions/procedures declaration
6
7  end PackageName;
8
9  package body PackageName is
10
11     -- constants assignment
12
13     -- functions/procedures description
14
15 end PackageName;
```

В пакете может содержаться объявление и описание констант и функций/процедур. Оформление данных секций совпадает с оформлением соответствующих секций в модели.

Для определения отступа строк пакета используются правила описания соответствующих элементов модуля на языке VHDL.

4.2. Оформление элементов и структур языка

4.2.1. Оформление имен

Для имен приняты следующие соглашения:

1. имена входных сигналов интерфейса начинаются с буквы `i` (`iInputSignal`);
2. имена выходных сигналов интерфейса начинаются с буквы `o` (`oOutputSignal`);
3. имена параметров начинаются с буквы `g` (`gWidth`);
4. имена констант начинаются с буквы `c` (`cConstant`);
5. имена функций, процедур, библиотек, пакетов и модулей не регламентированы;
6. использование меток (`label`) не регламентировано и может быть использовано в произвольном месте исходного кода для предоставления дополнительной информации, в том числе для маркировки функциональных областей в HDL-симуляторе.

4.2.2. Оформление операторов

Для языка VHDL определены следующие операторы:

1. оператор `if`;

2. оператор `process`;
3. оператор `generate`;
4. оператор `case`;
5. оператор `loop`.

Для всех операторов в случае небольшого размера секции разделяющие пустые строки не применяются. При увеличении размера секции разделяющие пустые строки могут применяться для улучшения читаемости кода.

Листинг 37. Пример оформления оператора `if`

```
if {condition} then
    -- statements
elsif {condition} then
    -- statements
else
    -- statements
end if;
```

Листинг 38. Пример оформления оператора `process`

```
[ProcessName]: process ({SensetiveList})
    variable vVariableName_1: type;
    ...
    variable vVariableName_N: type;
begin
    -- statements
end process [ProcessName];
```

Листинг 39. Пример оформления оператора `generate`

```
label1: for I in 0 to n generate
    -- statements
end generate;
```

Листинг 40. Пример оформления оператора `case` с простыми условиями

```
[Label]: case {signal} is
    when Value1 =>
        -- statements
    when Value2 =>
        -- statements
    when others =>
        -- statements
end case [Label];
```

Листинг 41. Пример оформления оператора `case` с множественными условиями

```
case {signal} is
    when Value1 | Value2 =>
        -- statements
    when others =>
        -- statements
end case;
```

Листинг 42. Пример оформления оператора `case` целочисленным сигналом в списке чувствительности

```
case integer({signal}) is
    when Value1 [to | downto] Value2 =>
        -- statements
    when others =>
        -- statements
end case;
```

Листинг 43. Пример оформления оператора `loop`

