Языки описания схем

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

Блок 19

Verilog:
немного синтаксического сахара

Лектор:
Подымов Владислав Васильевич

E-mail:
valdus@yandex.ru

ВМК МГУ, 2020, осенний семестр

$\mathcal{V}$: директивы компилятору, 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
```

// — начало однострочного комментария

/* ... */ — многострочный комментарий

Всё это работает так же, как и в *C/C++*

$\mathcal{V}$: целые числа и именованные блоки

`integer`

Это тип, относящийся к категории *типов переменных* и во многом аналогичный типу *reg*

Основные отличия от `reg`:

- Ширина по умолчанию:
  `reg` — 1, `integer` — не специфицирована, не меньше 32
- Знаковость: `reg` по умолчанию беззнаковый, `integer` — знаковый
- Аппаратная семантика:
  - переменная типа `reg` отображается в выход некоторого аппаратного элемента схемы
  - переменная типа `integer` не отображается ни в какую точку схемы[1], но значения этого типа можно использовать в выражениях

[1] Строго говоря, мнения об аппаратной поддержке `integer` расходятся: в программном стандарте говорится нечто похожее на то, что сказано здесь, а в стандарте синтеза говорится только "`integer` *поддерживается*" без пояснений

$\mathcal{V}$: целые числа и именованные блоки

Пример:

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

Этот фрагмент кода эквивалентен фрагменту

```
reg [8:0] in1, in2, in3, o;
always @* begin
  if(in1 + in2 > {1'b0, in3}) o = in1 + in2;
  else o = in1;
end
```

$\mathcal{V}$: параметры

При разработке схемы может появиться необходимость реализовать много *похожих* схем — например,

- параллельный регистр ширины 2
- параллельный регистр ширины 3
- ...
- параллельный регистр ширины `W`?

Похожие схемы можно описать в рамках одного модуля, содержащего параметры (*как* `W` *выше*)

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

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

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

$\mathcal{V}$: параметры

Объявление параметра заданного <типа> со значением по умолчанию, равным значению <константного выражения>:

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

Объявление параметра с типом *по умолчанию*:

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

Тип по умолчанию равен типу <выражения> справа от равенства

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

Под одним ключевым словом “`parameter`”
можно объявлять и несколько параметров:

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

$\mathcal{V}$: параметры

Вставка экземпляра модуля с параметрами:

```
<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

$\mathcal{V}$: 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
```

$\mathcal{V}$: параметры

Параметры, *как и порты*, можно задать в “шапке” объявления модуля:

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

<Список параметров> — это набор объявлений параметров без “;”, разделённых запятой

Пример:

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

Этот фрагмент кода эквивалентен фрагменту

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

$\mathcal{V}$: локальные параметры

При разработке “серьёзных” схем часто возникают параметры, изменение значений которых при вставке экземпляра **недопустимо**:

- Параметры, значения которых однозначно определяются значениями других параметров
- Параметры, 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	$\leftarrow$	все точки	$\rightarrow$	точки, не являющиеся портами
внешние параметры	$\leftarrow$	все параметры	$\rightarrow$	локальные параметры

$\mathcal{V}$: возведение в степень

Операция возведения `x` в степень `y`:

```
x ** y
```

Эта операция *поддерживается* только в следующих двух случаях:

1. `x` — константа 2
   - `2**y` — это значение $(00\ldots01\underbrace{00\ldots0}_{y})$
2. `x` и `y` — константные выражения
   - в частности, "`**`" можно без дополнительных ограничений использовать в объявлениях параметров

Пример:

```
parameter MAIN = 2;
localparam AUX = 2 ** MAIN;
```

Что это может означать:

- `MAIN` — ширина управляющего входа мультиплексора (по умолчанию 2)
- `AUX` — количество входов мультиплексора, однозначно определяемое значением параметра `MAIN` и недоступное для переопределения

$\mathcal{V}$: массивы

В $\mathcal{V}$ можно объявлять 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];
```

Ограничение: порты модуля не могут быть массивами

$\mathcal{V}$: 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
```

$\mathcal{V}$: именованные блоки и локальные точки

Во многих случаях составной команде и в целом тому,
что обрамляется словами “`begin-end`”, можно присвоить имя:

begin : <имя> <текст> end

Такая запись называется именованным блоком

В начале именованного блока можно объявлять локальные переменные:

- их значения можно изменять и использовать внутри блока
- их значения нельзя изменять и использовать снаружи блока[1]
- правила разрешения коллизий имён переменных
  в иерархии блоков — такие же, как и в *C/C++*

[1] Более точно, снаружи блока их тоже можно использовать и изменять, но это *неподдерживаемая* возможность языка

$\mathcal{V}$: именованные блоки и локальные точки

Пример:

```
wire [1:0] x = 2;
reg [2:0] r;
always @(posedge clk, posedge rst)
  if(rst) r <= 0;
  else begin : some_name
    integer x;
    x = 3;
    r <= r + x;
  end
```

Этот фрагмент кода эквивалентен фрагменту

```
wire [1:0] x = 2;
reg [2:0] r;
always @(posedge clk, posedge rst)
  if(rst) r <= 0;
  else r <= r + 3;
```

$\mathcal{V}$: процедурные циклы

В процедурах $\mathcal{V}$ есть *ограниченные* возможности использования *циклов*:

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

(*вроде бы семантика цикла очевидна?*)

Ограничение: множество значений <переменной>, перебираемых в цикле, должно однозначно (*константно*, *статически*) определяться текстом цикла

Пример: запись зеркально отражённого значения шины `y` в шину `x` по передним фронтам сигнала `clk`

```
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
```

$\mathcal{V}$: блоки генерации

Иногда в код $\mathcal{V}$ хочется/необходимо вставить
много однотипных экземпляров с похожими назначениями портов

Например, параллельный регистр ширины 4 можно реализовать 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

$\mathcal{V}$: 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` — особый тип переменных,
аналогичный `integer` и используемый в выражениях блоков генерации

`generate ... endgenerate` — блок генерации, внутри которого[1]
записывается последовательность команд генерации

[1] На самом деле слова “`generate`” и “`endgenerate`” можно и не писать, но рекомендуется их всегда записывать явно, чтобы не запутаться в собственном коде

$\mathcal{V}$: блоки генерации

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

Какие бывают команды генерации:

- то же объявление, что и в теле модуля (точек, параметров, процесса, экземпляра, ...)
  - **выполнение** команды: объявление добавляется в тело модуля
- составная команда:

  `begin` <последовательность команд генерации> `end`
  - **выполнение** команды: выполняются все команды между `begin` и `end`
- команда цикла, ветвления или выбора

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

$\mathcal{V}$: 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> = <начальное значение>;
  <выражение>;
  <переменная> = <следующее значение>
) <команда генерации>
```

<Переменная> должна иметь тип genvar

Для цикла в блоке генерации должны соблюдаться те же ограничения (однозначность/константность/статичность перебора), что и для процедурного цикла

$\mathcal{V}$: 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

$\mathcal{V}$: 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

$\mathcal{V}$: 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
```