Структуризация описания устройства.

Структуризация описания устройства Позволяет улучшить понимание программы а также обеспечивает возможности многократного использования уже разработанных и отлаженных узлов-**reuse**. (в том числе библиотечных)

Возможности, которые предоставляет Verilog для структуризации программ

-оператор **always**

- программные блоки

-декларация и вызов подпрограмм

- оператор вхождения компонента

Блоки

В предыдущих разделах мы неоднократно использовали блочное представление программы. Составные операторы, вложенные в операторы инициализации и в операторы постоянного повторения, являются простыми примерами блоков. Блок объединяет операторы, связанные общими правилами инициализации. Рассмотрим более подробно концепцию блока, принятую в языке Verilog.

Различают последовательные и параллельные блоки. Формальный синтаксис блока определен следующим образом:

<блок> ::=

<открывающее слово>

⌈ : <имя блока> ⌈ <раздел деклараций блока>⌉ ⌉

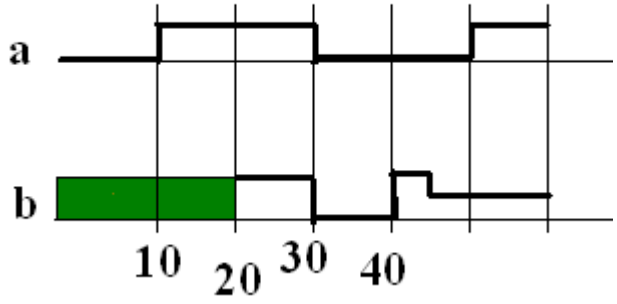
« <оператор> »

<закрывающее слово>

Листинг 3.51

```
initial
 begin a=0;
   #10 a= 1;
   #20 a= 0;
   #70 a=1;
   #20 a=0;
  end
always @(posedge a)
        begin
          fork
            #20 b=0;  //
            #10 b=a; // Это присваивание выполнено раньше предыдущего
            # 30 b=1'b1; // время отсчитывается от фронта сигнала a
```

```
        join
        #5 b=1'bz;  // время отсчитывается от выполнения последнего
                    // оператора в параллельном блоке
    end
endmodule
```



Подпрограммы

<декларация подпрограммы>::=

<заголовок подпрограммы>

« <декларация> »

« <оператор> »

<закрывающее слово подпрограммы>

<заголовок подпрограммы> ::= **task** <имя>; | **function** <имя>;

<закрывающее ключевое слово подпрограммы> ::= **endtask** | **endfunction**

Пример: вычислитель выражения

Z= AxB -CXD

Умножители представлены вызовами задачи mul. Для реализации операций сложения и вычитания используется вызов функции abd_subb, причем из примера видно, что вызовы функции входят в правые части операторов присваивания, в том числе могут содержаться внутри задачи. Результат вычитания произведений присваивается порту result при исполнении оператора **always** , который инициируется положительным перепадом сигнала готовности одного из умножителей.

Листинг 3.52

```
module  mult_and_subb (clock,start, a,b,c,d,result);
input clock,start, a,b,c,d;
output result;
wire [8:1] a,b,c,d;
reg ready;
reg [17:1] result;
```

```
reg [16:1]prod1,prod2;
function  add_subb;
input  opa,opb;
input direction; // 1 - сложение, 0 - вычитание
  if (direction) add_subb=opa+opb;
    else add_subb=opa-opb;
 endfunction

task mul;
parameter size=8;
input clk,start;
input [size:1] opa,opb;
output [2*size:1] product;
output ready;
reg ready, state;
reg [size:1]  shift_opb;
reg [2*size:1] shift_opa,product;
reg [3:0]count;
begin
 @ ( posedge clock or posedge start)
   if (start) state=0;
    else  case (state)
        0: begin if (start)
                shift_opa = opa;
                shift_opb = opb;
                product = 0;
                count=size;
                 ready= 0;
                 state=1;
            end
       1: begin if (count==0) begin
                           state=0;
                           ready=1;
                       end
             else if (shift_opb[1])
                    product = add_subb(product,shift_opa,1);
            shift_opa = shift_opa <<1;
            shift_opb = shift_opb>> 1;
            count= add_subb(count,1,0);
          end
```

```
   endcase
end
endtask
mul (8,clk,start,a,b,prod1,ready);
mul  (8,clk,start,c,d,prod2,);
always @(posedge ready) result=add_subb({1'b0,prod1},{1'b0,prod2},0);
endmodule
```

Подпрограмма определена внутри модуля, поэтому при использовании подпрограмм затруднен **reuse.** Включения модулей в другие модули является более универсальным подходом

Оператор вхождения компонента и иерархические проекты

. Проектные модули, относящиеся к одному проекту, могут находиться в одном файле или представляться несколькими файлами. В последнем случае проектные файлы должны компилироваться в библиотеку проекта в порядке их вхождения в иерархию снизу вверх.