```
[label]: [if/for {condition}] loop
    -- statements
end loop;
```

Для приведенных операторов отступы открывающей и закрывающей строк равны базовому отступу блока. Все внутренние строки имеют отступ, равный базовому, плюс один.

5. Правила оформления исходного кода на языке Python

5.1. Оформление исходного кода

Исходный код на языке Python следует оформлять в соответствии с положениями документа Python Enhancement Proposal 8 - Style Guide for Python Code[pep8], за исключением случаев, особо оговорённых в настоящей части. Если положения настоящей части или указанного документа допускают вариативность оформления тех или иных конструкций, следует придерживаться одного варианта оформления в рамках всего программного проекта.

При работе над сторонними программными проектами следует придерживаться принятых в нём правил оформления исходного кода.

5.1.1. Комментирование

5.1.1.1. Комментирование объектов классов (переменных)

Назначение объектов классов (переменных) должно быть документировано. Если комментарий находится не на строке объявления объекта (переменной), он должен размещаться непосредственно над объявлением.

5.1.1.2. Комментарии Doxygen

Для построения технической документации следует использовать средство Doxygen.

Комментарии Doxygen используются для документирования модулей (пакетов), классов, методов и объектов (переменных).

5.2. Разработка с учётом особенностей инструментальных средств

5.2.1. Работоспособность на поддерживаемых платформах

Должна существовать возможность запустить любое фиксированное состояние исходного кода, извлечённое из репозитория системы управления исходным кодом, с использованием любого интерпретатора (версии интерпретатора), для которого в проекте оговорена поддержка.

Необходимо тестирование разрабатываемых программ на всех платформах (уникальных сочетаниях интерпретатора, его версии и ИПП вычислительной системы), для которых оговорена возможность их использования. На всех платформах разрабатываемое ПО должно быть полностью работоспособным.

5.2.2. Дополнительные проверки корректности исходного кода

При текущей работе над проектом необходимо применять средства статического анализа исходного кода.

Необходимо применять средства анализа корректности и аудита безопасности разрабатываемого ПО в целом, когда это целесообразно, например, при разработке веб-интерфейсов (веб-приложений).

6. Правила оформления исходного кода на языке Perl 5

Выработка исчерпывающих правил оформления исходного кода на языке Perl 5 представляется затруднительной. В связи с этим при разработке исходного кода на Perl 5 предлагается придерживаться общих рекомендаций, приведённых в документе `perlstyle` (Perl style guide) из поставки Perl 5, а также соображений подобия оформления исходного кода оформлению исходного кода программ на языке C или Python, в зависимости от области применения.

Требования к комментированию исходного кода и разработке с учётом особенностей инструментальных средств для исходного кода на языке Perl 5 те же, что для исходного кода на языке Python, за исключением требования использовать комментарии Doxygen.

Список использованных источников

- [axi-spec] AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/hc/.
- [axis-spec] AMBA AXI-Stream Protocol Specification. https://developer.arm.com/documentation/ihi0051/b/.
- [c11-std] ISO/IEC 9899:2011 - Programming languages - C. https://www.iso.org/standard/57853.html.
- [c18-std] ISO/IEC 9899:2018 - Programming languages - C. https://www.iso.org/standard/74528.html.
- [c99-std] ISO/IEC 9899:1999 - Programming languages - C. https://www.iso.org/standard/29237.html.
- [doxygen-offsite] Doxygen. https://doxygen.nl/.
- [pep8] PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/.
- [sv-std] "IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

Приложение A: Демонстрационная программа на языке C

Листинг 44. Содержимое файла `example.h`

```c
/**
 * @file example.h
 * @brief Публичный экспортируемый интерфейс
 */

#ifndef EXAMPLE_H
#define EXAMPLE_H

/**
 * Назначение структуры
 */
struct some_struct {
  int a; /**< описание поля a */
  unsigned int b; /**< описание поля b */
  double c; /**< описание поля c */
};

/**
 * @param [in] i Исходное число
 * @return Результат вычислений
 *
 * Функция принимает целочисленный аргумент и возвращает результат вычисления
 * по формуле @f$i = \(P \cdot j + Q_x\) \bmod N@f$
 */
int
libexample_public_interface(int i);

#endif
```

Листинг 45. Содержимое файла `example-priv.h`

```
/**
 * @file example-priv.h
 * @brief Файл предназначен для включения только в исходный код проекта и не
 * должен распространяться и устанавливаться на машины пользователей
 */

#ifndef EXAMPLE_PRIV_H
#define EXAMPLE_PRIV_H

/**
 * @brief Служебная функция, используемая в различных исходных файлах
 * проекта, но не экспортируемая для сторонних приложений
 * @param [in,out] d Входной параметр
 */
void
private_interface(double *d);

#endif
```

Листинг 46. Содержимое файла `example.c`

```
/**
 * @file example.c
 * @brief Исходный код программы
 */

#include "example.h"
#include "example-priv.h"

#include <assert.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include <stddef.h>

#define NELEMS(a) (sizeof(a) / sizeof(a[0]))

/**
 * Проверка допустимости выполнения арифметических операций
 * @result Результат проверки (true, если дальнейшее выполнение допустимо)
 */
static bool
```

