Школа схемотехнического проектирования устройств обработки сигналов

Занятие 12.

Языки описания аппаратуры. Язык описания аппаратуры Verilog HDL

От редакции: В одном из последних номеров журнала мы закончили публикацию статьи С. Емца «Verilog — инструмент разработки цифровых систем» [1], в которой рассмотрены особенности использования языка описания аппаратуры Verilog применительно к моделированию и тестированию ASIC. В этом и следующих занятиях школы мы рассмотрим использование языка Verilog для описания FPGA, приведем примеры построения комбинационных и последовательностных устройств. В качестве иллюстрации построения синтезируемых описаний на Verilog см. статью В. Б. Стешенко «Примеры проектирования цифровых устройств с использованием языков описания аппаратуры», публикация которой начата в журнале «Схемотехника», № 7/2001.

Владимир Стешенко, к.т.н.

steshenk@sm.bmstu.ru

Общие сведения

Как известно [1–4], язык описания аппаратуры Verilog был разработан фирмой Gateway Design Automation в 1984 г. После поглощения последней компанией Caddence язык начал получать все более широкое распространение среди разработчиков и стал не менее популярен, чем VHDL.

В отличие от VHDL, структура и синтаксис которого напоминают такие «сложные» языки, как АДА или АЛГОЛ, синтаксис Verilog напоминает очень популярный в среде как программистов, так и разработчиков встроенных систем и систем ЦОС старый добрый C. Verilog позволяет достаточно эффективно выполнить описание и провести моделирование (simulate) и синтез цифровых схем благодаря применению встроенных примитивов (built-in primitives) и примитивов пользователя (user-defined primitives), средствам временного контроля (timing checks), моделирования задержки распространения от входа до выхода (pin-to-pin delay simulation), возможности задания внешних тестовых сигналов (external stimulus).

Как и VHDL, Verilog изначально предназначался для моделирования цифровых систем и как средство описания синтезируемых проектов стал использоваться с 1987 г. В настоящее время ведущие пакеты синтеза систем на ПЛИС, такие как продукты фирм Synopsis, Caddence, Mentor Graphics, многих производителей ПЛИС, поддерживают синтез с описания на языке Verilog.

В языке Verilog поддерживается набор типов логических вентилей (Gate Types). Для логических вентилей определены ключевые слова (Keywords): and (И), nand (И-НЕ), or (ИЛИ), nor (ИЛИ-НЕ), xor (Исключающее ИЛИ), xnor (Исключающее ИЛИ-НЕ), buf (Буферный элемент), not (Отрицание НЕ).

В Verilog при использовании вентилей необходимо задать входы и выходы элемента, а также (не обязательно) имя вентиля. Например, вентили and и or должны иметь один выход и два и более входов. Так, для вентиля И имеем:

```
and <name><list of arguments>
and myand(out, in1, in2, in3);
and (out, in1, in2
```

Вентили *buf* и *not* могут иметь один вход и один и более выходов. Имя вентиля является необязательным. Примеры использования приведены ниже:

```
buf mybuf(out1, out2, out3, in);
not (out, in);
```

Говоря о синтаксисе языка Verilog, следует помнить, что он является контекстно-зависимым языком, то есть строчные и прописные буквы различаются. Все ключевые слова задаются в нижнем регистре (строчные буквы).

Для обозначения пустого пространства (White Space) используются символы пробела, табуляции и новой строки.

В Verilog поддерживаются два типа комментариев. Они аналогичны принятым в C++ — если нужно закомментировать одну строку, то используются две косые черты в ее начале:

```
// это комментарий
```

Чтобы закомментировать несколько строк, используется следующая конструкция:

```
/* Это
комментарии... */
```

Очевидно, что комментарии не могут быть вложенными.

Операторы

В Verilog существуют три типа операторов — с одним, двумя и тремя операндами. Унарные операторы располагаются слева от операнда, бинарные — между операндами и тернарный оператор разделяет три операнда двумя операторами.

