Статическое в SystemVerilog

barkovian
13.10.2021 12:05

1. Введение

Условимся, что под словом “метод” мы будем иметь в виду и функции (`function`) и таски (`task`) класса. Функции и таски модуля будем называть “подпрограмма” (subroutine). Под словом “модуль” мы будем понимать также и интерфейс (`interface`), и программу (`program`). Различия между ними не являются существенными для нашего рассказа.

Ссылаясь на стандарт языка, будем иметь в виду документ IEEE Std 1800-2017.

На все примеры кода, кроме синтеза, прилагается ссылка на EDA Playground. Каждый пример был проверен в Riviera, Xcelium, Questa и VCS. Для самостоятельного воспроизведения примеров будет достаточно некорпоративного аккаунта EDA Playground с доступной Aldec Riviera.

Статической переменной в программировании называется переменная, создаваемая в момент запуска программы. Она доступна для работы с момента запуска и до завершения программы. Противоположностью статических переменных являются автоматические. Называются они так потому, что память для них выделяется *автоматически*, когда исполнение программы доходит до области видимости данной переменной, а также освобождается при выходе из области видимости.

В SystemVerilog статическими могут быть не только переменные, но и методы, а также целые блоки, такие как модуль и программа.

Мы начнём с наиболее практического вопроса: как работают статические функции и задачи. Сначала рассмотрим этот вопрос в контексте модулей, а потом перейдём к классам.

2. Модули
Определение

Сигнатура автоматической подпрограммы выглядит так:

```
task automatic foo(int a, int b);
function automatic int bar(int a, int b);
```

Обратите внимание, что слово `automatic` следует после слова `task`.

Если `automatic` опустить или заменить на `static`, то подпрограмма станет статической.

Время жизни локальных переменных и входных аргументов подпрограммы повторяет время жизни подпрограммы, однако для локальных переменных можно задать время жизни явно. Так, если подпрограмма статическая, то и её переменные будут статическими, если не стоит ключевое слово `automatic`. И наоборот, все переменные автоматической функции являются автоматическими, если объявлены без ключевого слова `static`.

Хотя мы и говорим, что подпрограмма может быть статической или автоматической, вместо “статическая подпрограмма” правильнее было бы говорить “подпрограмма, у которой все переменные являются статическими”. Однако это было бы слишком громоздко, а о чём на самом деле идёт речь, вы уже понимаете.

Стандарт не подразумевает возможности задать явно время жизни входных аргументов. Впрочем, этого можно добиться от Xcelium’a. Другие симуляторы не пошли на это преступление.

Примеры

Рассмотрим следующий пример.

Static and Dynamic tasks example

```
module test();
```

```
  task static static_add(int a, int b);
    #2;
    $display("Sum: %0d", a+b);
  endtask

  task automatic automatic_add(int a, int b);
    #2;
    $display("Sum: %0d", a+b);
  endtask

  initial begin
    $display("Test for static");
    fork
      begin
        static_add(1,2);
      end
      begin
        #1;
        static_add(3,4);
      end
    join

    $display("Test for automatic");
    fork
      begin
        automatic_add(1,2);
      end
      begin
        #1;
        automatic_add(3,4);
      end
    join

  end

endmodule
```

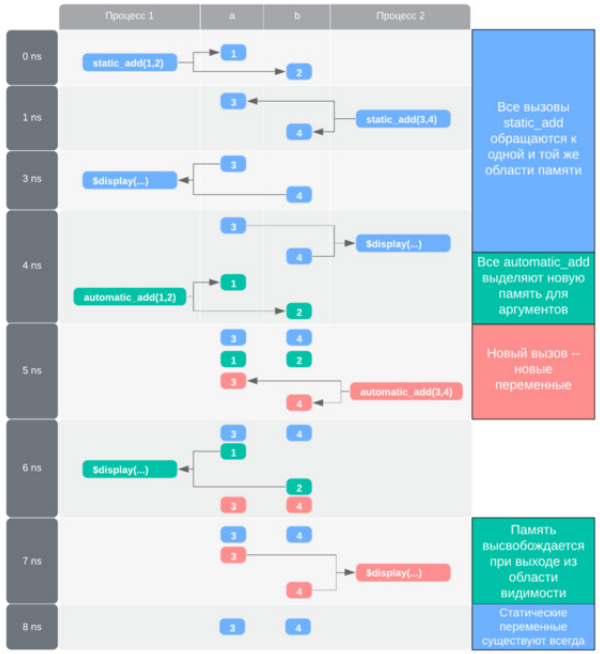
Симуляция даёт нам такой вывод:

```
  # KERNEL: Test for static
# KERNEL: Sum: 7
# KERNEL: Sum: 7
# KERNEL: Test for automatic
# KERNEL: Sum: 3
# KERNEL: Sum: 7
```