```
25 is_everything_safe(void)
26 {
27   /* Выполнение арифметических операций допустимо только в случае,
28      когда... */
29   do {
30     // Проверка выполняется, пока...
31     int32_t p = 9;
32     uint16_t q = 3;
33     size_t r = 8;
34     if (p - q - r < 0)
35       return false;
36   } while (false);
37   return true;
38 }
39
40 int
41 main(void)
42 {
43   int a[] = { 26, 83, 59, 57 };
44   const size_t sizea = NELEMS(a);
45
46   assert(sizea > 1 && "Массив должен содержать больше одного элемента");
47   while (true) {
48     double *v = (double *)malloc(sizeof(*v));
49
50     if (NULL == v)
51       return EXIT_FAILURE;
52     for (;;) {
53       for (size_t i = 1; i < sizea; ++i)
54         if (is_everything_safe() // проверка необходима для дальнейшей работы
55             && a[i-1] < a[i])
56           goto out;
57     }
58   }
59   private_interface(v);
60   free(v);
61
62 out:
63   switch (a[1]) {
64     case 2:
65     case 3:
66       (void)libexample_public_interface(4);
67       break;
68
69     case 4:
70       {
71         int z = a[2] + a[3];
72         printf("z<%d>\n", z);
```

```
      }
      break;

    case 5:
      puts("5");
      // fallthrough

    default:
      break;
  }

  if (!is_everything_safe()
      || 0 == printf("sin(a[1]) = %f\n", sin((double)a[1])))
    return EXIT_FAILURE;

  return EXIT_SUCCESS;
}
```

Приложение B: Демонстрационная программа на языке SystemVerilog

Листинг 47. Содержимое файла `quarter_sin_cos_lut.sv`

```
//------------------------------------------------------------------------------
//
// Модуль:      quarter_sin_cos_lut
// Описание:    Модуль представляет собой таблицу поиска первой четверти синуса
//              или косинуса.
//
// Параметры:   WIDTH (<= 18) - разрядность значений синуса или косинуса
//              DEPTH (8 | 10 | 12) - разрядность шины адреса
//              SINCOS (sin | cos) - выбор генерации таблицы синуса или косинуса
//              ROM_STYLE (block| distributed) - указывает инструменту синтеза,
//                  как определить ROM
//
// Входы:       clk - тактирование
//              ival - валидность входных данных
//              iaddr (unsigned) - адрес значения синуса или косинуса в памяти
//
// Выходы:      oval - валидность выходных данных
//              odata (signed) - значение синуса или косинуса
//
//------------------------------------------------------------------------------
module quarter_sin_cos_lut #(
    parameter int WIDTH = 18,
    parameter int DEPTH = 10,
    parameter string SINCOS = "sin",
    parameter string ROM_STYLE = "block"
)(
    input logic clk,
    input logic ival,
    input logic [DEPTH-1:0] iaddr,

    output logic oval,
    output logic signed [WIDTH-1:0] odata
);

// Разрядность значений синуса (косинуса) в памяти
localparam int ROM_WIDTH = 18;

// Валидация параметров
if (WIDTH > ROM_WIDTH) begin
    $fatal(1, "Parameter WIDTH must be less or equal 18.");
end
```

