Языки описания схем

mk.cs.msu.ru → Лекционные курсы → Языки описания схем

Блок K7

Ещё немного о Verilog
и
пара слов о SystemVerilog

Лектор:
Подымов Владислав Васильевич
E-mail:
valdus@yandex.ru

ВМК МГУ, 2024/2025, осенний семестр

Задачи (tasks)

Функция:

```
function [7:0] sum;
  input [7:0] x, y;
  sum = x + y;
endfunction
always @* #1
  w = sum(a, b);
```

Задача похожа на функцию с такими отличиями:

- Вместо `function`/`endfunction` пишется `task`/`endtask` и без указания типа
- В задаче нет результата и может быть любое число выходных портов (с присваиваниями в них в теле)
- Выполнение тела задачи не обязано завершаться в том же регионе, когда вызвано
- Задача вызывается не в выражении, а как процедурная команда

Задачи (tasks)

Пример: задача вычисления суммы и разности чисел:

```
task sumdiff;
  input [7:0] x, y;
  output [8:0] sum, diff;
  begin
    sum = x + y;
    diff = x - y;
  end
endtask

always @* #1
  sumdiff(a, b, s, d);
```

После слова `task` можно добавить `automatic` с тем же смыслом, что и для функций

Задачи поддерживаются только со словом `automatic` и с поправкой на ограничения синтезиуремости процедур, в которых они вызываются (виды присваиваний и т.п.)

Системные задачи и функции

Системные задачи и функции — это встроенные (заданные языком) задачи и функции

Имена всех этих задач и функций начинаются с символа `$`:

- `$display`, `$monitor`, `$strobe` — системные задачи
- `$clog2` — системная функция
- ...

По умолчанию системные функции не поддерживаются, а вызовы системных задач игнорируются

Исключения:

- Про `$signed` и `$unsigned` рассказывалось, что они поддерживаются
- Про `$clog2` рассказывалось, что эта функция фактически обычно поддерживается, но формально нет
- Рассказанное дальше про системные задачи `$readmemb` и `$readmemh`

$readmemb, $readmemh

```
$readmemb(<имя файла>, <имя массива шин>);
```

При выполнении этого вызова в *массив шин* присваиваются значения, записанные в особом текстовом формате в *файле*:

- Допускаются только символы 0, 1, x, z, _ (как в двоичных константах[1]), пробельные символы и комментарии
- Пробельными символами и комментариями разделяются записи чисел как в константах после «'»
- Эти числа присваиваются в соответствующие (по порядку) элементы массива

1 Я про это не рассказывал, но в записи констант можно визуально разделять разряды символов _

$readmemb, $readmemh

Запись

initial $readmemb(<*имя файла*>, <имя массива шин>);

поддерживается для синтеза ROM со значениями, взятыми из *файла*, в *массиве*

Задача `$readmemh` устроена так же, только для шестнадцатеричной системы счисления

Другие циклы

Помимо цикла `for` в языке содержатся также и другие виды циклов

Все циклы, кроме рассказанного про `for`, не поддерживаются, но могут быть полезны в симуляции

`while(<`*выражение*`>) <`*команда*`>`

Семантика очевидна?

`forever <`*команда*`>`

Эквивалентно `while(1)` *команда*

`repeat(<`*выражение*`>) <`*команда*`>`

Команда выполняется столько раз, каково значение *выражения*

Пример: более удобное моделирование тактового сигнала

```
initial begin
  clk = 0;
  forever #1 clk = !clk;
end
```

Вступительные слова о SystemVerilog (S$\mathcal{V}$)

Бурное развитие «чистого» языка C завершилось много лет назад (в 1999 году, после чего развитие уже не было таким бурным, хотя новые стандарты и выходили), после чего начало бурно разиваться «улучшенный и расширенный вариант» этого языка C++

Аналогично около 2005 года завершилось бурное развитие «чистого» $\mathcal{V}$, и с тех пор по сей день бурно развивается «улучшенный и расширенный вариант» $\mathcal{V}$, получившей название SystemVerilog (будем кратко записывать как S$\mathcal{V}$)

