Глава 2.

Пространства объявлений SystemVerilog.

Verlog имеет ограниченные пространства, в которых разработчики могут объявлять переменные и другую проектную информацию. SystemVerilog расширяет пространства объявлений в нескольких направлениях. Эти расширения облегчают моделирование сложных конструкций данных в проекте, и сокращает трудно-выявляемые ошибки при кодировании. System Verilog улучшает определения временных единиц симуляции.

Темы обсуждаемые в этой главе:

- Пакетные определения и импортирование определений из пакетов
- $unit компиляция пространства объявлений
- Объявления в неименованных блоках
- Улучшенные определения временных единиц

Перед тем как вдаваться в детали новых типов данных, которые предлагает SystemVerilog, важно знать, где разработчики могут определять информацию, которая используется в проекте. Чтобы проиллюстрировать новые пространства объявлений, в этой главе будут использоваться некоторые типы данных SystemVerilog, которые будут обсуждаться в следующих главах. Вкратце, некоторые типы данных, используемые в этой главе:

logic - 1-битовая переменная с 4-мя состояниями, такая как тип reg в Verilog; может объявляться как вектор (обсуждается в главе 3).

enum – перечислитель net или переменная с установленным значением, подобная перечислительному типу enum в языке C.

typedef - тип данных определённый пользователем, созданный из встроенных типов или других типов определённых пользователем, подобный typedef в языке C (обсуждается в главе 4).

struct - совокупность переменных, к которым можно обращаться индивидуально или как к единому целому, подобна типу struct в языке C.

2.1 Пакеты

Языку Verilog необходимы локальные объявления.

В Verilog переменные, сети (nets), задачи и функции, должны быть объявлены внутри модуля, между ключевыми словами **module ... endmodule**. Объекты, объявленные внутри модуля, являются локальными по отношению к модулю. Обращаться к этим объектам можно внутри модуля. Verilog также позволяет иерархические ссылки к этим объектам из других модулей в целях верификации, но эти кроссмодульные ссылки не предоставляют поведение системы на аппаратном уровне, и на уровне синтеза. Verilog также позволяет определять переменные в именованных блоках (**begin ... end** или **fork ... join**), задачах и функциях. Эти объявления доступны только внутри модуля.

Verilog не имеет пространства для глобальных объявлений, таких как глобальные функции. Объявление, которое используется во многих блоках проекта должно быть объявлено в каждом блоке. Это не только требует избыточности объявлений, но и может приводить к ошибкам, если объявление функции изменено в одном блоке, но не изменено в другом. Многие разработчики используют включаемые (include) файлы и другие трюки кодирования, чтобы обойти эти проблемы, но это тоже может привести к ошибкам в кодировании и проблемам поддержки проекта.

System Verilog добавляет в Verilog типы определяемые пользователем.

System Verilog добавляет в Verilog типы определяемые пользователем, используя **typedef**. Часто, типы определённые пользователем, используются в нескольких модулях. Используя правила Verilog, где объявления всегда локальны по отношению к модулю, необходимо продублировать определение в каждом модуле, где оно используется. Избыточность локальных определений нежелательна для типов определённых пользователем.

2.1.1 Пакетные определения.

SystemVerilog добавляет пакеты в Verilog.

Чтобы использовать типы определённые пользователем в нескольких модулях, System Verilog добавляет пакеты в язык Verilog. Концепция пакетов позаимствована из языка VHDL. Пакеты определяются между ключевыми словами **package** и **endpackage**.

Пакеты могут содержать синтезируемые конструкции :

- Определения констант **parameter** и **localparam**.
- Определения переменных **const**.
- Типы определённые пользователем **typedef**.
- Определения automatic **task** и **function**.
- **import** имортирование из других пакетов.
- Определения оператора **overload**.

Пакеты также могут содержать объявления глобальных переменных, статических задач и функций. Эти объявления не являются синтезируемыми и не обсуждаются в этой книге.

Пакетные определения независимы от модулей.

Пакет это отдельное пространство. Оно не встроено в Verilog модуль.
Простой пример пакетного определения:

Example 2-1: A package definition

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

Параметры в пакетах не могут быть переопределены.

Пакеты могут содержать **parameter, localparam** и **const** объявления, **parameter** и **localparam** это костантные конструкции Verilog. Const это константа System Verilog, обсуждается в секции 3.10 на странице 71. В Verilog, **parameter** может быть переопределён для каждого объекта модуля, в то время как **localparam** напрямую переопределён быть не может. Однако, в пакете **parameter** не может быть переопределён, так как не является частью объекта модуля. В пакете **parameter и localparam** - синонимы.

2.1.2 Обращение к содержимому пакета.

Модули и интерфейсы могут обращаться к объявлениям и определениям в пакете четырьмя способами :

- Напрямую, используя оператор области видимости.
- Импортирование специфичных элементов пакета в модуль или интерфейс.
- Импортирование всех элементов пакета в модуль или интерфейс.
- Импортирование элементов пакета в пространство объявлений $unit.

Первые три метода обсуждаются в этой секции. Импортирование в $unit обсуждается позже в этой главе в секции 2.2 на странице 14.

Обращение к пакету, используя оператор области видимости.

:: используется для обращения к элементам в пакете.

System Verilog добавляет :: *"scope resolution operator"* в Verilog. Этот оператор позволяет обращаться к пакету через имя пакета, и затем выбирать необходимые определения или объявления внутри пакета. Имя пакета и имя элемента пакета разделяются двойным двоеточием (::) . На пример, порт модуля может быть определён как тип instruction_t, где instruction_t определён в пакете definitions, пример 2-1 на странице 9.

