Школа схемотехнического проектирования устройств обработки сигналов

Занятие 13.
Языки описания аппаратуры. Язык описания аппаратуры Verilog HDL

Владимир Стешенко, к.т.н.

steshenk@sm.bmstu.ru

Модули проекта (Design Blocks Modules)

Рассмотрим правила построения иерархического проекта. В качестве примера мы возьмем триггер с разрешением (E-type flip flop). В Verilog возможны два подхода к его реализации:

- сверху вниз (Topdown) — начать с описания E-триггера и далее добавлять детали проекта;
- снизу вверх (Bottom up) — начать с базовых блоков, а затем объединить их в единый проект.

Мы воспользуемся технологией «снизу вверх», хотя, наверное, наиболее удобно использование обоих методов.

Начнем с описания D-триггера как модуля. Ключевые слова для задания модуля

module <module_name>... endmodule, as below.

Ниже приведено его описание.

```
module dff(q, data, clock);
output q;
input data, clock;
reg q;
always @(posedge clock)
q = data;
endmodule // dff
```

Поведенческое описание мультплексора с инверсией (inverting multiplexor) имеет вид:

```
module mux2_1(out, in1, in2, cntrl);
output out;
input in1, in2, cntrl;
assign out = cntrl? ~in1: ~in2;
endmodule // mux2_1
```

Тогда описание модуля верхнего уровня для E-триггера, состоящего из D-триггера и мультиплексора, принимает следующий вид:

```
module e_ff(q, data, enable, reset, clock);
output q;
input data, enable, reset, clock;
wire norout, muxout;
mux2_1 mod1 (muxout, data, q, enable);
nor (norout, reset, muxout);
dff dff0 (q, norout, clock);
endmodule
```

Обратим еще раз внимание на обращение к модулю, определяемому пользователем, через имя экземпляра модуля:

```
name_of_module instance_name (port_list);
```

Тестирование

Ниже приведен текст модуля для тестирования нашего триггера. Как и в примере с мультиплексором, он не имеет списка портов, поскольку является самым верхним в иерархии.

```
module e_ffStimulus;
reg data, enable, reset, clock;
wire q;
e_ff mod1 (q, data, enable, reset, clock);
initial begin
clock = 1'b0;
forever clock = #5 ~clock;
end
initial begin
enable = 1'b0; // переменная разрешения
reset = 1'b1; // активный уровень сброса — 1
#20 reset = 1'b0; // сброс.
data = 1'b1; установка в HIGH
#10 enable = 1'b1; // and then enable data latching
#10 data = 1'b1; // изменение данных
#20 data = 1'b0; // изменение данных
#30 data = 1'b1; // изменение данных
#10 data = 1'b0; // изменение данных
#10 data = 1'b1; // изменение данных
#20 enable = 1'b0; // запрет защелкивания
#10 data = 1'b0; // изменение данных
#10 reset = 1'b1; // сброс снова
#20 $finish; // все
end // такт остановлен
initial begin
$display($time, « reset, enable, data, q «);
$display($time, « %d %d %d %d»,
reset, enable, data, q);
forever #10 $display($time, « %d %d %d %d»,
reset, enable, data, q);
end
endmodule // e_ffStimulus
```

Порты (Ports)

Порты обеспечивают связь модуля с другими модулями проекта. Ниже приведены примеры портов для нашего случая.

```
module d_ff(q, d, reset, clock);
module e_ff(q, d, enable, reset, clock);
module Stimulus;
```

Каждый порт может быть объявлен как вход (input), выход (output) или двунаправленный (inout). Всем объявленным портам по умолчанию назначаются цепи (wire). Для изменения типа сигнала необходимо выполнить назначение отдельно. Например:

```
module d_ff(q, d, reset, clock);
output q; // обязательное объявление портов
```

```
input d, reset, clock; // как входы и выходы
reg q; // снова объявляем как регистр
```

Не забывать, что выходы модуля идут первыми в списке портов.

Правила соединения (Connection Rules)

Выше мы рассматривали два типа модулей — внешние и внутренние (outer and inner). В нашем примере внешним модулем был E-триггер, внутренним — триггер типа D. Ниже рассмотрим основные правила соединения модулей.