Входные аргументы подпрограммы `static_add` являются статическими, то есть всегда ссылаются на одну и ту же область памяти. Первый вызов произошёл в 0 нс, аргументы стали равны 1 и 2. В 1 нс в те же области памяти было записано 3 и 4. Так как обращение к переменным подпрограммы совершается только с задержкой 2 нс, то оба вызова увидят одинаковое значение переменных a и b.

Напротив, для автоматического брата нашей подпрограммы, `automatic_add`, выделение памяти для `a` и `b` происходит при каждом вызове, поэтому каждый вызов увидит собственное значение входного аргумента.

Более детальное объяснение происходящего представлено на следующей диаграмме.



Возможность безопасного вызова одной подпрограммы из разных процессов называется реентерабельностью. Использование таких подпрограмм уменьшает количество неожиданных побочных эффектов и является хорошим стилем.

Маленькое отступление. Не нужно бояться слов “выделение памяти” и думать, что теперь всё станет медленно. Место для локальных переменных выделяется в стеке (Stack-based memory allocation - Wikipedia). Это является естественным образом работы с памятью для большинства языков программирования, от ассемблера и C, до SystemVerilog и Haskell.

В конце статьи мы поставим небольшой эксперимент для изучения данного вопроса.

Рассмотрим пример модификации времени жизни локальных переменных функции ключевыми словами: Static and Dynamic variables example

```
module foo();

  function void good_increment(int inc_value);
    int counter;
    counter += inc_value;
    $display("Incremented value: %0d", counter);
  endfunction

  function void bad_increment(int inc_value);
    automatic int counter;
    counter += inc_value;
    $display("Incremented value: %0d", counter);
  endfunction

  initial begin
    $display("Test for static");
    good_increment(1);
    good_increment(2);

    $display("Test for automatic");
    bad_increment(1);
    bad_increment(2);
  end

endmodule

module tb;
  foo u_foo();
  initial begin
    #1;
    $display("We can access counter directly: %0d", ++u_foo.good_increment.counter);
  end
endmodule
```

Как и ожидалось, статическая переменная в `good_increment` сохраняет своё значение между вызовами, в отличие от автоматической в `bad_increment`. Ещё одна интересная деталь: к *статической переменной можно обратиться по иерархическому имени*. Отсюда можно заключить, что статические переменные в различных экземплярах одного модуля также являются различными. Действительно, в стандарте (13.3.2) есть на это явное требование. Подобное иерархическое обращение к автоматической переменной невозможно, так как переменной просто не существует (не выделена память) до входа программы в её область видимости.

Рассмотрим ещё один пример, где без автоматической переменной не обойтись. Довольно типичная задача — запустить несколько параллельных процессов из цикла, передав каждому процессу значение переменной цикла.

Fork Loop in a Module

```
module tb;

  task automatic wait_task(int delay);
    $display("%0d ns: Wait for %0d ns started", $time(), delay);
    #(delay * 1ns);
    $display("%0d ns: Wait for %0d ns completed", $time(), delay);
  endtask

  initial begin
    for(int i = 0; i < 3; i++) begin
      fork
        automatic int k = i;
        wait_task(k);
      join_none
    end
    wait fork;
  end

endmodule
```

Наивным решением было бы вызвать `wait_task(i)`, не используя промежуточные переменные. Однако новые процессы будут запущены только когда родительский приостановит работу (Таблица 9.1 стандарта), а в это время значение `i` уже будет равно 3. То есть, `wait_task(3)` будет запущен трижды. Читатель может внести соответствующие изменения в код примера и увидеть следующий вывод:

```
    # KERNEL: 0 ns: Wait for 3 ns started
  # KERNEL: 0 ns: Wait for 3 ns started
```

```
# KERNEL: 0 ns: Wait for 3 ns started
# KERNEL: 3 ns: Wait for 3 ns completed
# KERNEL: 3 ns: Wait for 3 ns completed
# KERNEL: 3 ns: Wait for 3 ns completed
```

Чтобы добиться желаемого поведения, следует объявить новую переменную внутри `fork` и инициализировать её значением `i`. По стандарту (9.3.2), инициализация будет выполнена сразу же. Однако в модулях все переменные по умолчанию статические, поэтому чтобы каждый процесс получил своё значение, нужно явно прописать `automatic`.