```
42
43 if (~((DEPTH == 8) || (DEPTH == 10) || (DEPTH == 12))) begin
44     $fatal(1, "Parameter DEPTH must be {8, 10, 12}.");
45 end
46
47 if (~((SINCOS == "sin") || (SINCOS == "cos"))) begin
48     $fatal(1, "Parameter SINCOS must be \"sin\" or \"cos\"");
49 end
50
51 if (~((ROM_STYLE == "block") || (ROM_STYLE == "distributed"))) begin
52     $fatal(1, "Parameter ROM_STYLE must be \"block\" or \"distributed\"");
53 end
54
55 // Выбор файла инициализации памяти
56 localparam string FILE_NAME = (DEPTH == 8) ? "quarter_sin_lut_18_256.mem" :
57     ((DEPTH == 10) ? "quarter_sin_lut_18_1024.mem" :
58     "quarter_sin_lut_18_4096.mem");
59
60 // LUT синуса (косинуса)
61 (* rom_style = ROM_STYLE *)
62 logic signed [ROM_WIDTH-1:0] rom[(2**DEPTH)-1:0];
63
64 logic signed [ROM_WIDTH-1:0] data;
65
66 assign odata = data[ROM_WIDTH-1 -: WIDTH];
67
68 // Инициализация памяти
69 initial begin
70     if (SINCOS == "sin") begin
71         $readmemh(FILE_NAME, rom, 0, (2 ** DEPTH) - 1);
72     end
73     else begin
74         $readmemh(FILE_NAME, rom, (2 ** DEPTH) - 1, 0);
75     end
76 end
77
78 // Чтение данных из памяти
79 always_ff @(posedge clk) begin
80     if (ival) begin
81         oval <= 1'b1;
82         data <= rom[iaddr];
83     end
84     else begin
85         oval <= 1'b0;
86     end
87 end
88
89 endmodule
```

Листинг 48. Содержимое файла `sin_cos_lut.sv`

```
 1 //-----------------------------------------------------------------------------
 2 //
 3 // Модуль:      sin_cos_lut
 4 // Описание:    Модуль преобразует входную фазу в значения синуса и косинуса.
 5 //              Для генерации синуса и косинуса используются таблицы поиска
 6 //              размером от 0 до pi/2. Два старших бита фазы используются для
 7 //              для определения четверти функций, а остальные биты определяют
 8 //              адрес значения функций в памяти.
 9 //
10 // Параметры:   WIDTH (<= 18) - разрядность значений синуса или косинуса
11 //              DEPTH (10 | 12 | 14) - использование LUT
12 //              ROM_STYLE (block | distributed) - указывает инструменту синтеза,
13 //                  как определить ROM
14 //
15 // Входы:       clk - тактирование
16 //              rst - синхронный сброс
17 //              ival - валидность входных данных
18 //              iphase (unsigned) - фаза синуса или косинуса от 0 до 2pi
19 //
20 // Выходы:      oval - валидность выходных данных
21 //              osin (signed) - значение синуса
22 //              ocos (signed) - значение косинуса
23 //
24 //-----------------------------------------------------------------------------
25 module sin_cos_lut #(
26     parameter int WIDTH = 18,
27     parameter int DEPTH = 10,
28     parameter string ROM_STYLE = "block"
29 )(
30     input logic clk,
31     input logic rst,
32     input logic ival,
33     input logic [DEPTH-1:0] iphase,
34
35     output logic oval,
36     output logic signed [WIDTH-1:0] osin,
37     output logic signed [WIDTH-1:0] ocos
38 );
39
40 // Разрядность значений синуса (косинуса) в памяти
41 localparam int ROM_WIDTH = 18;
42 // Разрядность адреса памяти меньше на 2, поскольку требуется хранить  только
43 // четверть синуса (косинуса)
44 localparam int ROM_DEPTH = DEPTH - 2;
45
46 // Валидация параметров
```