Входы (Inputs)

Входы внутреннего модуля всегда должны иметь тип цепь, поскольку они управляются внешними сигналами. Входы внешнего модуля могут иметь как тип цепь, так и регистр.

Выходы (Outputs)

Во внутреннем модуле выходы могут иметь как тип цепь, так и регистр, в то время как выходы внешнего модуля должны быть типа цепь, поскольку управляются внутренними модулями.

Двунаправленые выводы (Inouts)

Двунаправленный порт всегда имеет тип цепь.

Соответствие портов (Port Matching)

Очевидно, что при обращении к модулю необходимо совпадение разрядности портов.

Присоединение портов (Connecting Ports)

Присоединение портов осуществляется либо по порядку (ordered list), либо по имени порта (by name). Например:

```
module d_ff( q, d, reset, clock);
...
endmodule
module e_ff(q, d, enable, reset, clock);
output q;
input d, enable, reset, clock;
wire inD;
...
d_ff dff0(q, inD, reset, clock);
...
endmodule
```

Базовые блоки (Basic Blocks)

В Verilog приняты два типа назначений — непрерывное и процедурное. Непрерывное назначение может быть выполнено для цепей nets или для их объединений (concatenation of nets). Операнды могут иметь произвольный тип данных.

Процедурное назначение может быть выполнено для данных типов reg, integer, real или time.

Инициализация (Initial Block)

Ключевое слово: initial

Блок инициализации состоит из группы операторов, заключенных в операторные скобки begin… end и которые будут выполняться с момента старта моделирования. Ниже приведен пример инициализации моделирования.

```
initial
clock = 1'b0;
initial
begin
alpha = 0;
#10 alpha = 1; // генерация сигнала
#20 alpha = 0;
#5 alpha = 1;
#7 alpha = 0;
#10 alpha = 1;
#20 alpha = 0;
end;
```

Конструкция always (Always Block)

Ключевое слово: always

Блок always означает, что действие выполняется непрерывно, пока процесс моделирования не будет остановлен директивой $finish или $stop.

Директива $finish на самом деле определяет симуляцию, в то время как директива $stop — паузы. Ниже приведен пример формирования тактового импульса с использованием конструкции always. Заметим, что, вообще говоря, для этих целей удобнее использовать циклы.

```
module pulse;
reg clock;
initial clock = 1'b0; // запуск такта
always #10 clock = ~clock; // такт
initial #5000 $finish // конец моделирования
endmodule
```

Пример проектирования последовательностного устройства: двоичный счетчик

Рассмотрим пример проектирования двоичного счетчика, который считает от 0 до 12, по достижении 12 сбрасывается в 0 и начинает счет заново.

Рассмотрим описание на вентильном уровне.

```
// 4-bit binary counter
module counter4_bit(q, d, increment, load_data, global_reset, clock);
output [3:0] q;
input [3:0] d;
input load_data, global_reset, clock, increment;
wire t1, t2, t3; // internal wires
wire see12, reset; // internal wires
et_ff etff0(q[0], d[0], increment, load_data, reset, clock);
et_ff etff1(q[1], d[1], t1, load_data, reset, clock);
et_ff etff2(q[2], d[2], t2, load_data, reset, clock);
et_ff etff3(q[3], d[3], t3, load_data, reset, clock);
and a1(t1, increment, q[0]);
and a2(t2, t1, q[1]);
and a3(t3, t2, q[2]);
recog12 r1(see12, q); // счет
or or1(reset, see12, global_reset); // сброс
endmodule // counter4_bit
module et_ff(q, data, toggle, load_data, reset, clock);
output q;
input data, toggle, load_data, reset, clock;
wire m1, m2;
mux mux0(m1, ~q, q, toggle);
mux mux1(m2, data, m1, load_data);
dff dff0(q, m2, reset, clock);
endmodule // et_ff
module mux(out, in1, in2, cntrl);
output out;
input in1, in2, cntrl;
assign out = cntrl? in1: in2;
endmodule // mux
module dff(q, data, reset, clock);
output q;
input data, reset, clock;
reg q;
always @(posedge clock) // at every clock edge, if reset is 1, q is
if (reset == 1)
q = 0;
else q = data;
endmodule
module recog12(flag, in);
input [3:0] in;
output flag;
assign flag = (in == 4'b1100)? 1: 0;
endmodule // recog12
module stumulus;
wire [3:0] q;
reg [3:0] d;
reg load_data, global_reset, clk, increment;
counter4_bit mod1 (q, d, increment, load_data, global_reset, clk);
initial begin
global_reset = 0;
clk = 0;
increment = 0;
load_data = 0;
d = 4'b0100;
#10 global_reset = 1;
#20 global_reset = 0;
#20 load_data = 1;
#20 load_data = 0;
#20 increment = 1;
#200 global_reset = 1;
#20 global_reset = 0;
#50 load_data = 1;
#20 load_data = 0;
#10 increment = 0;
#20 $finish;
end // initial begin
always #5 clk = ~clk;
always #10 $display ($time,» %b %b %b %d -> %b %d»,
increment, load_data, global_reset, d, q, q);
endmodule // stumulus
```