Отметим ещё одну деталь из данного примера. Несмотря на то, что переменные модуля по умолчанию статические, из-за инициализации мы не сможем просто опустить `automatic`. Стандарт требует явного указания времени жизни переменной в статической подпрограмме или блоке, если она имеет инициализацию в своём объявлении. Таким образом пользователь указывает, хочет ли он инициализировать переменную на каждой итерации (`automatic`), или только один раз (`static`). Однако в данном случае использовать `static` не получится, так как стандарт запрещает инициализацию статической переменной значением автоматической. Это вполне понятное ограничение: статические переменные инициализируются в момент запуска симуляции, а в это время память для автоматических ещё не выделена. Ни один из проверенных симуляторов не скомпилировал ни вариант со static, ни вариант без модификатора.

Мы видели, как применяется модификатор времени жизни automatic к переменным и подпрограммам, однако он может также применяться и к целому модулю. Это меняет время жизни по умолчанию со статического на автоматическое для переменных подпрограмм и блоков `initial` и `always`.

Попробуйте в предыдущем примере поменять объявление модуля на

```
module automatic foo();
```

Пройдёт ли компиляция? Почему? Исправьте и проверьте, что вывод соответствует ожиданиям.

Рассмотрим ещё один пример модификации времени жизни для всего модуля. На этот раз с `always`.

Automatic always

```
module automatic foo;
  always #1 begin
    int x;
    $display("Automatic: %0d", x++);
  end

  always #1 begin
    static int x;
    $display("Static: %0d", x++);
  end
```

```
  initial begin
    #10;
    $finish;
  end
endmodule
```

Иметь все подпрограммы с автоматическим временем жизни по умолчанию может быть удобно. Так ли удобно это для `always`? Не уверен.

Особенности

Из различий во времени жизни переменных следует и другое различие между статическими и автоматическими подпрограммами: безопасность рекурсивного вызова. Стандарт не запрещает рекурсивный вызов статических подпрограмм, однако неосторожность может привести совсем не к тому результату, на который мы рассчитывали.

В этом примере за основу был взят код из пункта 13.4.2 стандарта.

```
module tryfact;

    function integer factorial (input [31:0] operand);
      if (operand >= 2) begin
//         $display("Entering operand = %0d", operand);
        factorial = factorial (operand - 1) * operand;
//         $display("Exiting operand = %0d", operand);
      end else
        factorial = 1;
    endfunction: factorial

  integer result;
  initial begin
    result = factorial(5);
    $display("%0d factorial = %0d", 5, result);
  end
endmodule: tryfact
```

Здесь выражение `factorial = factorial (operand - 1) * operand;` не может быть вычислено, пока не будет совершён следующий вызов `factorial`. Но этот вызов перепишет входной аргумент `operand`. Раскомментируйте строки с `$display`, чтобы увидеть этот эффект.

Когда использовать?

Когда нужно использовать статические подпрограммы, а когда — автоматические? Мне не известно случаев, когда была бы нужна именно статическая подпрограмма. Если нет стопроцентной уверенности, что нужна именно статическая подпрограмма, используйте автоматическую. Из-за своей реентерабельности они являются более безопасными — нет нужды отслеживать и синхронизировать все точки вызова.

Напротив, применение `automatic` ко всему модулю может быть опасно из-за изменения времени жизни переменных в блоке always. Такое поведение будет неожиданным для читателей кода, да и сам автор может забыть об этом нюансе.

3. Классы
Определение

Статический метод класса объявляется следующим образом:

```
static task static_add(int a, int b, output int c);
static function int static_add(int a, int b);
```

Обратите внимание, что ключевое слово `static` ставится до слова `task/function`. В модулях — наоборот: `static` ставилось после.