Точность этой аналогии можно подчеркнуть тем, что рабочее название этого языка в начале разработки было «Verilog++», но от этого названия (*увы*) отказались, вероятно, из соображений маркетинга (*не очень красиво звучит, если прочитать*)

Вступительные слова о SystemVerilog (S$\mathcal{V}$)

$\mathcal{V}$ целиком включается в S$\mathcal{V}$, поэтому (*в отличие от C++ по сравнению с C*) можно полноценно считать S$\mathcal{V}$ расширением языка $\mathcal{V}$ без каких бы то ни было поправок и без «переучивания»

Расширение S$\mathcal{V}$ оказалось настолько удобным и популярным, что «в современном схемном мире» зачастую, говоря $\mathcal{V}$, на самом деле имеют в виду S$\mathcal{V}$

При этом средства работы, заявленные как для $\mathcal{V}$, так и для S$\mathcal{V}$, как правило, умеют

- строго больше того, что допускается в $\mathcal{V}$ и при этом
- строго меньше того, что допускается в S$\mathcal{V}$

Вступительные слова о SystemVerilog (S$\mathcal{V}$)

Кроме того, для S$\mathcal{V}$ нет единого прописанного синтезируемого фрагмента (даже такого местами «туманного» и «нечёткого», как в стандарте синтеза для $\mathcal{V}$)

Поэтому в разговоре про S$\mathcal{V}$ всегда нужно иметь в виду поправку на то, что фактический набор того, что можно использовать как для симуляции, так и для синтеза существенно зависит от используемых средств обработки кода

Но тем не менее есть «популярные» конструкции языка, которые можно использовать «с высокой степенью уверенности»

S$\mathcal{V}$: Уточнённые `always`

Одна и та же процедура `always` в $\mathcal{V}$ используется для описания как комбинационных подсхем, так и регистров и вовсе не схем

Кроме того, в поддерживаемых вариантах использования этой процедуры списки событий оказываются целиком или частично избыточными

Всё это может приводить (и приводит) к ошибкам, допущенным по невнимательности

Поэтому в S$\mathcal{V}$ добавлены «уточнённые» варианты процедуры `always`, в которых явно отражён вид подсхемы, предполагаемый для реализации этой процедурой

SV: Уточнённые `always`

`always_comb`

Это ключевое слово, заменяющее запись `always @*` для реализации комбинационной схемы

Пример: подсхема, реализующая схему со входами `x`, `y` и выходами `s` и `d`, реализующими соответсвенно сумму и разность значений `x`, `y`

```
always_comb begin   s = x + y;   d = x - y; end
```

`always_latch`

Это ключевое слово, заменяющее запись `always @*` для реализации асинхронного регистра

Пример: D-защёлка

```
always_latch   if(en) q <= d;
```

SV: Уточнённые `always`

`always_ff`

Это ключевое слово, заменяющее запись `always` для реализации синхронного регистра (быть может, с дополнительными асинхронными входами)

Пример: D-триггер

```
always_ff @(posedge clk)   q <= d;
```

Ещё один пример: D-триггер с асинхронным сбросом

```
always_ff @(posedge clk iff rst == 0, posedge rst)
  if(rst) q < = 0;
  else q <= d;
```

`<`*событие*`> iff <`*условие*`>`

Эта запись имеет «естественный» смысл: *событием* запускается выполнение процедуры $\Leftrightarrow$ *условие* выполнено

SV: Тип `logic`

Наверняка многие уже повстречались с тем, что приходится прикладывать заметные усилия для того, чтобы правильно расставить категории типов точек, хотя можно было бы и не прикладывать, т.к. категории однозначно задаются тем, как используются эти точки

`logic`

Это тип, аналогичный `reg` и `wire`, но выражающий надежду разработчика на то, что средство обработки кода сможет само определить (*вывести*) подходящую категорию

Пример: модуль сумматора

```
module adder(x, y, z);
  input logic [7:0] x, y;
  output logic [8:0] z;
  always_comb
    z = x + y;
endmodule
```