Проанализируем этот пример более подробно.

```
module counter4_bit(q, d, increment, load_data, global_reset, clock);
output [3:0] q;
input [3:0] d;
input load_data, global_reset, clock, increment;
wire t1, t2, t3; // internal wires
wire see12, reset; // internal wires
```

Как нам уже известно, первые строки модуля содержат его имя и объявление портов. Выход счетчика q, все остальные входные сигналы — t1, t2, t3, see12 и reset объявлены как внутренние цепи, подключенные ко входам, соответствующим субмодулям проекта.

```
et_ff etff0(q[0], d[0], increment, load_data, reset, clock);
et_ff etff1(q[1], d[1], t1, load_data, reset, clock);
et_ff etff2(q[2], d[2], t2, load_data, reset, clock);
et_ff etff3(q[3], d[3], t3, load_data, reset, clock);
```

В этом фрагменте кода мы определяем четыре счетных триггера с разрешением (Enable/Toggle flip flops):

```
and a1(t1, increment, q[0]);
and a2(t2, t1, q[1]);
and a3(t3, t2, q[2]);
```

Сигнал увеличения счетчика, (increment signal) объединяется по И с выходом последнего триггера для переноса сигнала счета на следующий триггер:

```
recog12 r1(see12, q);
or or1(reset, see12, global_reset);
```

Каждый раз, когда выход q изменяется, переменная see12 модифицируется по достижении 12 и внутренний сигнал сброса обеспечит

сброс счетчика на очередном такте. Далее рассмотрим поведенческую модель нашего счетчика.

Поведенческая модель счетчика (Behavioural Model)

Предшествующее описание счетчика было осуществлено на уровне вентилей и триггеров, то есть на структурном уровне. С другой стороны, при наличии достаточно мощных средств синтеза поведенческое описание позволяет получить более наглядное описание проекта. Ниже приводится пример поведенческого описания (behavioural description) счетчика.

```
// 4-bit binary up-counter — of period 13
module counter4_bit(q, d, increment, load_data, global_reset, clock);
output [3:0] q;
input [3:0] d;
input load_data, global_reset, clock, increment;
reg [3:0] q;
always @(posedge clock)
if (global_reset)
q = 4'b0000;
else if (load_data)
q = d;
else if (increment) begin
if (q == 12)
q = 0;
else
q = q + 1;
end
endmodule // counter4_bit
```

В отличие от описания на регистровом уровне, различие в первых строках только одно: выход q объявлен как регистр.

Данная модель позволяет путем проведения небольших модификаций получить описание другого типа счетчика — счетчика в обратном направлении (down counter). Ниже приведен пример такого описания.

```
// 4-bit binary down-counter — of period 13
module counter4_bit(q, d, decrement, load_data, global_reset, clock);
always @(posedge clock)
if (global_reset)
q = 12;
else if (load_data)
q = d;
else if (decrement)
begin
if (q == 0)
q = 12;
else
q = q — 1;
end
endmodule
```

Далее рассмотрим реверсивный счетчик (счетчик с переменным направлением счета, Up-Down counter) с периодом 16.

```
// 4-bit binary up-down-counter — of period 16
module counter4_bit(q, d, updown, count, load_data, global_reset, clock);
always @(posedge clock)
if (reset)
q = 0;
else if (load_data)
q = d;
else if (count)
begin
if (updown)
q = q + 1;
else
q = q — 1;
end
endmodule
```

Временной контроль (Timing Control)

В языке Verilog существуют три варианта осуществления временного контроля (timing control) — использование задержек (delay), событий (event) и уровни чувствительности (level sensitive).