Примеры операторов:

```
clock = ~clock; // унарный оператор отрицания
                // clock — операнд
c = a || b; // || — бинарный оператор ИЛИ, a и b операнды
r = s? t: u; //?: — тернарный оператор
                // r = [если s истинно то t иначе u]
```

Как правило, при написании описаний на Verilog комментарии одной строкой (//) используются для ввода текстовых комментариев к коду, а конструкцию /* */ используют при отладке для «закомментирования» фрагментов кода.

Числа в Verilog

Целые числа (Integers)

Целые числа могут быть двоичными (binary), обозначаются **b** или **B**, десятичными (decimal, **d** или **D**), шестнадцатеричными (hexidecimal, h, H) или восьмеричными (octal, **o** или **O**). Для определения чисел используются следующие форматы:

1. <разрядность>'<основание><число> — полное описание числа;
2. <основание><число> — используется разрядность представления, заданная в системе по определению, но не менее 32 бит;
3. <число> — используется, когда основание по умолчанию десятичное.

Разрядность определяет число бит под представление числа. Например:

```
8'b10100010 // 8-битное число в двоичной системе
8'hA2 // 8-битное число в шестнадцатеричной системе
```

Неопределенное и высокоимпедансное состояния (X and Z values)

Символ x используется для задания неопределенного состояния, символ **z** показывает третье (высокоимпедансное). При использовании в качестве цифры в числах вместо символа **z** можно использовать «?». Это рекомендуется делать в операторах выбора (case expressions) для улучшения читаемости кода. Ниже приведены примеры использования символов x и z в числах.

```
4'b10x0
4'b101z
12'dz
12'd?
8'h4x
```

Отрицательные числа (Negative numbers)

Отрицательное число задается с помощью знака минус перед разрядностью числа. Примеры отрицательных чисел:

```
-8'd5
8'b-5 // неправильно, знак минус перед числом, а не разрядностью!
```

Подчеркивание (Underscore)

Знак подчеркивания (_, underscores) может быть записан в любом месте числа, что позволяет использовать его как разделитель разрядов, улучшающий читабельность.

```
16'b0001_1010_1000_1111 // использование подчеркивания
8'b_0001_1010 // некорректное использование подчеркивания
```

Действительные числа (Real)

Действительные числа (Real numbers) могут быть представлены либо в десятичном виде, либо в стандартной форме с плавающей точкой (scientific format). Ниже приведены примеры действительных чисел:

```
1.8
3_2387.3398_3047
3.8e10 // e или E для обозначения порядка
2.1e-9
3. // неправильно!
```

Строки (Strings)

Строка заключается в кавычки и не может занимать более одной линии. Примеры использования строк:

```
«hello world»; // правильное использование строк
«good
b
y
e
wo
rld»; // неправильное использование строк
```

Цепи в Verilog (Nets)

Для обозначения цепей используются следующие ключевые слова: **wire**, **supply0**, **supply1**. Величина по умолчанию (default value) — z. Разрядность по умолчанию (default size) — 1 бит.

По своему назначению цепь (Net) в Verilog сходна с сигналом в VHDL. Цепи обеспечивают непрерывное модифицирование сигналов на выходах цифровой схемы относительно изменения сигналов на ее входах.

Если драйвер (источник) сигнала цепи имеет некоторое значение, то и цепь принимает то же значение. Если драйверы цепи принимают различные значения, цепь принимает значение наиболее «сильного» сигнала (**strongest**), если же «сила» каждого сигнала равнозначна, то цепь принимает неопределенное состояние (x).

Для обозначения цепи наиболее часто применяется ключевое слово wire, ключевые слова **supply0** и **supply1** используются для моделирования источников питания (power supplies) в схеме.

Регистры (Registers)

Для обозначения регистров применяется ключевое слово reg. Величина по умолчанию — x. Разрядность по умолчанию — 1 бит.

Основное различие между цепями (nets) и регистрами (registers) состоит в том, что значение регистра должно быть назначено явно. Эта величина сохраняется до тех пор, пока не сделано новое назначение. Рассмотрим использование этого свойства на примере триггера с разрешением (E-type flip flop):

```
module E_ff(q, data, enable, reset, clock);
output q;
input data, enable, reset, clock;
reg q;

always @(posedge clock)
if (reset == 0)
q = 1'b0;
else if (enable==1)
q = data;
endmodule
```

Регистр **q** хранит записанную в него величину до тех пор, пока не произойдет новое назначение сигнала.