Example 2-2: Explicit package references using the : : scope resolution operator

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  always_ff @(posedge clock) begin
    case (IW.opcode)
      definitions::ADD : result = IW.a + IW.b;
      definitions::SUB : result = IW.a - IW.b;
      definitions::MUL : result = definitions::
                                  multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Явное обращение к пакету помогает документировать исходный код.

Явное обращение к содержимому пакета помогает документировать исходный код проекта. В примере 2-2, выше, использование имени пакета делает код очевидным благодаря определениям instruction_t, ADD, SUB, MUL и multiplier. Однако, когда необходимо обратиться к элементам пакета в модуле много раз, то явное обращение через имя пакета может оказаться избыточным. В этом случае более приемлемым будет импорт элементов пакета в блок.

Импортирование элементов пакета

Импортирование делает элементы пакета видимыми локально.

System Verilog позволяет импортировать определённые элементы пакета в модуль, используя оператор **import**. Когда пакетное определение или объявление импортировано в модуль или интерфейс, то оно становится видимым внутри модуля или интерфейса, как будто бы оно было определено локально внутри модуля или интерфейса. Больше не необходимости каждый раз явно обращаться к элементу пакета.

Импортирование пакетного определения или объявления может упростить код внутри модуля. Пример 2-2 изменён и показан ниже как пример 2-3, используя оператор import чтобы сделать имена меток перечислительного типа локальными внутри модуля. Оператор case может обращаться к этим именам без явного имени пакета.

Example 2-3: Importing specific package items into a module

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);

  import definitions::ADD;
  import definitions::SUB;
  import definitions::MUL;
  import definitions::multiplier;

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Импортирование определения перечислительного типа не импортирует метки, используемые внутри этого определения.

В примере 2-3 , выше, следующий оператор import работать не будет

import definitions :: opcode_t;

перечислительные метки должны быть импортированы, чтобы иметь к ним локальный доступ.

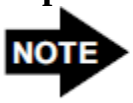
Этот оператор import сделает тип определяемый пользователем opcode_t видимым в модуле. Однако, перечислительные метки нельзя будет использовать. Каждая перечислительная метка должна быть явно импортирована, для того чтобы метки стали видимыми как локальные имена внутри модуля. Когда необходимо импортировать много элементов из пакета, наиболее удобно использовать wildcard import.

Импортирование всех(wildcard) элементов пакета

*Все элементы в пакете можно сделать видимыми, используя (*).*

System Verilog позволяет импортировать элементы пакета, используя wildcard, вместо именованных элементов пакета. Wildcard обозначается звёздочкой. Например:

import definitions ::*; // wildcard import



wildcard не импортирует автоматически всё содержимое пакета.

wildcard не импортирует автоматически целый пакет.

Когда элементы пакета импортируются, используя wildcard, импортируются только те элементы, которые действительно используются в модуле. Определения и объявления в пакете, к которым нет обращений, не импортируются. Локальные определения и объявления внутри модуля или интерфейса первыми импортируются через wildcard. Имена элементов пакета также первыми импортируются через wildcard. С точки зрения разработчика, wildcard импорт просто добавляет пакет в правила поиска по

идентификатору. Локальные объявления будут искаться первыми (следуя правилам поиска Verilog внутри модуля), затем происходит поиск в пакетах, которые были импортированы, используя wildcard. И в завершении, будет происходить поиск в пространстве объявлений $unit SystemVerilog. Пространство $unit обсуждается в секции 2.2 в этой главе.

Пример 2-4, ниже, использует оператор импорта wildcard. Добавляется пакет к пути поиска идентификатора. Когда оператор case обращается к меткам ADD, SUB и MUL, а также функции multiplier, он будет искать определения этих имён в пакете definitions.

Example 2-4: Using a package wildcard import

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  import definitions::*;  // wildcard import

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

В примерах 2-3, 2-4, порт IW обращается к имени пакета явно. Невозможно добавить оператор **import** между ключевым словом module и определением порта. Существует способ избежать явного обращения в список порта через имя пакета, используя пространство объявлений $unit. Пространство $unit обсуждается в 2.2.

2.1.3 Рекомендации по синтезу

Для синтеза, пакетные задачи и функции должны быть automatic.

Когда происходит обращение к задаче или функции, которая определена в пакете, синтез будет дублировать функциональность задачи или функции

и рассматривать их, как будто бы они определены внутри модуля. Чтобы быть синтезирумыми, задачи и функции, определённые в пакете, должны быть объявлены как automatic, и не могут содержать статических переменных. В случае automatic, для задачи или функции, память будет выделяться при каждом вызове задачи или функции. Таким образом, каждый модуль, который обращается к automatic задаче или функции в пакете видит уникальную копию выделенной памяти задачи или функции, которая не разделяется с другим модулем. Это гарантирует, что поведение пред-синтезного обращения к пакетной задаче или функции будет таким же как и пост-синтезное, где функциональность задачи или функции реализована в одном или нескольких модулях.

По тем же причинам, синтез не поддерживает объявления переменных в пакетах. При симуляции, пакетная переменная будет разделяться другими модулями, которые импортировали эту переменную. Один модуль может записать в переменную, другой модуль будет видеть новое значение. Этот тип межмодульного взаимодействия без (passing values) прохождения значений через порты модуля не является синтезируемым.

2.2 Объявления еденицы компиляции $unit

System Veriolg добавляет в Verilog концепцию единицы компиляции. Единица компиляции это все файлы исходного кода скомпилированные одновременно. Единицы компиляции предоставляют данные, чтобы отдельно компилировать суб-блоки всего проекта. Суб-блок может содержать один или несколько модулей. Модули могут находиться в одном или нескольких файлах. Суб-блок проекта может также содержать интерфейсные блоки (представлены в главе 10) и блоки тестов (SystemVerilog for Verification).

