Языки описания схем

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

Блок 24

Verilog:
Синтаксический сахар
и ещё несколько возможностей языка

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

ВМК МГУ, 2023/2024, осенний семестр

Вступление

На данный момент про $\mathcal{V}$ рассказано всё необходимое для реализации **любой** цифровой схемы в поддерживаемом фрагменте языка, за исключением нескольких особых случаев:

- Схемы с высоким импедансом
  - (*это будет в конце курса*)
- RAM и ROM
  - (*в этом курсе они не пригодятся, и это в целом несложно, в стандарте всё явно написано*)

Кроме того, рассказано всё необходимое для реализации сценариев выполнения схемы с выводом в консоль и генерацией диаграмм сигналов в рамках отладки

Но помимо «всего необходимого» в языке предусмотрены конструкции и возможности, делающие реализацию и отладку схемы не только возможной, но и (*до некоторой степени*) удобной

Неявное объявление типа `wire`

Как упоминалось ранее, в $\mathcal{V}$ можно использовать точки, не объявляя их, и такие точки по умолчанию имеют тип `wire`

Эта возможность дополняется и другой, позволяющей избежать многократного повторения слова `wire` во всевозможных объявлениях

Во многих случаях там, где должно быть написано слово `wire`, можно не писать это слово, если рядом стоят другие ключевые слова, применяющиеся вместе с `wire`

Например:

`input <точка>;`	= `input wire <точка>;`
`output <точка>;`	= `output wire <точка>;`
`input [msb:lsb] <точка>;`	= `input wire [msb:lsb] <точка>;`
`output [msb:lsb] <точка>;`	= `output wire [msb:lsb] <точка>;`

Инициализация точек

В $\mathcal{V}$ можно писать код, схожий с инициализацией переменных в C/C++

Для переменных смысл этих конструкций близок к C/C++,
а для соединений — не очень

Такая инициализация может считаться
сокращением для типовых конструкций:

```
<тип переменной> <точка> = <выражение>;
```
=
```
<тип переменной> <точка>;
initial <точка> = <выражение>;
```

```
<тип соединения> <точка> = <выражение>;
```
=
```
<тип соединения> <точка>;
assign <точка> = <выражение>;
```

Директивы, комментарии

```
‘include <имя файла>
‘define <имя макроса> <текст макроса>
‘define <имя макроса>(<аргументы>) <текст макроса>
‘undef <имя макроса>
‘ifdef <имя макроса>
‘ifndef <имя макроса>
‘elsif <имя макроса>
‘else
‘endif
// <однострочный комментарий> \n
/* <многострочный комментарий> */
```

Всё это устроено как в C/C++ с поправкой на «‘» вместо «#»

Тип `integer`

`integer`

Это тип переменных, во многом аналогичный `reg`-шинам

Основные особенности (и отличия от `reg`-шины):

- Ширина точек этого типа — неспецифицированная не менее 32
- Значения этого типа знаковые
- Переменные этого типа не отображаются ни в какие точки схемы, но значения этого типа можно использовать в выражениях
  - *Строго говоря, мнения о поддерживаемости этого типа расходятся: здесь пересказаны предложения из программного стандарта, а в стандарте синтеза сказано «`integer` поддерживается» без пояснений*

Тип integer

Пример: следующие два фрагмента кода эквивалентны в аппаратном смысле

```
reg [8:0] in1, in2, in3, o;
integer sum;
always @* begin
  sum = in1 + in2;
  if(sum > in3) o = sum;
  else o = in1;
end
```

```
reg [8:0] in1, in2, in3, o;
always @* begin
  if(in1 + in2 > in3) o = in1 + in2;
  else o = in1;
end
```

Именованные блоки и локальные объявления

Каждой составной команде и, более широко, каждому блоку кода, обрамлённому словами `begin-end`, можно присовоить *имя*, сделав этот блок именованным:

```
begin : <имя>
  ...
end
```

В начале именованного блока можно объявлять локальные переменные

Значения локальных переменных

- можно изменять и использовать внутри блока и
- нельзя изменять и использовать вовне блока
  - *Точнее, можно с получением доступа через имена экземпляров и блоков аналогично именам полей структур в C/C++ (через точку), но это неподдерживаемая возможность*