Для того чтобы провести моделирование этого примера, напишем модуль верхнего уровня (higher level module), который формирует тестовый сигнал и позволяет провести наблюдение за выходами триггера. В этом случае сигнал q является цепью, драйвером которой служит модуль E_ff.

```
module stimulus;
reg data, enable, clock, reset;
wire q;

initial begin
clock = 1'b0;
forever #5 clock = ~clock;
end

E_ff eff0(q, data, enable, reset, clock);

initial begin
reset = 1'b0;
#10 reset = 1'b1;
data = 1'b1;
#20 enable = 1;
#10 data = 1'b0;
#10 data = 1'b1;
#10 enable = 0;
#10 data = 1'b0;
#10 data = 1'b1;
#10 enable = 1;
#10 reset = 1'b0;
#30 $finish;
end

initial
$monitor($time, « q = %d», q);

endmodule
```

Векторы (Vectors)

Как цепи, так и регистры могут иметь произвольную разрядность, если объявлять их как векторы. Ниже приведены примеры объявлений векторов:

```
reg [3:0] output; // выход 4-разрядного регистра
wire [31:0] data; // 32-битная цепь
reg [7:0] a;

data[3:0] = output; // частичное назначение
output = 4'b0101; // Назначение на целый регистр
```

Очень важным является порядок назначения элементов в векторе. Первый элемент регистра является наиболее значимым (most significant):

```
reg [3:0] a; // a3 — старший бит
reg [0:3] b; // b0 — старший бит.
```

Массивы (Arrays)

Регистры (Registers), целые числа (integers) и временные (time) типы данных можно объявлять как массивы, как это показано в нижеследующем примере:

```
Объявление
<data_type_spec> {size} <variable_name> {array_size}

Использование
<variable_name> {array_reference} {bit_reference}

reg data [7:0]; // 8 1-разрядных элементов

integer [3:0] out [31:0]; // 32 4-битных элемента

data[5]; // 5 8-битных элементов
```

Регистровые файлы (Memories)

Регистровый файл представляет собой массив регистров (array of registers). Ниже представлен синтаксис объявления регистрового файла.

```
reg [15:0] mem16_1024 [1023:0]; // регистровый файл 1K X 16

mem16_1024[489]; // 489 элемент файла mem16_1024
```

Для обозначения регистровых файлов желательно использовать информативные имена, например **mem16_1024**, чтобы избежать путаницы.

Элементы с третьим состоянием (Tri-state)

Как уже упоминалось, ситуация, когда драйверов одной цепи больше одного, в языке Verilog разрешается в пользу источника, имеющего большую «силу» (signal «strengths»). Наименьшую «силу» имеет сигнал z, обозначающий третье или высокоимпедансное (high-impedance) состояние. Правда, эти рассуждения актуальны именно на этапе моделирования, но никак не синтеза ПЛИС, о них следует помнить при разработке тестов. Таким образом, драйвер в третьем состоянии не влияет на итоговое значение сигнала цепи. Пример драйвера с третьим состоянием приведен ниже.

```
module triDriver(bus, drive, value);
inout [3:0] bus;
input drive;
input [3:0] value;

assign #2 bus = (drive == 1)? value: 4'bz;

endmodule // triDriver
```

В данном примере, когда управляющий сигнал принимает высокий уровень, шина принимает значение входной величины, в противном случае она переходит в третье состояние.

Арифметические операторы (Arithmetic operators)

Бинарные операторы умножения, деления, сложения, вычитания, определения остатка от деления представлены в следующем примере.

```
module arithTest;
reg [3:0] a, b;

initial begin
a = 4'b1100; // 12
b = 4'b0011; // 3

$displayb(a * b); // умножение — 4'b1000
// 4 МСБ
$display(a / b); // деление 4
$display(a + b); // сложение 15
$display(a — b); // вычитание 9
$display((a + 1'b1) % b); // остаток 1

end

endmodule // arithTest
```

Унарные плюс и минус имеют более высокий приоритет (precedance), чем бинарные операторы.