Области видимости единицы компиляции содержат внешние объявления.

System Verilog расширяет пространство объявлений Verilog, разрешая быть объявлениям вне пакета, модуля, интерфейса и границ программного блока. Эти внешние объявления находятся в области видимости единицы компиляции и видимы для всех модулей которые компилируются одновременно.

Области видимости единицы компиляции может одержать:

- Объявления единицы времени и точности (см. 2.4)
- Объявления переменных
- Объявления сетей (net)
- Объявления констант
- Типы данных определённые пользователем, с использованием typedef, enum или class
- Определения задачи и функции.

Следующий пример показывает внешние объявления константы, переменной, типа определённого пользователем, и функции.

Example 2-5: External declarations in the compilation-unit scope (not synthesizable)

```
/******************* External declarations ******************/
parameter VERSION = "1.2a";    // external constant

reg resetN = 1;                // external variable (active low)

typedef struct packed {        // external user-defined type
  reg [31:0] address;
  reg [31:0] data;
  reg [ 7:0] opcode;
} instruction_word_t;

function automatic int log2 (input int n);  // external function
  if (n <=1) return(1);
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return(log2);
endfunction

/********************* module definition *********************/
// external declaration is used to define port types
module register (output instruction_word_t  q,
                 input  instruction_word_t  d,

                 input  wire                clock );

  always @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;  // use external reset
    else q <= d;
endmodule
```

Внешние объявления области видимости единицы компиляции не являются глобальными.

Объявление в области видимости единицы компиляции это не тоже самое, что глобальное объявление. Глобальное объявление, такое как глобальная переменная или функция разделяется всеми модулями проекта, независимо от того, компилируются исходные файлы отдельно или одновременно.

Область видимости компиляции существует только для исходных файлов, которые компилируются одновременно. Каждый раз, когда компилируются исходные файлы, создаётся уникальная область видимости единицы компиляции. На пример, если модуль CPU и модуль controller обращаются к внешней переменной reset, то возможны два сценария:

- Если два модуля скомпилированы одновременно, то будет создана единственная область видимости единицы компиляции. Внешняя переменная reset будет общей для обоих модулей.

- Если каждый модуль скомпилирован отдельно, то будет две области видимости единицы компиляции, возможно с двумя переменными reset.

Компиляция, с объявлением внешней переменной reset, пройдёт успешно. Другой файл, при раздельной компиляции, не увидит объявления reset от предыдущей компиляции. В зависимости от того как используется reset, вторая компиляция может потерпеть неудачу, благодаря необъявленной переменной, или быть успешной, делая reset неявным соединением (net). Это является опасным. Если вторая компиляция успешна, делая reset неявным соединением (net), то будет два сигнала reset, один в каждой компиляции. Два разных сигнала reset не будут соединены.

2.2.1 Рекомендации по кодированию

$unit следует использовать только для импортирования пакетов

1. Не делайте объявлений в пространстве $unit! Все объявления следует делать в именованных пакетах.

2. Когда необходимо, пакеты могут быть импортированы в $unit. Это полезно когда модуль или интерфейс содержат много портов, которые имеют пользовательский тип, и определения типа в пакете.

Прямое объявление объектов в пространстве единицы компиляции может привести к ошибкам, когда файлы компилируются отдельно. Также это может привести к спагетти-коду, если объявления разбросаны по многим файлам, что затрудняет поддержку, повторное использование и отладку.

2.2.2 Правила поиска идентификатора SystemVerilog

К объявлениям в области видимости единицы компиляции можно обращаться везде в иерархии модулей, которые являются частью единицы компиляции.

Область видимости единицы компиляции стоит третьей в очереди на поиск

1. Первыми ищутся локальные объявления, как определено в стандарте IEEE 1364 Verilog standart.

2. Вторыми объявления в пакете, которые имортированы в текущую область видимости через wildcard (:: *).

3. Третьими ищутся объявления в области видимости единицы компиляции.

4. Четвёртыми ищутся объявления внутри иерархии проекта, следуя правилам поиска IEEE 1364 Verilog.

Правила поиска SystemVerilog гарантируют обратную совместимость с Verilog.

2.2.3 Порядок исходного кода

Идентификаторы данных, определения типов должны быть объявлены до их использования.

Переменные и сети в области видимости единицы компиляции

Необъявленные идентификаторы имеют неявный тип net

Есть важное замечание по использованию внешних объявлений. Verilog поддерживает неявное объявление типов, в определённом контексте где необъявленный идентификатор рассматривается как тип net (wire). В Verilog необходимо объявлять тип идентификаторов до их использования,
когда контекст не подразумевает использование неявного типа, или когда тип отличается от типа по умолчанию net.

Внешние объявления должны быть определены перед использованием

Это правило, неявное объявление типа, влияет на объявление переменных и сетей в области видимости единицы компиляции. Программный инструментарий должен получать внешнее объявление до обращения к идентификатору. Если нет, то имя будет рассматриваться как необъявленный идентификатор, и применяться правила для неявных типов.