Если `static` опустить, то метод станет нестатическим. Ключевое слово `automatic` в этом контексте не допускается.

Стоп... Почему не допускается? Зачем мы говорим “нестатический” вместо “автоматический”? Почему ключевое слово ставится не там? Если вам уже кажется, что со статическими методами всё совершенно не так, как с подпрограммами в модулях, то вы совершенно правы. Мы разберёмся со всеми этими вопросами. В первую очередь, давайте посмотрим, что такое статические методы и как они работают.

Статический метод класса связан с самим классом, а не экземпляром класса. Это означает следующее:

1. Статический метод можно вызвать не имея ни одного экземпляра, используя имя класса как область видимости.
2. Статический метод не может обратиться к нестатическим свойствам класса.

С точки зрения работы никаких других отличий статических методов от нестатических нет.

Объявление статической переменной класса выглядит схожим образом.

```
class foo;
  static int bar;
endclass
```

Примеры

Для иллюстрации внесём минимальные изменения в первый пример, превратив модули в классы: Static and Dynamic tasks example for class.

```
class foo;

  static task static_add(int a, int b);
    #2;
    $display("Sum: %0d", a+b);
  endtask

  task automatic_add(int a, int b);
    #2;
    $display("Sum: %0d", a+b);
  endtask

  task my_initial;
    $display("Test for static");
    fork
      begin
        static_add(1,2);
      end
      begin
        #1;
        static_add(3,4);
      end
    join

    $display("Test for automatic");
    fork
      begin
        automatic_add(1,2);
      end
      begin
        #1;
        automatic_add(3,4);
      end
    join
  endtask

  function new();
    fork
      my_initial();
```

```
    join_none
  endfunction

endclass

module tb;
  foo foo_h;

  initial begin
    foo_h = new();
    #10;
    $display("Call using only a class name");
    foo::static_add(2, 2);
  end

endmodule
```

Вывод:

```
    # KERNEL: Test for static
  # KERNEL: Sum: 3
  # KERNEL: Sum: 7
  # KERNEL: Test for automatic
  # KERNEL: Sum: 3
  # KERNEL: Sum: 7
  # KERNEL: Call using only a class name
  # KERNEL: Sum: 4
```

Видим, что нет никакой разницы между работой статического и не-статического методов. Оба они отработали как автоматическая подпрограмма в модуле. Однако экземпляр класса не является необходимым для использования статического метода:

```
foo::static_add(2, 2);
```

Вызвать таким образом не-статический не получится — это ошибка компиляции. Более того, вызывать статические методы лучше именно так, с использованием имени класса, а не объекта. Так для читателя сразу становится ясно, что этот метод — статический.

Статическое свойство класса подчиняется тем же правилам, что и статические методы — принадлежит классу, а не экземпляру, то есть доступ к ней возможен по имени класса, и каждый экземпляр класса при доступе к статическому свойству будет обращаться к одной и той же области памяти.

Статической можно объявить и переменную в методе класса (не важно, является ли сам метод статическим или нет). И в этом случае все экземпляры класса будут обращаться к одной и той же области

памяти.

Проиллюстрируем на примере: Static and Dynamic variables example for class.