```
47 if (WIDTH > ROM_WIDTH) begin
48     $fatal(1, "Parameter WIDTH must be less or equal 18.");
49 end
50
51 if (~((DEPTH == 10) || (DEPTH == 12) || (DEPTH == 14))) begin
52     $fatal(1, "Parameter DEPTH must be {10, 12, 14}.");
53 end
54
55 if (~((ROM_STYLE == "block") || (ROM_STYLE == "distributed"))) begin
56     $fatal(1, "Parameter ROM_STYLE must be \"block\" or \"distributed\"");
57 end
58
59 logic [1:0] quarter_period; // Биты фазы, используемые для определения четверти
60 logic [ROM_DEPTH-1:0] addr; // Биты фазы, формирующие адрес памяти
61
62 // Выходные данные таблицы поиска синуса
63 logic quarter_sin_lut_oval;
64 logic signed [WIDTH-1:0] quarter_sin_lut_odata;
65 // Выходные данные таблицы поиска косинуса
66 logic quarter_cos_lut_oval;
67 logic signed [WIDTH-1:0] quarter_cos_lut_odata;
68
69 // В зависимости от четверти читаем из памяти либо в обратном порядке,
70 // либо прямом
71 assign addr = (iphase[DEPTH-2]) ? ~iphase[DEPTH-3:0] : iphase[DEPTH-3:0];
72
73 // Таблица поиска синуса
74 quarter_sin_cos_lut #(
75     .WIDTH(WIDTH),
76     .DEPTH(ROM_DEPTH),
77     .SINCOS("sin"),
78     .ROM_STYLE(ROM_STYLE)
79 )
80 quarter_sin_lut (
81     .clk(clk),
82     .ival(ival),
83     .iaddr(addr),
84
85     .oval(quarter_sin_lut_oval),
86     .odata(quarter_sin_lut_odata)
87 );
88
89 // Таблица поиска косинуса
90 quarter_sin_cos_lut #(
91     .WIDTH(WIDTH),
92     .DEPTH(ROM_DEPTH),
93     .SINCOS("cos"),
94     .ROM_STYLE(ROM_STYLE)
```

```
 95 )
 96 quarter_cos_lut (
 97     .clk(clk),
 98     .ival(ival),
 99     .iaddr(addr),
100 
101     .oval(quarter_cos_lut_oval),
102     .odata(quarter_cos_lut_odata)
103 );
104 
105 always_ff @(posedge clk) begin
106     if (rst) begin
107         quarter_period <= 2'b00;
108         oval <= 1'b0;
109         osin <= {WIDTH{1'b0}};
110         ocos <= {WIDTH{1'b0}};
111     end
112     else begin
113         oval <= 1'b0;
114 
115         if (ival) begin
116             quarter_period <= iphase[DEPTH-1 -: 2];
117         end
118 
119         // Определение четверти
120         if (quarter_sin_lut_oval | quarter_cos_lut_oval) begin
121             oval <= 1'b1;
122             case (quarter_period)
123                 2'b00: begin
124                     osin <= quarter_sin_lut_odata;
125                     ocos <= quarter_cos_lut_odata;
126                 end
127                 2'b01: begin
128                     osin <= quarter_sin_lut_odata;
129                     ocos <= -quarter_cos_lut_odata;
130                 end
131                 2'b10: begin
132                     osin <= -quarter_sin_lut_odata;
133                     ocos <= -quarter_cos_lut_odata;
134                 end
135                 2'b11: begin
136                     osin <= -quarter_sin_lut_odata;
137                     ocos <= quarter_cos_lut_odata;
138                 end
139             endcase
140         end
141     end
142 end
```

```
143
144 endmodule
```

Листинг 49. Содержимое файла `nco.sv`

```
  1 //-----------------------------------------------------------------------------
  2 //
  3 // Модуль:      nco
  4 // Описание:    Генератор с числовым управлением. Генерирует на выходе функции
  5 //              синуса и косинуса.
  6 //
  7 // Задержка:    4 такта
  8 //
  9 // Параметры:   PHASE_WIDTH (<= 48) - разрядность входной фазы
 10 //              OUTPUT_WIDTH (<= 18) - разрядность значений синуса и косинуса
 11 //              DEPTH (10 | 12 | 14) - использование LUT
 12 //              ACC_USE_DSP (yes | no) - указывает инструменту синтеза, как
 13 //                  определить операции в аккумуляторе фазы
 14 //              ROM_STYLE (block | distributed) - указывает инструменту синтеза,
 15 //                  как определить ROM
 16 //
 17 // Входы:       clk - тактирование
 18 //              rst - синхронный сброс
 19 //              ival - валидность входных данных
 20 //              iphase_inc (signed) - инкремент фазы
 21 //              iphase_offset (signed) - фазовое смещение
 22 //
 23 // Выходы:      oval - валидность выходных данных
 24 //              osin (signed) - значение синуса
 25 //              ocos (signed) - значение косинуса
 26 //
 27 //-----------------------------------------------------------------------------
 28 module nco #(
 29     parameter int PHASE_WIDTH = 14,
 30     parameter int OUTPUT_WIDTH = 18,
 31     parameter int DEPTH = 14,
 32     parameter string ACC_USE_DSP = "yes",
 33     parameter string ROM_STYLE = "block"
 34 )(
 35     input logic clk,
 36     input logic rst,
 37     input logic ival,
 38     input logic signed [PHASE_WIDTH-1:0] iphase_inc,
 39     input logic signed [PHASE_WIDTH-1:0] iphase_offset,
 40
 41     output logic oval,
 42     output logic signed [OUTPUT_WIDTH-1:0] osin,
 43     output logic signed [OUTPUT_WIDTH-1:0] ocos
```