Следующий пример показывает, как порядок исходного кода может влиять на использование внешнего объявления отношению к модулю. Этот пример не будет генерировать какой-либо тип компиляции или выдавать ошибку. Для модуля *parity_gen,* программный инструментарий будет автоматически рассматривать *parity*, как неявный локальный тип net, пока обращение к *parity* происходит до внешнего объявления. С другой стороны, модуль *parity_check* стоит после внешнего объявления *parity* в исходном коде. Поэтому *parity_check* модуль будет использовать внешнее объявление переменной.

```
module parity_gen (input wire [63:0] data );
  assign parity = ^data; // parity is an
endmodule                // implicit local net


reg parity;  // external declaration is not
             // used by module parity_gen
             // because the declaration comes
             // after it has been referenced


module parity_check (input  wire [63:0] data,
                     output logic       err);
  assign err = (^data != parity); // parity is
                                  // the $unit
endmodule                         // variable
```

2.2.3 Рекомендации по кодированию для импортирования пакетов в $unit

В SystemVerilog можно объявлять порты модуля как типы, определяемые пользователем. Стиль кодирования, в этой книге**,** рекомендует размещать определения в одном или нескольких пакетах. Пример 2-2, ранее показанный в этой главе, иллюстрирует использование пакетов. Часть этого примера:

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
```

Явное обращение к пакету, как показано выше, избыточно, когда много портов модуля являются типами определяемыми пользователем. Альтернативный стиль это импортирование пакета в область видимости единицы компиляции $unit, предшествующей объявлению модуля. Это делает определения пользовательского типа видимыми для поиска. На пример:

```
// import specific package items into $unit
import definitions::instruction_t;

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
```

Также пакет может быть импортирован в пространство $unit используя wildcard (: : *). Помните, что wildcard импортирует не все элементы пакета. Он просто добавляет пакет в путь к исходнику (source path). Следующий фрагмент кода иллюстрирует этот стиль.

```
// wildcard import package items into $unit
import definitions::*;

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
```

Импортирование пакетов в $unit с раздельной компиляцией

Импортирование пакетов в пространство $unit следует рассмотреть также внимательно, как и создание объявлений и определений в $unit. Порядок зависимостей и многочисленные $unit могут быть проблемой.

Файл очерёдности компиляции зависимостей

При импортировании компонент из пакета (отдельных компонент или всех через wildcard), процесс импортирования должен происходить до обращения к компонентам пакета. Если выражения импорта пакета находятся не в файлах модуля или интерфейса, которые обращаются к компонентам пакета, то файл с выражениями импорта пакета должен быть первым в файле очерёдности компиляции. Если файл очерёдности компиляции некорректен, тогда компиляция модуля или интерфейса потерпит неудачу или приведёт к неявным соединениям (nets), вместо ожидаемых компонентов пакета.

Компиляции нескольких файлов и компиляция одного файла

Компиляторы, симуляторы и другой инструментарий, который может читать исходный код Verilog и SystemVerilog, компилируют один или несколько файлов одновременно. Когда несколько файлов компилируется как единая компиляционная единица, то и пространство $unit будет одно. Импорт пакета в пространство $unit делает компоненты пакета видимыми для всех модулей и интерфейсов. Однако, если файлы скомпилированы раздельно, тогда будет несколько отдельных единиц компиляции $unit. Пакет, импортированный в одну $unit, не видим в другой $unit.

Использование выражений импорта в каждом файле

Решением этих проблем, с импортированием компонентов пакета в пространство единицы компиляции $unit, является размещение выражения импорта в каждом файле, перед определением модуля или интерфейса. Это решение хорошо работает, когда каждый файл компилируется отдельно. Однако, нужно быть внимательным когда несколько файлов компилируется как одна единица компиляции. Будет неправильным импортировать компоненты одного пакета более одного раза в одно и то же пространство $unit (это также как объявлять переменную дважды в одном пространстве имён).

Условная компиляция с импортированием пакета в $unit

Для импортирования компонентов пакета в пространство $unit, как с компилированием одного файла, так и нескольких файлов, можно использовать условную компиляцию для включения выражений импорта. Для того чтобы показать, что выражения импорта скомпилированы в текущее пространство $unit, устанавливается флаг в **`define** .

В следующем примере, пакетные определения находятся в отдельном файле с именем definitions.pkg (файл может иметь любое имя и расширение). После ключевого слова endpackage, весь пакет импортируется в пространство $unit. Таким образом, когда пакет скомпилирован, определения внутри пакета автоматически становятся видимыми в текущем пространстве $unit.

Внутри пакета definitions.pkg установлен флаг, показывающий, когда файл скомпилирован. Условная компиляция охватывает содержимое всего файла.

Если флаг не установлен, то пакет будет скомпилирован и импортирован в $unit. Если флаг установлен (показывает, что пакет уже скомпилирован и импортирован в текущее пространство $unit), то содержимое файла игнорируется.

Example 2-6: Package with conditional compilation (file name: definitions.pkg)

```
`ifndef DEFS_DONE  // if the already-compiled flag is not set...
  `define DEFS_DONE  // set the flag
  package definitions;

    parameter VERSION = "1.1";

    typedef enum {ADD, SUB, MUL} opcodes_t;

    typedef struct {
      logic [31:0] a, b;
      opcodes_t    opcode;
    } instruction_t;

    function automatic [31:0] multiplier (input [31:0] a, b);
      // code for a custom 32-bit multiplier goes here
      return a * b;  // abstract multiplier (no error detection)
    endfunction
  endpackage