Коллизии имён локальных переменных в иерархии блоков (областей видимости) разрешаются так же, как и в C/C++

Именованные блоки и локальные объявления

Пример: следующие два фрагмента кода эквивалентны в аппаратном смысле

```
reg [8:0] in1, in2, in3, o;
integer sum;
always @* begin
  sum = in1 + in2;
  if(sum > in3) o = sum;
  else o = in1;
end
```

```
reg [8:0] in1, in2, in3, o;
always @* begin : some_name
  integer sum;
  sum = in1 + in2;
  if(sum > in3) o = sum;
  else o = in1;
end
```

Процедурные циклы

```
for(<переменная> = <выражение (начальное значение)>;
    <выражение (условие продолжения цикла)>;
    <переменная> = <выражение (следующее значение)>
   ) <команда>
```

Внутри процедур можно записывать и такие циклы со смыслом как в C/C++ и с ограниченным по сравнению с C/C++ синтаксисом

Цикл поддерживается только в том случае,
если множество значений *переменной*, перебираемых в цикле,
определено статически (на этапе компиляции; ∼ `constexpr` в C/C++)

Есть и другие виды циклов (`repeat`, `forever`, `while`), но все они не поддерживаются, и поэтому здесь подробно не обсуждаются

Процедурные циклы

Пример

Подсхема, разворачивающая задом наперёд шину x ширины 8
и направляющая результат в шину y, может быть реализована так:

```
reg [7:0] x, y;
always @(posedge clk) begin : some_name
  integer i;
  for(i = 0; i < 8; i = i + 1)
    x[i] <= y[7-i];
end
```

Функции

```
function <тип функции> <имя функции>;
  <объявления входов и точек>
  <команда>
endfunction
```

Объявления входов — это объявления точек с ключевым словом `input`

При вызове функции (как в C/C++, и использующемся в выражениях)

- аргументы — это *входы* в порядке их *объявления*, и
- возвращается значение указанного «*типа функции*»: «[<*msb*>:<*lsb*>]», «signed [<*msb*>:<*lsb*>]» или «`integer`»

За исключением сказанного про входы,
все *объявления точек* в функции считаются локальными

Имя функции используется в *команде* как имя выходной переменной,
в которую функцией возвращается значение

Команда должна быть составлена так, чтобы выполняться целиком
в одном регионе и с полным однозначным определением результата

Функции

```
function <тип функции> <имя функции>;
  <объявления входов и точек>
  <команда>
endfunction
```

Пример

Функция, возвращающая максимум двух чисел ширины 5:

```
function [4:0] my_max;
  input [4:0] x, y;
  if(x >= y) my_max = x;
  else my_max = y;
endfunction
```

Использование этой функции:

```
assign z = u + my_max(3, v);
```

В функции могут вызываться другие функции — в т.ч. рекурсивно, но поддерживается только рекурсия со «статической глубиной»

Возведение в степень и логарифмирование

```
x ** y
```

Это операция возведения `x` в степень `y`, поддерживаемая в двух случаях (а в остальных неподдерживаемая):

1. `x` — это константа 2
   - `2 ** y` — это $(00\ldots01\underbrace{00\ldots0}_{y})$
2. `x` и `y` — константы

```
$clog2(x)
```

Это округление вверх логарифма `x` по основанию 2

Эта функция довольно полезна — например, для преобразования ширины шины в наименьшую ширину числа для индексации этой шины

К сожалению, строго говоря, в $\mathcal{V}$ эта функция не поддерживается

Но всё же ввиду полезности она обычно поддерживается на практике

Поэтому будем считать эту функцию поддерживаемой

Массивы

В $\mathcal{V}$ можно объявлять (как в C/++) многомерные массивы точек с явным указанием константных границ индексации:

- Массив из 4-х проводов, индексация с ноля
  ```
  wire arr[0:3];
  ```
- Массив из 3-х `reg`-шин ширины 5, индексация с тройки
  ```
  reg [4:0] arr[3:5];
  ```
- Двумерный массив (3x3) `reg`-значений, первая индексация с ноля, вторая — с единицы
  ```
  reg arr[0:2][1:3];
  ```
- Двумерный массив `wire`-шин ширины 5 того же размера и с той же индексацией, что и в предыдущем пункте
  ```
  wire [4:0] arr[0:2][1:3];
  ```