Задержки (Delay)

Синтаксис:

```
timing_control_statement::== delay_based statement*
delay_based::== # delay_value
```

В данном методе вводится задержка между поступлением переменной на вход блока и результатом.

```
initial begin
a = 0; // выполняется в момент t= 0
#10 b = 2; // выполняется в момент t= 10
#15 c = a; // выполняется в момент t= 25
#b c = 4; // выполняется в момент t= 27
b=5; // выполняется в момент t= 27
end
```

Величина задержки может быть определена как постоянная или переменная. Следует помнить, что системное время измеряется в тех единицах, которые установлены в системе. Ниже приводится пример формирования тактового сигнала (creation of a clock signal).

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

В данном примере каждые пять единиц времени сигнал переходит из высокого уровня в низкий, и наоборот.

Событийный контроль (Event-Based control)

При событийном контроле справедлив следующий синтаксис оператора события (в форме Бэкуса — Науэра):

```
event_control_statement::==
@ event_identifier
| @ (event_expression)
event_expression::==
| exp
| event_id
| posedge exp
| negedge exp
| event_exp or event_exp.
```

Происходит вычисление результата при каждом изменении входных сигналов:

```
@ (clock) a = b;// при каждом такте выполняется
a = b
@ (negedge clock) a = b; // когда clock -> 0, то a=b
a = @(posedge clock) b;
```

Защелкивание (Triggers)

Функционирование защелки рассмотрим на следующем примере.

```
event data_in;
always @(negedge clock)
if (data[8]==1) -> data_in;
always @(data_in) mem[0:1] = buf;
```

Здесь в каждом отрицательном фронте синхроимпульса проверяется, равняется ли значение data [8] единице. Если так, то data_in = 1.

Список сигналов возбуждения (Sensitivity List)

Синтаксис:

```
event_list_statement::==@ (event_exp or event_exp)
event_exp::== (event_exp or event_exp).
```

Если мы желаем, чтобы блок выполнялся тогда, когда изменяется любая из входных переменных, то список сигналов возбуждения разделяется операторами or

```
always @ (i_change or he_changes or she_changes)
somebody_changed = 1;
```

Здесь изменение хотя бы одной переменной приводит к изменению остальных.

Задержка распространения в вентиле (Gate Delays)

Синтакс:

```
delay_statement::== gate_name delay_exp gate_details
delay_exp::== # delay_time
```

Данный тип задержки применим только к простейшим вентилям, определяемым в Verilog.

```
and #(5) a1(out, in1, in2);
```

Операторы ветвления (Branch Statements)

Оператор if (if statement)

Синтаксис:

```
if (conditional_expression) statement{ else statement}
```

Как известно, оператор if (if statement) является оператором условного перехода и выполняется стандартным для языков высокого уровня способом. Вычисляется условие, и в зависимости от результата выполняется либо первый оператор, либо второй. Группы операторов в ветви выделяются операторными скобками (как в языке Паскаль) begin… end. Рассмотрим реализацию нашего мультиплексора с помощью оператора If.

```
module multiplexor4_1 (out, in1, in2, in3,in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
assign out = (~cntrl2 & ~cntrl1 & in1) |
( cntrl2 & ~cntrl1 & in2) |
(~cntrl2 & cntrl1 & in3) |
( cntrl2 & cntrl1 & in4) ;
endmodule
```

Ниже приведен пример реализации мультиплексора 4 в 1 с помощью логических уравнений.

```
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)
if (cntrl1==1)
if (cntrl2==1)
out = in4;
else out = in3;
else
if (cntrl2==1)
out = in2;
else out = in1;
endmodule
```

Оператор выбора (case statement)

Рассмотренный выше метод реализации мультиплексора удобен при малом числе входов. При их увеличении гораздо разумнее использовать оператор выбора (case statement), причем его синтаксис аналогичен оператору выбора в C.