  import definitions::*;  // import package into $unit

`endif
```

Строку

```
`include "definitions.pkg"
```

следует ставить в начало каждого файла проекта или тестбенч файла, которым необходимы определения пакета. Компиляция файла проекта или тестбенч файла включает в себя пакет и выражение импорта. Условная компиляция обеспечивает компиляцию и импорт пакета. Если пакет был уже скомпилирован и импортирован с текущее пространство $unit, то компиляция этого файла будет пропущена.

Для этого стиля кодирования, пакетный файл следует предоставлять компилятору, используя директиву **`include.**

Пакет следует компилировать, используя директиву `include**.**

Условная компиляция использует директиву **`include** чтобы скомпилировать файл definitions.pkg как часть компиляции другого файла. Это сделано для того чтобы гарантировать, что импорт – выражение будет импортировать пакет в тоже пространство $unit, которое используется при компиляции проектного файла или тестбенч файла. Если бы файл definitions.pkg был предоставлен компилятору напрямую – в командной строке, то пакет и выражение импорта могли быть скомпилированы не в то пространство $unit, которое используется проектным и тестбенч блоками.

Имя файла, в предыдущем примере, не имеет расширения .v или .sv. Файл имеет расширение .pkg, чтобы показать, что он не является ни проектным файлом, ни тестбенч файлом, и его не следует использовать в командной строке симулятора, компилятора или другого программного инструментария. Расширение .pkg выбрано произвольно и может быть каким-либо другим.

Примеры 2-7 и 2-8 иллюстрируют проектный файл и тестбенч файл, которые включают пакет в текущую компиляцию. Компоненты пакета условно включены в текущее пространство $unit, используя импорт wildcard (::*). Это делает компоненты пакета видимыми для модуля.

Example 2-7: A design file that includes the conditionally-compiled package file

```
`include "definitions.pkg" // compile the package file

module ALU
(input  instruction_t  IW,
 input  logic          clock,
 output logic [31:0]   result
);
  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

Example 2-8: A testbench file that includes the conditionally-compiled package file

```
`include "definitions.pkg" // compile the package file

module test;
  instruction_t test_word;
  logic [31:0]  alu_out;
  logic         clock = 0;

  ALU dut (.IW(test_word), .result(alu_out), .clock(clock));

  always #10 clock = ~clock;

  initial begin
    @(negedge clock)
    test_word.a = 5;
    test_word.b = 7;
    test_word.opcode = ADD;
    @(negedge clock)
    $display("alu_out = %d (expected 12)", alu_out);
    $finish;
  end
endmodule
```

`include *работает при компиляции одного и нескольких файлов*

При компиляции одного файла, пакет будет скомпилирован и импортирован в каждую единицу компиляции $unit. Это гарантирует, что каждая единица $unit видит те же компоненты пакета. Когда каждая $unit уникальна, то не будет конфликта имён при компиляции пакета.

При компиляции нескольких файлов, условная компиляция гарантирует, что пакет скомпилирован и импортирован в общее пространство $unit, которое разделяется всеми модулями. Скомпилированный файл проекта или тестбенч файл импортирует пакет, гарантируя, что компоненты пакета видимы для всех последующих файлов.

Условная компиляция, показанная в этой секции, не работает с глобальными переменными, статическими задачами и статическими функциями.

Переменные пакета являются разделяемыми переменными (не синтезируемыми)

Пакеты могут содержать объявления переменных. Переменная пакета является разделяемой всеми блоками проекта (и тест блоками) которые импортируют переменную. Поведение переменных пакета будет радикально отличаться при компиляции одного и нескольких файлов. При компиляции нескольких файлов, пакет импортируется в единое пространство компиляции $unit. Каждый проектный блок или тест блок будет видеть одни и те же переменные пакета. Значение, записанное в переменную пакета одним блоком, будет видимым для всех остальных блоков. При компиляции одного файла, каждое пространство $unit будет иметь уникальную переменную, которая имеет такое же имя как переменная в другом пространстве $unit. Значения, записанные в переменную пакета одним проектным или тестовым блоком, не будет видимым для другого проектного или тестового блоков.

Статические задачи и функции в пакетах не синтезируемы

Статические задачи и функции, или автоматические задачи и функции со статическим хранением, имеют потенциальную проблему. При компиляции нескольких файлов, единое пространство $unit будет импортировать один экземпляр задачи или функции. Статическая область хранения, внутри задачи или функции, является видимой для всех проектных и верификационных блоков. При компиляции одного файла, каждая отдельная $unit будет импортировать уникальный экземпляр задачи или функции. Статическая область хранения задачи или функции не является разделяемой между проектными и тест блоками.

Это ограничение на условно компилируемый импорт в $unit, не должно быть проблемой в моделях, которые написаны для синтеза, потому что синтез не поддерживает объявления переменных, статических задач и функций в пакетах (см секцию 2.1.3)

2.2.4 Рекомендации по синтезу

Синтезируемые конструкции, которые могут быть объявлены в области видимости единицы компиляции (внешние к определениям модуля и интерфейса) :

- **typedef** определения пользовательского типа
- автоматические функции
- автоматические задачи
- константы **parameter** и **localparam**
- импорт пакета

Использование пакетов вместо $unit является лучшим вариантом кодирования

Пользовательские типы, определённые в области видимости единицы компиляции синтезируемы. Но такой стиль не рекомендуется.
Размещение определений пользовательских типов в именованных пакетах является лучшим стилем. Использование пакетов сокращает риски получить спагетти-код и проблемы с зависимостями.

Внешние задачи и функции должны быть автоматическими

Объявление задач и функций в пространстве единицы компиляции $unit так же не рекомендуется. Однако, задачи и функции определённые в $unit синтезируемы. Когда модуль обращается к задаче или функции, которые определены в области видимости единицы компиляции, синтез будет дублировать код задачи или функции и рассматривать их, как будто они определены внутри модуля. Чтобы быть синтезируемыми, задачи и функции, определённые в области видимости единицы компиляции, должны быть объявлены как **automatic**, и не могут содержать статических переменных. Потому что область хранения задачи или функции выделяется каждый раз при вызове. Таким образом, каждый модуль который обращается к автоматической задаче или функции, в области видимости единицы компиляции, видит уникальную копию области хранения задачи или функции, которая не является разделяемой другими модулями. Это гарантирует одинаковое поведение задачи или функции до и после синтеза.