```
44 );
45
46 // Максимальная разрядность фазы, определяется максимальной разрядностью для
47 // сложения в DSP
48 localparam int MAX_PHASE_WIDTH = 48;
49 // Максимальная разрядность значений синуса и косинуса
50 localparam int MAX_OUTPUT_WIDTH = 18;
51
52 // Валидация параметров
53 if (PHASE_WIDTH > MAX_PHASE_WIDTH) begin
54     $fatal(1, "Parameter WIDTH must be less or equal 48.");
55 end
56
57 if (OUTPUT_WIDTH > MAX_OUTPUT_WIDTH) begin
58     $fatal(1, "Parameter WIDTH must be less or equal 18.");
59 end
60
61 if (~((DEPTH == 10) || (DEPTH == 12) || (DEPTH == 14))) begin
62     $fatal(1, "Parameter DEPTH must be {10, 12, 14}.");
63 end
64
65 if (~((ROM_STYLE == "block") || (ROM_STYLE == "distributed"))) begin
66     $fatal(1, "Parameter ROM_STYLE must be \"block\" or \"distributed\"");
67 end
68
69 if (~((ACC_USE_DSP == "yes") || (ACC_USE_DSP == "no"))) begin
70     $fatal(1, "Parameter ACC_USE_DSP must be \"yes\" or \"no\"");
71 end
72
73 logic [2:0] ival_d; // Линия задержки сигнала ival
74 // Аккумулятор фазы
75 (* use_dsp = ACC_USE_DSP *) logic [PHASE_WIDTH-1:0] phase_acc;
76 logic [PHASE_WIDTH-1:0] phase_offset; // Фазовое смещение
77 // Аккумулятор + фазовое смещение
78 (* use_dsp = ACC_USE_DSP *) logic [PHASE_WIDTH-1:0] phase;
79 logic [DEPTH-1:0] quantized_phase; // Усеченный сигнал phase
80
81 assign quantized_phase = phase[PHASE_WIDTH-1 -: DEPTH];
82
83 // Таблица поиска синуса косинуса
84 sin_cos_lut #(
85     .WIDTH(OUTPUT_WIDTH),
86     .DEPTH(DEPTH),
87     .ROM_STYLE(ROM_STYLE)
88 )
89 sin_cos_lut_inst_0 (
90     .clk(clk),
91     .rst(rst),
```

```
92      .ival(ival_d[1]),
93      .iphase(quantized_phase),
94
95      .oval(oval),
96      .osin(osin),
97      .ocos(ocos)
98  );
99
100 always_ff @(posedge clk) begin
101     if (rst) begin
102         ival_d <= 3'b000;
103         phase_acc <= {PHASE_WIDTH{1'b0}};
104         phase_offset <= {PHASE_WIDTH{1'b0}};
105         phase <= {PHASE_WIDTH{1'b0}};
106     end
107     else begin
108         ival_d <= {ival_d[1:0], ival};
109
110         if (ival) begin
111             phase_acc <= phase_acc + iphase_inc;
112             phase_offset <= iphase_offset;
113         end
114
115         if (ival_d[0]) begin
116             phase <= phase_acc + phase_offset;
117         end
118     end
119 end
120
121 endmodule
```