SV: Перечисления

В *грамотной* разработке схемы весьма нередко встречается необходимость реализовать символьный автомат

В *грамотно* составленной типовой реализации состояниям автомата присваиваются имена: локальные параметры с заданными числовыми значениями

Если не использовать локальные параметры для имён, то вероятность ошибиться с соответствием состояний и чисел очень высока

Но если использовать, то вероятность ошибиться хотя и ниже, но тоже высока:

```
localparam [2:0]
  WAIT = 3'b001,
  LOAD = 3'b010,
  DONE = 3'b001; // так зарябило в глазах от 0 и 1,
                 что проглядел ошибку
```

SV: Перечисления

В C++ для таких случаев есть перечисления, автоматически назначающие именам уникальные числа

И в SV есть то же и для тех же целей, только с явным указанием чисел (и синтаксическим запретом писать одинаковые числа):

```
enum logic [2:0] {
  WAIT = 3'b001, LOAD = 3'b010,
  DONE = /*3'b001 - ошибка компиляции*/ 3'b011
} state;
```

Разумеется, `enum` можно использовать не только в реализации автомата, но и в целом для сокрытия неважных числовых значений под содержательно важными наглядными названиями

SV: Структуры и синонимы типов (typedef)

В больших и сложных схемах полезно иметь не только «плоские» типы точек и шин, но и то, что известно по языку C как структура (аггрегированный тип)

В SV структуры можно объявлять примерно так же, как и в C/C++:

```
typedef struct {
  logic [7:0] first, second;
} pair;

pair x;
always @(posedge clk) begin
  x.first <= a;
  x.second <= x.first;
end
```

Кроме того, typedef можно использовать для объявления синонимов типов, как в C/C++:

```
typedef logic [7:0] byte_t;
```

SV: Пакеты

Пакет — это конструкция того же уровня, что и модуль, вместо `module`/`endmodule` обрамлённая словами `package`/`endpackage` и предназначенная для объявления всего того, что можно впоследствии «подключить» в модуль или другую конструкцию того же уровня, чтобы избежать повторных объявлений

В пакете можно объявлять:

- Параметры
- Константные значений (*не будем подробно это обсуждать*)
- Перечисления и структуры (в общем, всё то, что записывается возле `typedef`)
- Задачи и функции
- Подключение элементов других пакетов

SV: Пакеты

Пример

Объявление пакета:

```
package junk_defs;
  parameter PAR = 1;
  typedef logic [7:0] byte_t;
  typedef logic [31:0] word_t;
  function automatic logic parity(input [7:0] d);
    return ^d;
  endfunction
endpackage
```

(Да, в SV есть и return с тем же смыслом, что в C/C++)

Использование пакета:

```
module M();
  input junk_defs::byte_t x;
  import junk_defs::PAR;
  always @(posedge clk)
    x <= PAR;
endmodule
```

SV: Интерфейсы

Интерфейс — это ещё одна конструкция того же уровня, что и модуль, предназначенная для инкапсуляции объявлений точек и портов и обрамлённая словами `interface`/`endinterface` вместо `module`/`endmodule`

Пример

Объявление интерфейса:

```
interface spi_bus;
  logic clk, miso, mosi;
  modport m_ports(output clk, mosi, input miso);
  modport s_ports(input clk, mosi, output miso);
endinterface
```

Ключевое слово `modport` позволяет «настраивать» направления портов интерфейса при его использовании

Использование интерфейса в объявлении модуля:

```
module master(spi_bus spi, input clk);
```

Использование экземпляра интерфейса в экземпляров модулей:

```
spi_bus bus(); master m(.spi(bus.m_ports));
```

SV: Много другого

Лаконичные назначения портов

Блочные (многомерные) шины

Заполняющая константа

Явное преобразование типов

++, --

Равенство шаблону

Оператор проверки принадлежности множеству

Асимметричный аналог `casex`

Оператор перепаковки шины

Оператор вычисления ширины выражения

Копирование массива одним присваиванием

... ...