Константа **parameter**, определённая в области видимости единицы компиляции, не может быть переопределена, поскольку не является частью экземпляра модуля. Синтез рассматривает константы, объявленные в области видимости единицы компиляции, как литеральные значения. Объявление параметров в пространстве $unit не является хорошим стилем, поскольку константы не будут видны модулям, которые скомпилированы отдельно от файла, который содержит объявления констант.

2.3 Объявления в неименованных блочных выражениях

Локальные переменные в именованных блоках

Verilog позволяет объявлять локальные переменные в именованных блоках **begin…end** и **fork…join**. Часто локальные переменные объявляются для управления циклом. Локальная переменная предотвращает непреднамеренный доступ к переменной на уровне модуля с таким же именем, но используемой для других целей. Следующий фрагмент кода имеет объявления двух переменных, обе с именем i. Цикл **for**, в именованном блоке, будет использовать локальную переменную i, которая объявлена в именованном блоке, не трогая переменную i объявленную на уровне модуля.

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin: loop        // named block
      integer i;       // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

Иерархические ссылки на локальные переменные

Обращение к переменной, объявленной в именованном блоке, может осуществляться через иерархический путь, который включает имя блока. Обычно, только тестбенч или другие верификационные программы, обращаются к переменной с использованием иерархического пути. Иерархические ссылки не синтезируемы, и не показывают аппаратного поведения. Иерархический путь к переменной, внутри именованного блока, так же используется файлами VCD (Value Change Dump), временными диаграммами и другими отладочными инструментами. Следующий фрагмент тестбенча использует иерархические пути, чтобы вывести в консоль значение переменных с именем i, из предыдущего примера.

```
module test;
  reg clock;
  chip chip (.clock(clock));

  always #5 clock = ~clock;

  initial begin
    clock = 0;
    repeat (5) @(negedge clock) ;
    $display("chip.i = %0d", chip.i);
    $display("chip.loop.i = %0d", chip.loop.i);

     $finish;
   end
 endmodule
```

2.3.2 Локальные переменные в неименованных блоках

SystemVerilog расширяет возможности Verilog для объявления переменных в неименованных блоках. Синтаксис похож на объявления в именованных блоках:

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin              // unnamed block
      integer i;       // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

Иерархические ссылки к переменным в неименованных блоках

Локальные переменные в неименованных блоках не имеют иерархического пути

Поскольку нет имени блока, то к локальным переменным, в неименованном блоке, невозможно обращаться по иерархическому пути. Тестбенч или VCD файл не могут обращаться к локальной переменной.

Именованные блоки защищают локальные переменные

Объявление переменных в неименованных блоках может служить защитой локальных переменных от внешних, кросс-модульных обращений. Без иерархического пути, локальные переменные не могут быть доступны из вне.

Определяемые иерархические пути при отладке

Расширение, позволяющее объявлять переменную в неименованной области, не является уникальным для SystemVerilog. Язык Verilog имеет похожую ситуацию. Примитивы, определяемые пользователем (UDPs) могут иметь переменные объявленные внутри, но язык Verilog не требует чтобы имя экземпляра было назначено экземплярам примитива. Это также создаёт переменную в неименованной области. В этой ситуации программные инструменты будут определять имя экземпляра, для того чтобы обращаться к переменной внутри UDP. Программные инструменты могут также назначить определяемое имя неименованному блоку, для того чтобы отладочные утилиты обращались к локальным переменным в неименованном блоке. SystemVerilog стандарт не требует и не запрещает определение имени для неименованных блоков, также как и Verilog стандарт не требует, и не запрещает имён экземпляра для неименованных экземпляров примитива.

Секция 7.7 обсуждает именованные блоки; и секция 7.8 представляет выражение имён, которые могут быть использованы для предоставления имени области локальных переменных.

2.4 Единицы времени симуляции и точность

Язык Verilog не специфицирует единицы времени как значения. Значения времени просто сравниваются друг с другом. Задержка 3 больше чем задержка 1, и меньше чем задержка 10. Следующее выражение, тактовый генератор, который может использоваться в тестбенче:

```
forever #5 clock = ~clock;
```

Какой период такта ? 10 пикосекунд ? 10 наносекунд ? 10 миллисекунд ? В этом выражении нет информации, чтобы ответить на этот вопрос. Необходимо заглянуть в исходный код, чтобы определить единицы времени в это выражении.

2.4.1 Директива timescale в Verilog

Verilog специфицирует единицы времени

Вместо определения единиц времени со значениями, Verilog определяет единицы времени как команду для программного инструментария (software tool), используя директиву компилятора **`timescale**. Эта директива имеет две компоненты: единицы времени и точность. Компонента точности определяет временные единицы для представления времени после десятичной точки.
В следующем примере,

```
`timescale 1ns / 10ps
```

используется единица времени 1 наносекунда, и точность 10 пикосекунд.