Следует заметить, что если хотя бы один бит в одном из операндов неопределен (равен **x**), то и результат операции также будет неопределен.

Логические операторы (Logical Operators)

К логическим операциям относятся И (and), ИЛИ (or) и НЕ (not). Результат логической операции может принимать значения истинно, то есть true (1), или ложно, то есть false (0), а также иметь неопределенное состояние — unknown (x). При выполнении логического оператора все неопределенные величины, как и операнды в третьем состоянии, принимаются как имеющие низкий логический уровень (false). В качестве операнда может выступать как переменная (variable), так и логическое выражение (expression). Пример работы логических операторов приведен ниже.

```
module logicalTest;

reg [3:0] a, b, c;

initial begin
a = 2; b = 0; c = 4'hx;

$display(a && b); // оператор И результат 0
$display(a || b); // оператор ИЛИ результат 1
$display(!a); // оператор НЕ результат 0
$display(a || c); // 1, unknown || 1 (=1)
$display(!c); // unknown
end

endmodule // logicalTest
```

Операторы отношения (Relational Operators)

К операторам отношения относятся операторы «больше», «меньше», «больше или равно», «меньше или равно». Результат операции — истина или ложь. Если хотя бы один операнд неопределен, то и результат операции будет неопределен.

```
module relatTest;
reg [3:0] a, b,c, d;

initial begin
a=2;
b=5;
c=2;
d=4'hx;

$display(a < b); // true, 1
$display(a > b); false, 0
$display(a >= c); // true, 1
$display(d <= a); unknown
end
endmodule // relatTest
```

Операторы эквивалентности (Equality)

К операторам эквивалентности (equality operators) относятся операторы логического равенства (logical equality), логического неравенства (logical inequality), выборочного равенства (case equality) и неравенства (inequality). Эти операторы сравнивают операнды побитно. Логические операторы возвращают неопределенный результат, если операнд содержит неопределенные биты, в отличие от выборочных операторов. В случае неравной длины операндов более короткий операнд дополняется нулями.

Ниже приведен пример использования операторов эквивалентности.

```
module equTest;
reg [3:0] a, b,c, d, e, f;

initial begin
a = 4; b = 7;
c = 4'b010;
d = 4'bx10;
e = 4'bx101;
f = 4'bxx01;

$displayb(c); // результат 0010
$displayb(d); // результат xx10

$display(a == b); // logical equality, результат 0
$display(c!= d); // logical inequality, результат x
$display(c!= f); // logical inequality, результат 1
$display(d === e); // case equality, результат 0
$display(c!== d); // case inequality, результат 1
end
endmodule // equTest
```

Операторы эквивалентности и присваивания — это совершенно различные операторы.

Поразрядные операторы (Bitwise Operators)

К поразрядным операторам относятся поразрядное отрицание, поразрядные логические И, ИЛИ, исключающее ИЛИ, исключающее ИЛИ-НЕ. Поразрядные операторы выполняются только над операндами, имеющими одинаковую разрядность. В том случае, если разрядность одного операнда меньше другого, недостающие разряды дополняются нулями. Ниже приведен пример использования поразрядных операторов.

```
module bitTest;
reg [3:0] a, b,c;

initial begin
a = 4'b1100; b = 4'b0011; c = 4'b0101;
$displayb(~a); // поразрядное отрицание, результат 4'b0011
$displayb(a & c); // поразрядное И, результат 4'b0100
$displayb(a | b); // поразрядное ИЛИ, результат 4'b1111
$displayb(b ^ c); // поразрядное xor, результат 4'b0110
$displayb(a ~^ c); // поразрядное xnor, результат 4'b0110
end
endmodule // bitTest
```

Операторы приведения (Reduction Operator)

К операторам приведения относятся И, ИЛИ, И-НЕ, ИЛИ-НЕ, исключающее ИЛИ, исключающее ИЛИ-НЕ (два варианта). Они выполняются над многоразрядным операндом пошагово, бит за битом, начиная с двух крайних левых разрядов, выдавая на выходе одноразрядный результат. Очевидно, что такой подход позволяет реализовать проверку на четность (нечетность). Ниже приведены примеры использования операторов приведения.

```
module reductTest;
reg [3:0] a, b,c;

initial begin
```

```
a = 4'b1111;
b = 4'b0101;
c = 4'b0011;

$displayb(& a);//, (то же 1&1&1&1), равен 1 1
$displayb(| b); // (то же 0|1|0|1), равен 1
$displayb(^ b); // искл.ИЛИ (то же 0^1^0^1), результат 0
end

endmodule // reductTest
```

Безусловно, следует улавливать различия между логическими операторами, поразрядными операторами и операторами приведения. Несмотря на схожесть символов этих операторов, число операндов в каждом случае различно.