Вхождение проектного модуля в проект высшего иерархического уровня отображается *оператором вхождения* формальный синтаксис которого определяется следующим образом:

<оператор вхождения> ::=

   <имя модуля>⌈ <присвоение значений параметрам> ⌉

   <объявление вхождения> «,<объявление вхождения> »;

<присвоение значений параметрам> ::=

   # (<выражение> «,<выражение> »)

<объявление вхождения> ::=

   <имя вхождения> (список соединений)

<список соединений> ::=

   <соединение порта модуля> «,<соединение порта модуля> »;

   |<именованное соединение> «,<именованное соединение> »

<соединение порта модуля> ::= <выражение> |<пробел>

<именованное соединение> ::= .<имя порта> (⌈<выражение>⌉)

Программа (листинг 3.53) представляет устройство с четырьмя параллельно работающими блоками, каждый из которых выполняет попарное умножение тетрад кодов операндов с последующим взвешенным суммированием частичных произведений по формуле:

Product = mult * fact =

= (16 * mult[7:4] + mult[3:0]) * (16 * fact[7:4] + fact[3:0]) =

= 256(mult[7:4] * fact[7:4]) +

+ 16(mult[7:4] * fact[3:0] + mult[3:0] * fact[7:4]) + fact[3:0] * mult[3:0]=

= 256 * higher + 16 * (middle1 + middle2) + lower;

Для трех встроенных умножителей использовано сопоставление портов по имени, а для одного последнего — позиционное сопоставление.

Пусть в библиотеке есть модуль

```
 module Multiply (clock,.start,.a,.b,
        .result,.ready),
 parameter  n=8,
              m= 16;
input a,b;
wire [n-1:0] a,b;

output result,ready;
reg [m-1:0] result;
reg ready;
…..
endmodule
```

тогда

```
module  mult8x8 (clk,strt, mult,factor,product);
input clk,strt;
input mult,factor;
output product;
wire [7:0] mult,factor;
reg [7:0] v1, v2, v3,v4,elder_part;
reg [8:0] middle_part;
wire [15:0]  product;
wire fin;
multiply #(4,8)
  lower (.clock(clk),.start(strt),.a(mult[3:0]),.b(factor[3:0]),
        .result(v1),.ready(fin)),
  middle1(.clock(clk),.start(strt),.a(mult[7:4]),
        .b(factor[3:0]),.result(v2)),
  middle2(.clock(clk),.start(strt),
          .a(mult[3:0]),.b(factor[7:4]),.result(v3)),
  elder  (clk,strt,mult[7:4],factor[7:4],v4,);
always @(posedge fin)
 begin
   middle_part={1'b0,v2}+ {1'b0,v3}+ {5'd0,v1[7:4]};
   elder_part= {3'd0,middle_part[8:4]}+v4;
 end
assign
product={elder_part, middle_part[3:0],v1[3:0]};
```

end module;

Представление в Verilog типовых цифровых узлов

Комбинационные схемы

В практике проектирования используются различные способы задания переключательных функций

-алгебраический

-алгоритмический

-табличный;

-структурное описание (декомпозиция в выбранном элементном базисе).

Алгебраическая форма представляется операцией присваивания, в правой части которой записывается логическое выражение переключательной функции в обозначениях логических операций принятых в языке С (И - &&, ИЛИ -|| , НЕ - ~).

Но кроме собственно логики весьма существенны правила инициализации.

Оператор **assign** безусловно порождает схему без памяти. Но описание «сложной» логики (например факторизация, повторения) затруднены.

При использовании оператора always можно использовать более гибкие способы описания, включая декомпозиции, условия, циклы. Но при этом надо учитывать следующее

- список чувствительности оператора должен включать **все** аргументы;
- в случаях декомпозиции (факторизации) промежуточные результаты должны вычисляться блокирующими присваиваниями

Алгоритмическая форма эффективна при большом количестве переменных. Используют декомпозицию функции, сводя ее к композиции функций меньшего числа аргументов.