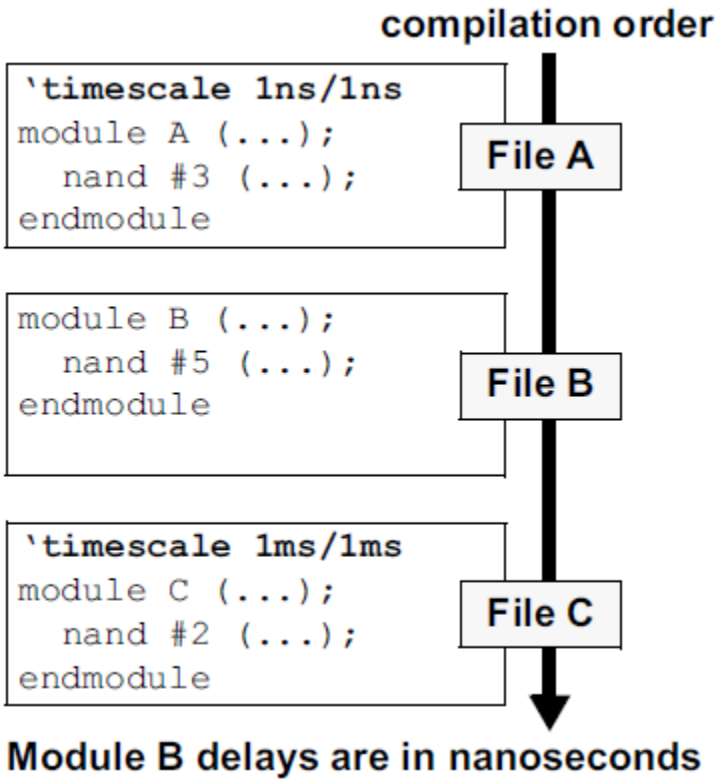
Несколько директив `timescale

Директива **`timescale** может быть определена в одном или более исходных файлах. Директивы с разными значениями могут быть определены для разных областей проекта. В этой ситуации, программный инструментарий должен найти общий делитель во всех специфицированных единицах времени, и масштабировать все задержки в каждой области проекта, в соответствии с общим делителем.

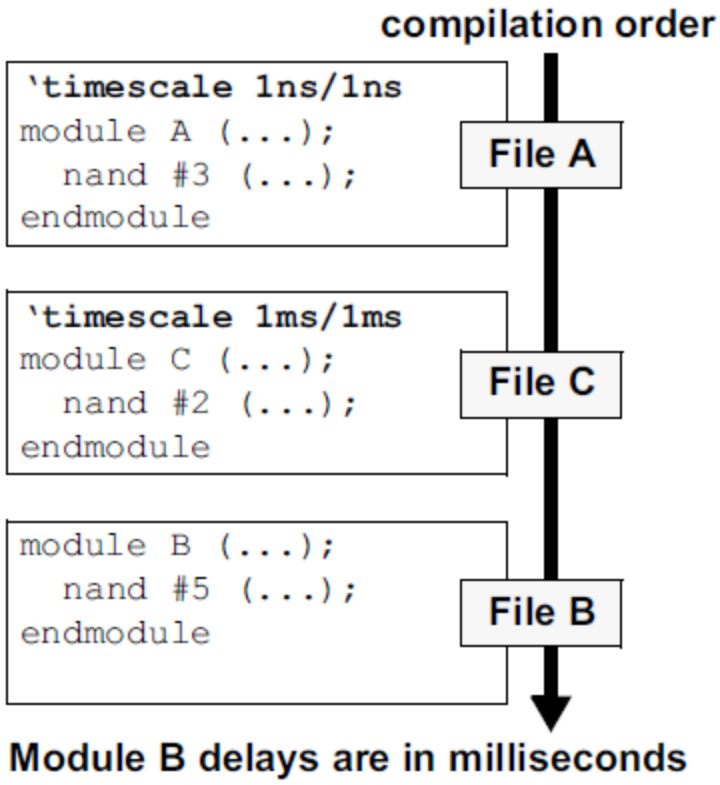
Директива `timescale создаёт зависимости

Директива **`timescale** не связана с конкретными модулями или файлами. Директива это команда для программного инструментария, и действует до тех пор, пока не придёт новая команда **`timescale**. Это создаёт зависимость, в какой последовательности программный инструментарий читает исходные файлы Verilog. Без директивы **`timescale** чтение файла происходит в соответствии с предыдущими файлами.

На следующем рисунке, файлы A и C содержат директиву `timescale, которая устанавливает единицы времени и точность для кода, который следует за директивой. Файл B не содержит директивы **`timescale**.



Если первым читается файл A, затем B и затем C, то для модуля B будет эффективна директива **`timescale** 1ns/1ns. Поэтому задержка в модуле B будет 5 наносекунд. Если исходные файлы читаются компилятором в другой последовательности, эффект от директив может отличаться. Рисунок ниже показывает порядок чтения файлов такой, A затем C и затем B.



В этом случае для модуля B эффективной будет директива **`timescale** 1ms/1ms. Поэтому задержка в модуле B будет 5 миллисекунд. Результаты симуляции этих двух вариантов будут разными.

2.4.2 Значения времени с временными единицами

Единицы времени определяются как часть значения времени

SystemVerilog позволяет определять единицы времени как часть значения времени.

```
forever #5ns clock = ~clock;
```

Это убирает все неопределённости в представлении времени. Предыдущий пример с тактовым генератором – период такта 10 наносекунд (5нс высокий уровень, 5нс низкий).

Единицы времени, которые можно использовать приведены в таблице.