Операторы сдвига (Shift Operator)

Операторы сдвига позволяют осуществить сдвиг операнда как вправо, так и влево. Ниже приведен пример их использования:

```
module shiftTest;
reg [3:0] a;

initial begin
a = 4'b1010;

$displayb(a << 1); // сдвиг влево на 1, результат 4'b0100
$displayb(a >> 2); // сдвиг вправо на 1, результат 4'b0010
end

endmodule // shiftTest
```

Этот оператор часто применяют для реализации регистров сдвига, длинных алгоритмов перемножения и т. п.

Конкатенация (объединение, Concatenation)

Объединение позволяет увеличить разрядность (size) цепей (nets), регистров (registers) и т. д.

```
module concatTest;
reg a;
reg [1:0] b;
reg [5:0] c;

initial begin
a = 1'b1;
b = 2'b00;
c = 6'b101001;

$displayb({a, b}); // результат 3 разрядное число 3'b100
$displayb({c[5:3], a}); // результат 4 разрядное число 4'b1011

end

endmodule // concatTest
```

Повторение (Replication)

Повторение (Replication) может быть использовано для многократного повторения объединения (concatenation), как показано в нижеследующем примере.

```
module replicTest;
reg a;
reg [1:0] b;
reg [5:0] c;

initial begin
a = 1'b1;
b = 2'b00;

$displayb({4{a}}); // результат — 1111
c = {4{a}};
$displayb(c); // результат — 001111
end

endmodule // replicTest
```

Системные директивы (System Tasks)

Наверное, этот раздел не будет интересен тем, кто пишет только синтезируемые описания на Verilog. Однако поскольку язык является не только средством описания проекта, но и довольно мощным инструментом для поведенческого моделирования систем, то следует сказать несколько слов о встроенных директивах компилятора, позволяющих выполнить моделирование и провести анализ его результатов.

Директивы вывода результатов моделирования (Writing to Standard Output)

Наиболее часто применяется директива **$display**. Она может быть использована для вывода на экран строк, выражений или переменных. Ниже приведен пример использования директивы **$display**.

```
$display(«Hello world»);
--- результат Hello world

$display($time) // текущее время моделирования.

counter = 4'b10;
$display(« The count is %b», counter);
--- результат The count is 0010
```

Синтакс определения формата вывода фактически аналогичен синтаксису **printf** в языке программирования C. Ниже приведено его описание для директивы **$display**.

Таблица 1

Формат	Описание
%d or %D	Decimal
%b or %B	Binary
%h or %H	Hexadecimal
%o or %O	Octal
%m or %M	Hierarchical name
%t or %T	Time format
%e or %E	Real in scientific format
%f or %F	Real in decimal formal
%g or %G	Real in shorter of above two

Для специальных символов используются следующие эскейп-последовательности (escape sequence), приведенные в табл. 2.

Таблица 2

\n	Новая строка
\t	Табуляция
\\	\
\"	"
%%	%

Директива **$write** идентична директиве **$display**, за исключением того, что она не осуществляет автоматический переход на новую строку в конце вывода информации

Если спецификация вывода не определена, то по умолчанию используются форматы, приведенные в табл. 3.

Таблица 3

Директива	Формат по умолчанию
$display	decimal
$displayb	binary
$displayh	hexadecimal
$displayo	octal
$write	decimal
$writeb	binary
$writeh	hexadecimal
$writeo	octal

Контроль процесса моделирования (Monitoring a Simulation)

Формат директивы $monitor практически аналогичен формату **$display**. Разница заключается в том, что выход формируется при любом изменении переменных, которое произойдет в определенное время. Наблюдение может быть включено или отключено с помощью директив **$monitoron** или **$monitoroff** соответственно. По умолчанию в начале моделирования наблюдение за его ходом включено.