Массивы в качестве портов модулей запрещены

Массивы

Пример: параллельный регистр ширины 8, выходная шина которого «собирается» из внутреннего двумерного массива шин ширины 2

```
module register (input clk, input [7:0] in, output [7:0] out);
  reg [1:0] arr[0:1][1:2];
  always @(posedge clk)
    {arr[0][1], arr[0][2], arr[1][1], arr[1][2]} <= in;
  assign out = {arr[0][1], arr[0][2], arr[1][1], arr[1][2]};
endmodule
```

Параметры

При разработке схемы может появиться необходимость реализовать много *похожих* схем

Например: параллельный регистр ширины 2; параллельный регистр ширины 3; параллельный регистр ширины 4; ...

Хотелось бы один раз реализовать параллельный регистр произвольной ширины с уточнением этой ширины в экземплярах

Для таких целей в $\mathcal{V}$ включены параметры модулей

Параметр во многом похож на переменную, **но** значение параметра можно задать **только**

- в объявлении этого параметра (значение по умолчанию) и
- в объявлении экземпляра модуля (значение не по умолчанию)

Значение параметра в выражениях считается константным (как `constexpr` в C/C++)

Параметры

Объявление *параметра* заданного *типа* и со значением по умолчанию, определяющимся заданным *константным выражением*, устроено так:

```
parameter <тип> <имя параметра> = <константное выражение>;
```

Тип можно опустить, и тогда подразумевается тип *выражения*

Замечание: тип целочисленных констант (0, 1, 2, ...) — `integer`

Под одним ключевым словом «`parameter`» можно объявлять и несколько параметров аналогично объявлению нескольких точек:

```
parameter P1 = 1, P2 = 2, P3 = 3;
```

Параметры

Вставка экземпляра модуля с параметрами устроена так:

```
<имя модуля> #(<назначения параметров через запятую>)
<имя экземпляра> (<назначения портов>);
```

```
<назначение параметра> ::=
    .<имя параметра>(<константное выражение>)
```

Если имя параметра не встречается в *назначениях*,
то используется значение параметра по умолчанию

Если текст «#(...)» отсутствует,
то используются значения всех параметров по умолчанию

Параметры

Пример: параллельный регистр ширины 8 (`reg8`), составленный из параллельных регистров (`register`) ширины 3 и ширины 5

```
module register(clk, d, q);
  parameter W = 3;
  input clk;
  input [W-1:0] d;
  output reg [W-1:0] q;
  always @(posedge clk) q <= d;
endmodule

module reg8(input clk, input [7:0] d, output [7:0] q);
  register          r1(.clk(clk), .d(d[2:0]), .q(q[2:0]));
  register #(.W(5)) r2(.clk(clk), .d(d[7:3]), .q(q[7:3]));
endmodule
```

Параметры

Параметры, как и порты, можно задать в «шапке» объявления модуля:

```
module <имя модуля>
#(<список параметров>) (<назначения портов>);
```

Список параметров — это
набор объявлений параметров с «,» вместо «;»

Пример: следующие два фрагмента кода эквивалентны

```
module register(clk, in, out);
  parameter W = 3;
  input clk; input [W-1:0] in; output reg [W-1:0] out;

module register #(parameter W = 3)
(input clk, input [W-1:0] in, output reg [W-1:0] out);
```

Локальные параметры

При разработке «серьёзных» схем оказываются полезными параметры, изменение значений которых при вставке экземпляра **недопустимо**

Например, это параметры,

- значения которых
  однозначно определяются значениями других параметров
- в которых разработчик модуля сохранил
  технические детали («тонкие настройки»), изменение которых может привести к неэффективной работе модуля и ошибкам

В таких случаях принято использовать локальные параметры:
в их объявлении вместо «`parameter`» пишется «`localparam`»

Для ясности параметры, объявленные при помощи слова `parameter`, будем называть внешними

Аналогия между параметрами и точками:

порты	←	все точки	→	точки, не являющиеся портами
внешние параметры	←	все параметры	→	локальные параметры

Блоки генерации

Иногда в код хочется/необходимо вставить
много однотипных экземпляров с похожими назначениями портов

Например, параллельный регистр ширины 4 можно было бы реализовать так (*но не делайте так!*):

```
module Dff(input clk, input d, output reg q);
  always @(posedge clk) q <= d;
endmodule

module register(input clk, input [3:0] d, output [3:0] q);
  Dff d0(.clk(clk), .d(d[0]), .q(q[0]));
  Dff d1(.clk(clk), .d(d[1]), .q(q[1]));
  Dff d2(.clk(clk), .d(d[2]), .q(q[2]));
  Dff d3(.clk(clk), .d(d[3]), .q(q[3]));
endmodule
```

Вставку таких похожих друг на друга экземпляров можно сделать более компактной и наглядной при помощи блоков генерации

Блоки генерации

```
genvar i;
generate
  for(i = 0; i < 4; i = i + 1)
    Dff d(.clk(clk), .d(in[i]), .q(out[i]));
endgenerate
```

`genvar`

Это тип переменных, аналогичный `integer`
и используемый в выражениях блоков генерации

```
generate
  ...
endgenerate
```

Это блок генерации, внутри которого записывается последовательность команд генерации

Слова «`generate`» и «`endgenerate`» можно опускать, но не рекомендуется, чтобы не запутаться в собственном коде

Блоки генерации

```
genvar i;
generate
  for(i = 0; i < 4; i = i + 1)
    Dff d(.clk(clk), .d(in[i]), .q(out[i]));
endgenerate
```

Основные команды генерации:

- Такое же объявление, как и в теле модуля (точек, параметров, процесса, экземпляра, ...)
  - **выполнение** команды: объявление добавляется в тело модуля
- Составная команда (`begin-end`), цикл (`for`), ветвление (`if`), выбор (`case`), как в теле процедуры, но с командами генерации вместо процедурных команд
  - **выполнение** команды такое же, как у процедурной команды

Все имена, объявленные внутри блока генерации, локальны для этого блока

Блоки генерации

```
genvar i;
generate
  for(i = 0; i < 4; i = i + 1)
    Dff d(.clk(clk), .d(in[i]), .q(out[i]));
endgenerate
```

Цикл блока генерации

```
for(<переменная> = <выражение (начальное значение)>;
    <выражение (условие продолжения цикла)>;
    <переменная> = <выражение (следующее значение)>
   ) <команда>
```

отличается от процедурного цикла только тем, что

- *переменная* должна иметь тип genvar и
- записывается *команда* генерации вместо процедурной

Для цикла блока генерации должны соблюдаться те же ограничения однозначности/константности/статичности перебора, что и для процедурного цикла

Блоки генерации

```
genvar i;
generate
  for(i = 0; i < 4; i = i + 1)
    Dff d(.clk(clk), .d(in[i]), .q(out[i]));
endgenerate
```

Выполнение цикла:

- согласно заданию цикла перебираются значения *переменной*
- на очередной итерации перебора в *команду генерации* подставляется текущее значение *переменной*, и *команда* выполняется

Блоки генерации

Отличия ветвления и выбора в блоке генерации
от процедурных ветвления и выбора:

- все переменные, используемые в условиях
  (условие ветвления, выражения выбора — главное и в случаях)
  должны иметь тип `genvar`
- на месте процедурной команды записывается команда генерации
- выполнение процедурной команды
  заменяется на выполнение команды генерации

Блоки генерации

Другой пример: в зависимости от значения параметра `revx`,

- либо в шину `y` пересылается значение шины `x`,
  а в шину `v` — зеркально отражённое значение шины `u`,
- либо в шину `v` пересылается значение шины `u`,
  а в шину `y` — зеркально отражённое значение шины `x`

```
parameter revx = 0;
wire [7:0] x, y, u, v;
genvar i;
generate
  for(i = 0; i < 8; i = i + 1)
    if(revx) begin
      assign y[i] = x[7-i];
      assign v[i] = u[i];
    end
    else begin
      assign y[i] = x[i];
      assign v[i] = u[7-i];
    end
endgenerate
```