Table 2-1: SystemVerilog time units

Unit	Description
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds
step	the smallest unit of time being used by the software tool (used in SystemVerilog testbench clocking blocks)

между значением времени и единицей времени ставить пробел нельзя

Когда единица времени определяется как часть значения времени, между значением и единицей времени пробел ставить нельзя.

```
#3.2ps      // legal
#4.1 ps     // illegal: no space allowed
```

2.4.3 Уровень видимости единицы времени и точности

В SystemVerilog можно определять единицы времени и точности, локально как часть модуля, интерфейса или программного блока, вместо команд для программного инструментария (интерфейсы обсуждаются в главе 10, программные блоки представлены в книге *SystemVerilog for Verification*).

timeunit и timeprecision как часть определения модуля

В SystemVerilog определение единиц времени получило дальнейшее расширение с ключевыми словами **timeunit** и **timeprecision**. Эти ключевые

слова используются для определения единицы времени и точности внутри модуля, как часть определения модуля.

```
module chip (...);
  timeunit 1ns;
  timeprecision 10ps;
  ...
endmodule
```

Ключевые слова timeunit и timeprecision связывают информацию о единице времени и точности напрямую с модулем, интерфейсом или программным блоком, вместо команд для программного инструментария. Это решает проблему с зависимостью, которая существует в Verilog, при использовании директивы **`timescale**.

Единицы времени, которые определяются ключевыми словами **timeunit** и **timeprecision**, это такие же единицы как в директиве **`timescale** в Verilog**.** Это единицы, которые приведены в таблице 2-1, исключая единицу **step.** Единицы времени и точности могут быть определены значениями 1, 10 или 100.

Выражения **timeunit** и **timeprecision** должны быть определены сразу после объявления модуля, интерфейса или программы, перед другими объявлениями или выражениями.

timeunit и timeprecision должны быть первыми

timeunit и **timeprecision** должны быть первыми выражениями внутри модуля, сразу после списка портов и перед другими объявлениями или выражениями. Нужно отметить, что Verilog разрешает объявления внутри списка портов. Это не работает для выражений **timeunit** и **timeprecision**. Эти выражения должны стоять сразу после объявления модуля. На пример:

```
module adder (input  wire [63:0] a, b,
              output reg  [63:0] sum,
              output reg         carry);
  timeunit 1ns;
  timeprecision 10ps;
  ...
endmodule
```

2.4.3 Единицы времени единицы-компиляции и точность

Внешние timeunit и timeprecision

Объявление **timeunit** и/или **timeprecision** может быть определено в области видимости единицы компиляции (описано ранее в этой главе в секции 2.2). Объявления должны стоять перед любыми другими объявлениями. Объявление **timeunit** или **timeprecision** в области видимости единицы компиляции применяется ко всем модулям, программным блокам и интерфейсам, которые не имеют локальных объявлений **timeunit** или **timeprecision**, и которые не скомпилированы с директивой **`timescale**.

В области видимости единицы компиляции может быть более одного выражения **timeunit** или **timeprecision,** пока все выражения имеют одинаковое значение.

Порядок поиска единицы времени и точности

Порядок поиска единицы времени и точности

В Systemerilog, единица времени и точности могут быть определены в нескольких местах. Поэтому, существует порядок поиска для определения единицы времени и точности.

- Используется специфицированная единица времени как часть значения времени

- Иначе, используется специфицированная локальная единица времени и точности в модуле, интерфейсе или программном блоке.

- Иначе, если объявление модуля или интерфейса вложено в другой модуль или интерфейс, используется единица времени и точности родительского модуля или интерфейса. Объявления вложенного модуля обсуждаются в главе 9, интерфейсы обсуждаются в главе 10.

- Иначе, используется единица времени и точности **`timescale**.

- Иначе, используется единица времени и точности определённая в области видимости единицы компиляции.

- Иначе, по умолчанию, используется единица времени и точности симулятора.

Обратная совместимость

Этот порядок поиска позволяет моделям, используя расширения SystemVerilog, быть полностью обратно совместимыми с моделями, написанными для Verilog. Следующий пример показывает смешивание задержек с единицами времени, объявлениями **timeunit** и **timeprecision** в модуле и на уровне области видимости единицы компиляции, и директивы компилятора **`timescale.** Комментарии показывают приоритет объявления.

Example 2-9: Mixed declarations of time units and precision (not synthesizable)

```
timeunit 1ns;          // external time unit and precision
timeprecision 1ns;

module my_chip ( ... );
  timeprecision 1ps; // local precision (priority over external)
  always @(posedge data_request) begin
    #2.5 send_packet;  // uses external units & local precision
    #3.75ns check_crc; // specific units take precedence
  end
  task send_packet();
    ...
  endtask
  task check_crc();
  ...
  endtask
endmodule

`timescale 1ps/1ps  // directive takes precedence over external
module FSM ( ... );
  timeunit 1ns;     // local units take priority over directive
  always @(State) begin
    #1.2 case (State) // uses local units & timescale precision
      WAITE: #20ps ...;  // specific units take precedence
    ...
  end
endmodule
```

2.5 Резюме

В этой главе были представлены пакеты SystemVerilog и пространство объявления $unit. Пакеты предоставляют пространство объявления, где могут быть определены типы определяемые пользователем, задачи, функции и константы. Определения в пакетах могут быть импортированы в

любое количество блоков проекта. Отдельные компоненты пакета могут быть импортированы, или определения пакета могут быть добавлены в проектный путь поиска, используя wildcard импорт.
Пространство объявления $unit предоставляет квази глобальное пространство объявления. Определения не находятся внутри проектного блока, тестбенч блока или пакета в пространстве единицы компиляции. При использовании $unit нужно избегать зависимостей и различий между компиляцией одного и нескольких файлов. Эта глава предоставила рекомендации по кодированию для правильного использования пространства единицы компиляции $unit.

Также, в SystemVerilog можно определять локальные переменные в неименованных блоках **begin…end**. Это упрощает объявление локальных переменных, а также скрывает локальную переменную. Локальные переменные в неименованных блоках защищены от чтения или модификации со стороны кода вне блока.

SystemVerilog расширяет возможности для определения единиц времени и точности в симуляции. Эти расширения убирают зависимости директивы **`timescale**.