Окончание моделирования (Ending a simulation)

Директива **$finish** завершает симуляцию и передает управление системе моделирования. Директива **$stop** приостанавливает моделирование и переводит систему с Verilog в интерактивный режим.

Проектирование комбинационных схем

Рассмотрим проектирование комбинационных логических устройств на примере мультиплексора 4 в 1. При проектировании мультиплексора рассмотрим различные методы его построения.

Реализация на уровне логических вентилей (Gate Level Implementation)

Рассмотрим пример реализации нашего мультиплексора 4 в 1 на уровне логических вентилей.

```
module multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);

output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
wire notcntlr1, notcntrl2, w, x, y, z;

not (notcntrl1, cntrl1);
not (notcntrl2, cntrl2);

and (w, in1, notcntrl1, notcntrl2);
and (x, in2, notcntrl1, cntrl2);
and (y, in3, cntrl1, notcntrl2);
and (z, in4, cntrl1, cntrl2);

or (out, w, x, y, z);

endmodule
```

Рассмотрим этот пример построчно.

```
module multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);
```

Первая строка представляет собой описание модуля. Ключевое слово **module** используется вместе с именем модуля, по которому

осуществляется ссылка на модуль. В скобках приведен список портов модуля (port list), причем вначале перечисляются выходы, затем входы. Каждая строка завершается точкой с запятой — это, как известно, характерно для многих языков высокого уровня.

```
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
```

Все порты в списке должны быть объявлены как входы (**input**), выходы (**output**) или двунаправленые выводы (**inout**). В этом случае они по умолчанию назначаются типом цепь (**wire**), если нет других указаний. Когда назначено имя цепи, система моделирования на базе Verilog ожидает неявное назначение выходного сигнала, оценивая его, чтобы осуществлять передачу этого сигнала к внешним модулям.

```
wire notcntrl1, notcntrl2, w, x, y, z;
```

В данной строке определяются внутренние цепи, осуществляющие объединение узлов мультиплексора. Как видим, понятия цепи в Verilog и сигнала в VHDL очень схожи:

```
not (notcntrl1, cntrl1);
not (notcntrl2, cntrl2);
```

В этих строчках описываются вентили НЕ с входами **cntrl1** и **cntrl2** и выходами **notcntrl1** и **notcntrl2** соответственно. Следует помнить, что в описании портов вентиля всегда вначале идут выходы, затем входы:

```
and (w, in1, notcntrl1, notcntrl2);
and (x, in2, notcntrl1, cntrl2);
and (y, in3, cntrl1, notcntrl2);
and (z, in4, cntrl1, cntrl2);

or (out, w, x, y, z);
```

Аналогично описываются и вентили И и ИЛИ

```
endmodule
```

Конец модуля завершается ключевым словом **endmodule**.

Реализация мультиплексора с помощью логических операторов (Logic Statement Implementation)

Реализация мультиплексора с использованием логических операторов имеет следующий вид:

```
module multiplexor4_1 (out, in1, in2, in3,in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;

assign out = (in1 & ~cntrl1 & ~cntrl2) |
(in2 & ~cntrl1 & cntrl2) |
(in3 & cntrl1 & ~cntrl2) |
(in4 & cntrl1 & cntrl2);
endmodule
```

В начале следуют имя проекта и описание портов.

```
module multiplexor4_1 (out, in1, in2, in3,in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
```

Обратим внимание, что ни порты, ни их количество не изменились — стыковка с внешними модулями также останется неизменной. В какой-то мере это напоминает проект с использованием VHDL, когда для одного интерфейса можно описать несколько архитектурных тел.

```
assign out = (in1 & ~cntrl1 & ~cntrl2) |
(in2 & ~cntrl1 & cntrl2) |
(in3 & cntrl1 & ~cntrl2) |
(in4 & cntrl1 & cntrl2);
endmodule
```

С помощью конструкции **assign** осуществляется непрерывное назначение (continuous assignment) цепи out. В этом случае ее значение заново вычисляется каждый раз, когда меняется хотя бы один из операндов.