Синтаксис:

```
conditional::== case (condition) case_item+ endcase
case_item::== expression+ (seperated by commas): statement* | default: statement*
```

Рассмотрим реализацию мультиплексора 4 в 1 с помощью оператора выбора (case statement).

```
module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
reg out; // note must be a register
always @(in1 | in2 | in3 | in4 | cntrl1 | cntrl2)
case ({cntrl2, cntrl1}) // concatenation
2'b00: out = in1;
2'b01: out = in2;
2'b10: out = in3;
2'b11: out = in4;
default: $display(«Please check control bits»);
endcase
endmodule
```

Вариантами оператора выбора являются операторы casez и casex.

Оператор ветвления (conditional operator)

Оператор по синтаксису аналогичен такому же оператору в C. Так же как и в C, этот оператор может быть заменен конструкцией if-else.

Синтаксис:

```
conditional_expression? true_expression: false_expression
```

Ниже приведен пример реализации мультиплексора 2 в 1:

```
assign out = control? in2: in1;
```

Циклы (Looping Constructs)

В Verilog существует четыре типа циклов — конструкции while, for, repeat и forever. Циклы возможно использовать только внутри блоков initial и always. Основное назначение циклов:

while — выполнение операторов пока условие истинно;

for — в этом цикле возможно инициализировать, проверять и увеличивать индексную переменную явным способом;

repeat — цикл с заданным числом повторений;

forever — вечный цикл, то есть повторение до конца процесса моделирования.

Цикл While (While Loop)

Синтаксис:

```
looping_statement::== while (conditional) statement
```

Цикл while выполняется до тех пор, пока условие истинно. В качестве условия может быть использовано любое логическое выражение. Операторы в цикле группируются с использованием операторных скобок begin… end. Ниже приведен пример использования while.

```
/* How many cups of volume 33 ml does it
take to fill a jug of volume 1 litre */
`define JUGVOL 1000
`define CUPVOL 33
module cup_and_jugs;
integer cup, jug;
initial begin
cup = 0; jug = 0;
while (jug < `JUGVOL) begin
jug = jug + `CUPVOL;
cup = cup + 1;
end // while (jug < JUG_VOL)
$display(«It takes %d cups», cup);
end // initial begin
endmodule // cup_and_jugs
Notice the use of the «define» statement.
```

Цикл For (For Loop)

Синтаксис:

```
for (reg_initialisation ; conditional ; reg_update) statement
```

Цикл for в Verilog аналогичен циклу for в языке C. Пример его использования:

```
integer list [31:0];
integer index;
initial begin
for(index = 0; index < 32; index = index + 1)
list[index] = index + index;
end
```

Цикл Repeat (Repeat Loop)

Синтаксис: looping_statement::== repeat (conditional) statement

Цикл repeat выполняется конечное число раз. Условие (conditional) может быть константой, переменной или сигналом, но обязательно — целым числом. Ниже приведен пример цикла repeat.

```
module comply;
int count;
initial begin
count = 128;
repeat (count) begin
$display(«%d seconds to comply», count);
count = count — 1;
end
end
endmodule
```

Вечный цикл (Forever Loop)

Синтаксис:

```
forever statement
```

Цикл forever выполняется непрерывно, пока моделирование не будет остановлено командой $finish. Ниже приведен пример вечного цикла

```
reg clock;
initial begin
clock = 1'b0;
forever #5 clock = ~clock;
end
initial #2000 $finish;
```

Задание векторов входных сигналов для моделирования (Verilog Input Vectors)

Для задания входных векторов при моделировании системы требуется весьма ограниченное подмножество языка Verilog. Рассмотрим простой пример

```
initial
begin
clk = 1'b0;
globalReset = 1'b1;
in = 1'b1;
end
```

Выражение 1'b0 представляет собой двоичное представление 0.

Существуют две основные части:

```
initial
begin
-----
end
always
begin
-----
end
```

Блок initial начинает выполняться с началом моделирования (simulation), поэтому в нем инициализируются сигналы и определяется последовательность их изменений.

Тактовый сигнал существует в течение всего времени моделирования. Задержка между моментами смены сигналов (событиями, events) определяется с помощью знака #. Например:

```
#160 globalReset = 0;
#160 in = 0;
#160 in = 1;
#320 in = 0;
```

В приведенном примере после 160 единиц времени ожидания сигнал globalReset устанавливается в 0, затем следуют 160 единиц ожидания и установка сигнала in в 0 и т. д.