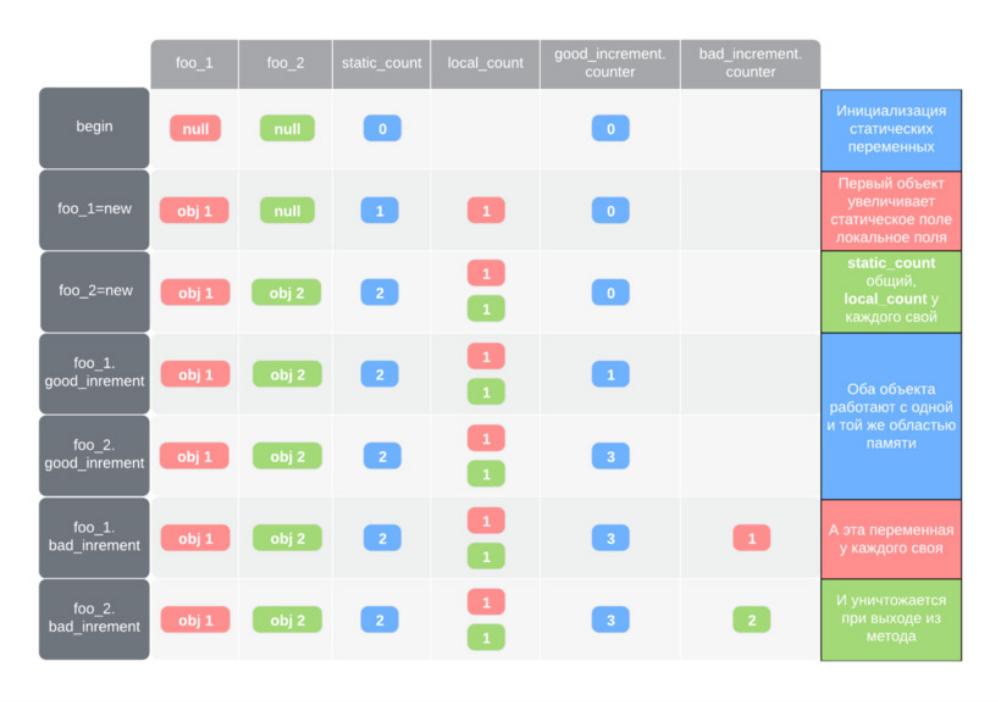
```
class foo;

  static int static_count;
  int local_count;
  function new();
    static_count++;
    local_count++;
  endfunction

  function void good_increment(int inc_value);
    static int counter;
    counter += inc_value;
    $display("Incremented value: %0d", counter);
  endfunction

  function void bad_increment(int inc_value);
    int counter;
    counter += inc_value;
    $display("Incremented value: %0d", counter);
  endfunction
endclass

module tb;
  foo foo_1;
  foo foo_2;
  initial begin
    foo_1 = new();
    $display("Create the first instance and check count.  Static: %0d Local: %0d", foo_1.static_count
    foo_2 = new();
    $display("Create the second instance and check count. Static: %0d Local: %0d", foo::static_count,
    $display("Test for static");
    foo_1.good_increment(1);
    foo_2.good_increment(2);

    $display("Test for automatic");
    foo_1.bad_increment(1);
    foo_2.bad_increment(2);
```

```
  end
endmodule
```

Здесь нужно обратить внимание на несколько вещей.

Во-первых, хорошо видно, что статическая переменная класса `static_count` является общей для всех экземпляров. Во-вторых, обратиться к ней мы можем как через экземпляр, так и через имя класса. В-третьих, повторные вызовы не-статического метода `good_increment` увеличивают значение одной и той же переменной, как и ожидалось.

На следующей диаграмме приведён подробный разбор примера.



Вспомним пример с запуском процессов из цикла. Так как в подпрограммах модуля переменные по умолчанию имеют статическое время жизни, нам требовалось объявлять промежуточную переменную как `automatic`. В классах это излишне, так как переменные методов являются по умолчанию автоматическими, то есть достаточно будет такого кода:

```
for(int i = 0; i < 3; i++) begin
  fork
    int k = i;
```

```
    wait_task(k);
  join_none
end
wait fork;
```

Особенности

Теперь мы понимаем, как работают статические методы, и можем вернуться к вопросу различий между статической подпрограммой модуля и статическим методом класса. В стандарте имеется явное указание на отличие в пункте 8.10. В первом случае слово “статический” описывает время жизни переменных подпрограммы. Во втором случае — время жизни метода в классе. Мы не можем вызвать нестатический метод до создания экземпляра, в этом смысле можно сказать, что нестатический метод начинает своё существование с созданием объекта и заканчивает с его уничтожением. Статический метод же существует всегда.

По этой же причине слово “автоматический” нельзя использовать для описания нестатических методов. Строго говоря, автоматическими могут быть только переменные, хоть мы и называем так подпрограммы для простоты речи. Заметим, что локальные переменные методов являются автоматическими по умолчанию и в этом смысле метод можно назвать автоматическим. Именно это имеет в виду стандарт в пункте 8.6, когда говорит, что методы имеют автоматическое время жизни.