Реализация с помощью оператора выбора (Case Statement Implementation)

Рассмотрим использование оператора выбора (case statement) для реализации мультиплексора. Следует заметить, что этот способ наиболее прост и наименее трудоемок.

```
module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
reg out;

always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
case ({cntrl1, cntrl2})
2'b00: out = in1;
2'b01: out = in2;
2'b10: out = in3;
2'b11: out = in4;
default: $display(«Please check control bits»);
endcase
endmodule
```

Первые три строки знакомы — добавить к сказанному ранее практически нечего.

```
module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
reg out;
```

Единственное отличие — выход **out** определен как регистр (**register**). Это сделано для того, чтобы назначать его значения явно и не управлять ими. Такое назначение сигнала называется процедурным назначением (procedural assignment). Данные типа цепь (wire) не могут быть назначены явно, они нуждаются в сигнале-драйвере. Такое назначение называется непрерывным назначением (continuous assignment).

```
always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
```

Эта конструкция читается так же, как пишется, то есть значение вычисляется всегда при изменении хотя бы одного операнда — система постоянно их отслеживает. Несколько забегая вперед, следует отметить, что данная конструкция является синтезируемой во многих системах проектирования, в частности в Max + Plus фирмы Altera. Список переменных называется списком чувствительности (sensitivity list), поскольку данная конструкция чувствительна к их изменениям. Данный термин уже известен нам из описания языка VHDL.

Ключевое слово always @(expr or expr...);

```
case ({cntrl2, cntrl1})
2'b00: out = in1;
2'b01: out = in2;
2'b10: out = in3;
2'b11: out = in4;
default: $display(«Please check control bits»);
endcase
endmodule
```

Синтаксис оператора выбора case в Verilog сходен с синтаксисом оператора выбора в языке C. Условием является контатенация или объединение переменных cntr2 и cntr1 в двухразрядное число. Завершает оператор выбора ключевое слово **endcase**.

Здесь нелишне будет напомнить, что следует различать процедурное и непрерывное назначение сигналов, что и иллюстрируют вышеприведенные примеры.

Реализация с использованием условного оператора (Conditional Operator Implementation)

Ничего нового мы не увидим — отчетливо видно, что в нашем случае условный оператор проигрывает оператору case в наглядности представления (хотя для большинства систем проектирования синтезируемые реализации окажутся идентичными):

```
module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;

assign out = cntrl1? (cntrl2? in4: in3): (cntrl2? in2: in1);

endmodule
```

Тестовый модуль (The Stimulus Module)

Вообще говоря, первоначально язык Verilog задумывался как язык верификации цифровых устройств (Verify Logic). Поэтому одной из мощных возможностей языка является наличие средств для задания тестовых сигналов. Ниже приводится пример такого модуля для нашего мультиплексора 4 в 1.

```
module muxstimulus;
reg IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;
wire OUT;

multiplexor4_1 mux1_4(OUT, IN1, IN2, IN3, IN4, CNTRL1, CNTRL2);

initial begin
IN1 = 1; IN2 = 0; IN3 = 1; IN4 = 0;
$display(«Initial arbitrary values»);
#0 $display(«input1 = %b, input2 = %b, input3 = %b, input4 = %b\n»,
IN1, IN2, IN3, IN4);

{CNTRL1, CNTRL2} = 2'b00;
#1 $display(«cntrl1=%b, cntrl2=%b, output is %b», CNTRL1, CNTRL2,
OUT);

{CNTRL1, CNTRL2} = 2'b01;
#1 $display(«cntrl1=%b, cntrl2=%b output is %b», CNTRL1, CNTRL2,
OUT);

{CNTRL1, CNTRL2} = 2'b10;
#1 $display(«cntrl1=%b, cntrl2=%b output is %b», CNTRL1, CNTRL2,
OUT);

{CNTRL1, CNTRL2} = 2'b11;
#1 $display(«cntrl1=%b, cntrl2=%b output is %b», CNTRL1, CNTRL2,
OUT);

end

endmodule
```