Наиболее часто конструкция always используется для задания тактового сигнала. Например:

```
always
begin
#10 clk = ~clk;
end
```

Здесь сигнал clk должен быть инвертирован каждые 10 единиц измерения времени. По сути, это означает, что период тактового сигнала — 20 единиц измерения времени (time units).

Кроме того, в процессе моделирования неплохо использовать цикл repeat для задания заданного числа повторений.

```
repeat(10)
begin
#40 in = 0;
#20 in = 1;
end
```

Для генерации случайных чисел в диапазоне [0, n –1] используется следующая команда:

```
{$random} % n
```

Комбинируя эти команды, создадим генератор пачек из 14 случайных чисел каждые 20 единиц времени (time units):

```
repeat(14)
begin
#20 in = {$random} / 16 % 2;
end
```

Список операторов Verilog

Чтобы обобщить рассмотренные ранее ключевые моменты работы с языком, приведем список его операторов.

Тип оператора	Символ	Операция	Число операндов
Арифметические (Arithmetic)	*	Умножение	2
	/	Деление	2
	+	Сумма	2
	–	Разность	2
	%	Остаток	2
Логические (Logical)	!	НЕ	1
	&&	И	2
	||	ИЛИ	2
Отношения (Relational)	>	Больше	2
	<	Меньше	2
	>=	Больше или равно	2
	<=	Меньше или равно	2

Эквивалентности (Equality)	=	=	Равно	2
	!	=	Не равно	2
	=		Выборочное равенство	2
	!		Выборочное неравенство	2
Поразрядные (Bitwise)	~		Побитовое отрицание	1
	&		Побитовое И	2
	|		Побитовое ИЛИ	2
	^		Побитовое искл. ИЛИ	2
	^~or~^		побитовое искл. ИЛИ-НЕ	2
Приведение (Reduction)	&		Приведенное И	1
	~&		Приведенное И-НЕ	1
	|		Приведенное ИЛИ	1
	~|		Приведенное ИЛИ-НЕ	1
	^		Приведенное искл. ИЛИ	1
	^~or^~		Приведенное искл. ИЛИ-НЕ	1
Сдвиг (Shift)	>>		Правый сдвиг	2
	<<		Левый сдвиг	2
Конкатенация (Concatenation)			Объединение	Любое
Репликация (Replication)			Повторение	Любое
Условный (Conditional)	?:		Условный	3

Ключевые слова (Keywords)

Ниже приведены ключевые слова языка Verilog в алфавитном порядке. Естественно, их нельзя использовать для имен вентилей, модулей, портов и т. п.

always, and, assign, attribute, begin, buf, bufif0, bufif1, case, casex, casez, cmos, deassign, default, defpram, disable, edge, else, end, endattribute, endcase, endfunction, endmodule, endprimitive, endspecify, endtable, endtask, event, for, force, forever, fork, function, highz0, highz1, if, initial, inout, input, integer, join, large, macromodule, medium, module, nand, negedge, nmos, nor, not, notif0, notif1, or, output, parameter, pmos, posedge, primitive, pull0, pull1, pulldown, pullup, rcmos, real, realtime, reg, release, repeat, rtranif1, scalared, signed, small, specify, specpram, strength, strong0, strong1, supply0, supply1, table, task, time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg, unsigned, vectored, wait, wand, weak0, weak1, while, wire, wor, xnor, xor.

Директивы компилятора (в алфавитном порядке)

$bitstoreal, $countdrivers, $display, $fclose, $fdisplay, $fmonitor, $fopen, $fstrobe, $fwrite, $finish, $getpattern, $history, $incsave, $input, $itor, $key, $list, $log, $monitor, $monitoroff, monitoron, $nokey.

Типы цепей (Net Types)

supply0, supply1, tri, triand, trior, trireg, tri0, tri1, wand, wire, wor.

Мы вкратце рассмотрели основные конструкции языка Verilog. В следующем занятии школы мы рассмотрим особенности применения нового класса устройств — программируемых аналоговых интегральных схем.

Продолжение следует

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.).