Остановимся подробнее на автоматическом времени жизни метода. Стандарт явно запрещает задавать методу класса статическое время жизни (в смысле подпрограмм модуля), что указано в пунктах 8.6 и 8.10. Однако все 4 симулятора смогли скомпилировать даже

```
static task static
```

Когда-то эта конструкция была допустима, однако как минимум с версии стандарта 2012 года писать `task/function static` в классе больше нельзя. Думаю, не стоит объяснять, что пользоваться такой нелегальной возможностью симулятора не стоит. Если вдруг вы видите, что от метода класса вам нужно такое же поведение, как от статической подпрограммы модуля, этого можно легко добиться, задействуя статические методы вместе со статическими переменными класса.

Возможно, этой вольностью симуляторов объясняется и ещё одно нарушение стандарта. Попробуем обратиться к статической переменной, объявленной внутри метода: Try to access a static variable in a method

```
class foo;
  static function void increment();
    static int cnt = 0;
    $display(++cnt);
  endfunction
```

```
  function void local_increment();
    static int cnt = 0;
    $display(++cnt);
  endfunction

endclass

module tb;
  foo bar;
  initial begin
    $display("Non-static:");
    foo::increment();
    $display(++foo::increment.cnt);
    $display("Static:");
    bar = new();
    bar.local_increment();
    $display(++bar.local_increment.cnt);
  end
endmodule
```

Все 4 симулятора смогли обратиться к статической переменной статического метода, Riviera и VCS смогли даже обратиться к статической переменной не-статического метода, хотя я не думал, что эту ахинею скомпилирует хоть кто-то. Из пункта 6.21 стандарта следует, что обращаться по иерархическому имени к статической переменной автоматической функции/таски нельзя, а, как мы уже видели, методы класса в обязательном порядке имеют автоматическое время жизни. В любом случае, обращения к статическим переменным любых методов извне самого метода стоит избегать: это увеличивает связность и ломает само понятие локальной переменной.

Мы также избегаем называть нестатические методы динамическими. Во-первых, это может создать ненужные ассоциации со статической/динамической диспетчеризацией (о которой мы будем вынуждены вкратце упомянуть в разделе Эксперименты), а во-вторых, динамический — это то, что изменяется во время исполнения программы, как динамический массив. К методам это не имеет отношения.

Когда использовать?

В каких случаях нужно использовать статические методы? Как правило, в двух ситуациях: когда создание экземпляра класса через new по какой-то причине не является возможным, или когда метод использует только статические поля класса и входные аргументы, не обращаясь к полям экземпляра. Рассмотрим примеры таких ситуаций.

Паттерн синглтон (singleton, одиночка)

Иногда нужно, чтобы на весь тестбенч был только один экземпляр некоторого класса, а доступ к нему был у всех желающих. Это может быть некая глобальная конфигурация, или класс для доступа к файловой системе. В таком случае конструктор должен быть вызван только один раз, а повторные вызовы должны быть невозможны.

```
class foo;
  protected static foo singleton;
  protected function new(); endfunction
  static function foo get();
    if (singleton == null)
      singleton = new();
    return singleton;
  endfunction
endclass
```

Конструктор объявлен как `protected`, что делает невозможным создание объекта где-либо, кроме как в самом классе. Единственный указатель на объект так же защищён от возможного обнуления. В любой точке кода, где виден класс `foo`, можно вызвать `foo::get()` и получить единственный экземпляр класса.

Можно было бы создавать объект не в методе get, а при инициализации:

```
protected static foo singleton = new();
```

Однако Questa это не компилирует, ссылаясь на то, что конструктор объявлен как `protetcted`. Стандарт в пункте 8.18 разрешает обращение к защищённым методам и полям класса внутри класса, даже если это другой экземпляр, поэтому правомерность поведения Questa сомнительна.

Паттерн фабрика

Разберём ещё две ситуации, в которых непосредственное использование конструктора невозможно.

- Нужно подменить создаваемый класс на его наследник. Как это сделать без модификации кода создания?
- В SystemVerilog невозможна перегрузка (overload) методов, то есть создание методов с одинаковым названием, но различными входными аргументами. Как обойти это ограничение для конструкторов?

В обоих случаях можно воспользоваться партерном “фабрика”. Если коротко, то это класс или метод, который создаёт экземпляры класса. Этот паттерн реализован в UVM и прекрасно справляется с первой задачей. Приведём пример решения второй задачи на примере класса, который хранит в себе текущее время.

```
class my_time;
```