Рассмотрим данный пример подробнее.

```
module muxstimulus;
```

Поскольку наш генератор тестов является модулем верхнего уровня иерархии (top level module), то в его описании отсутствует список портов. Для задания модуля используется ключевое слово **module**.

```
reg IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;
wire OUT;
```

В данном случае мы обеспечиваем подачу тестовых сигналов на входы нашего мультиплексора и снятие сигнала с его выхода. Следовательно, мы должны назначить сигналы для входов мультиплексора и сигналу, который управляется (driven) его выходом. Поэтому для входов использованы данные типа **reg**, а для выхода — **wire**.

```
multiplexor4_1 mux1_4(OUT, IN1, IN2, IN3, IN4, CNTRL1, CNTRL2);
```

В этой строке происходит обращение к тестируемому модулю **multiplexor4_1**, в таких случаях принят следующий синтаксис:

```
<module_name> <instance_name> (port list);
```

Имя экземпляра (**instance name**) очень похоже на понятие языка VHDL и является необходимым при обращении к определяемым пользователем модулям (user defined modules), чтобы не нарушать иерархии проекта.

Список портов (**port list**) устанавливает соответствие между сигналами тестового и тестируемого модулей, при этом порядок переменных в списке портов должен соответствовать порядку их объявления в тестовом модуле.

```
initial begin
IN1 = 1; IN2 = 0; IN3 = 1; IN4 = 0;
$display(«Initial arbitrary values»);
#0 $display(«input1 = %b, input2 = %b, input3 = %b, input4 = %b\n»,
IN1, IN2, IN3, IN4);
```

Собственно моделирование (simulation) осуществляется конструкцией, определяемой ключевыми словами initial begin... end. Она предназначена для объединения инструкций, которые могут выполняться одновременно.

Перед инициализацией процесса моделирования сообщение об этом выводится с помощью **$display**. Конструкция **#0** перед директивой вывода означает, что вывод на экран осуществляется после назначения сигналов на входы. Синтаксис директивы **$display** подобен синтаксису функции **printf** в языке C.

```
$display( expr1, expr2,...., exprN);
```

ExprN может быть переменной, выражением или строкой.

```
{CNTRL1, CNTRL2} = 2'b00;
#1 $display(«cntrl1=%b, cntrl2=%b, output is %b», CNTRL1, CNTRL2,
OUT);
```

В следующем блоке осуществляется назначение сигналов управления:

```
CNTRL1 = 0;
CNTRL2 = 0;
```

Оператор конкатенации (concatenation operator) { } может быть использован для группового назначения. Следует помнить, что количество разрядов в числе и переменной должно совпадать.

```
{CNTRL1, CNTRL2} = 2'b01;
#1 $display(«cntrl1=%b, cntrl2=%b output is %b», CNTRL1, CNTRL2,
OUT);


{CNTRL1, CNTRL2} = 2'b10;
#1 $display(«cntrl1=%b, cntrl2=%b output is %b», CNTRL1, CNTRL2,
OUT);


{CNTRL1, CNTRL2} = 2'b11;
#1 $display(«cntrl1=%b, cntrl2=%b output is %b», CNTRL1, CNTRL2,
OUT);


end
endmodule
```

В данном фрагменте сигналы управления изменяются, и изменение выхода отслеживается директивой $display, которая выполняется с некоторой задержкой относительно сигналов управления, что позволяет отобразить корректные значения.

В следующем занятии мы завершим рассмотрение языков описания аппаратуры высокого уровня.

Литература

1. Емец С. Verilog — инструмент разработки цифровых систем… — Схемотехника № 1–4, 2001.
2. IEEE Std 1364-95, Verilog LRM. 1995. The Institute of Electrical and Electronics Engineers. Available from The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017 USA.
3. Palnitkar, S. 1996. Verilog HDL: A Guide to Digital Design and Synthesis. Upper Saddle River, NJ: Prentice-Hall, 396 p. ISBN 0-13-451675-3.
4. Thomas, D. E., and P. Moorby. 1991. The Verilog Hardware Description Language. 1st ed. Dordrecht, Netherlands: Kluwer, 223 p. ISBN 0-7923-9126-8, TK7885.7.T48 (1st ed.). ISBN 0-7923-9523-9 (2nd ed.).