Теоретической основой декомпозиции является разложение Шеннона

$$f(x_1, x_2, \ldots x_N) = \hat{}x_1\, f(0, x_2, \ldots x_N) \vee x_1\, f(1, x_2, \ldots x_N)\ .$$

В языке VerilogHDL такое вычисление может быть представлено фрагментом:

IF (~x1) z= <подформула, полученная из f заменой x_1 на нуль >;
ELSE z= <подформула, полученная из f заменой x_1 на единицу >;

Полученные подформулы в свою очередь могут быть разложении до следующим аргументам.

```
always @ (a,b,c,d)
--z=(b and a) or (b and d and (not c)) or (not b and not a and  c);
begin
 if (b)    z=a || (d &&  ~c);
  else if (~a)    z= ~d || c;
       ELSE      z='b0;
. . .
END
```

Табличная форма

В Verilog возможны следующие подходы для описания комбинационной схемы исходя из табличного задания логической функции

- выборка значения из константного массива. Таблица определяется как массив-параметр, каждый элемент которого равен значению выхода на коде, численный эквивалент которого соответствует индексу этого элемента. Например, для функции предыдущего примера можно записать

```
Parameter my_function='16b;
```

(младшие биты в массиве и в коде индексе слева)

Значение выхода можно получить, используя оператор

```
Assign z= my_function [{d,c,b,a}];
```

Однако такой подход не слишком удобен для описания функций с большим числом переменных.

2. Использование оператора выбора **case** или **casex**

В случаях, когда число нулевых значений функции на всех наборах существенно превышает число единичных (или наоборот) более компактную запись дают оператор выбора. Достаточно перечислить кодовые комбинации, число значений функции на которых превалирует, а остальные комбинации определить как «прочие» (default). Версия casex позволяет объединить варианты, совпадающие только в части входных сигналов (знак ? обозначает входы, состояние которых несущественно)

Для рассматриваемого примера можно записать

```
casex ( { d,c,b,a })
    3'b000 : z_0<=1;
    3"b010 : z_0<= 1;
     3'b1?1: z_0<=1;
     default  z<='0';
```

Примитивы

Подклассом встраиваемых модулей являются примитивы Общее их свойство — они имеют единственный выходной порт. Различают предопределенные примитивы и примитивы, определяемые пользователем (User Defined Primitives, UDP).

<имя вхождения>< and| or | xor> # <число> (<имявыхода>,<имя входа> «,<имя входа>»);

Примитивы, определяемые пользователем

Определение UDP подчиняется следующим синтаксическим правилам:

<UDP> ::=

primitive <имя UDP> (<имя выхода>,
<имя входа> «,<имя входа> »);
input <имя входа> «,<имя входа> »;
output <имя выхода>;
⌈ <спецификация выхода> ⌉ // только для последовательностных UDP
⌈ <определение исходного состояния> ⌉// только для
// последовательностных UDP
<таблица истинности>
endprimitive

Синтаксис таблицы для комбинационного UDP определен следующим образом:

<таблица для комбинационного UDP> ::=

table
« <список значений входов> : <значение выхода комбинационного UDP>;»
endtable

<список значений входов> ::= <значение входа> « <значение входа> »

<значение входа> ::= 0 | 1 | x | ? | b

<значение выхода комбинационного UDP> ::= 0 | 1 | x

В качестве примера приведена программа, содержащая определение примитива vote (голосование), реализующего распространенную функцию большинства

major =x&y | y&z | x&z;

и модуль двухразрядного сумматора, включающий этот примитив.

```
primitive vote (major,x,y,z);
        input x,y,z;
        output major;
table
// x   y   z   :   major
   x   x   ?   :   x;
   ?   x   x   :   x;
   x   ?   x   :   x;

   1   1   ?   :   1;
   ?   1   1   :   1;
   1   ?   1   :   1;
   0   0   ?   :   0;
   0   ?   0   :   0;
   ?   0   0   :   0;
endtable
```

```
endprimitive

module add_2bit ( in1,in2, cin, result,cout);
input  in1,in2, cin;
output result, cout;
wire [1:0] in1,in2,result;
wire [1:0] carry;
vote  #4 c1 (carry[0],in1[0],in2[0],cin),
          c2 (carry[1],in1[1],in2[1],carry[0]);
assign #4 cout=carry[1];
 result[0]=(in1[0] && in2[0] && cin) || carry[0] && (in1[0] || in2[0] ||
                                                    cin),
 result[1]= in1[1] && in2[1] && carry[0] ||
….
Endmodule
```

Триггерные устройства

- асинхронное управление;
- статическое управление, или управление уровнем;
- динамическое управление, или управление фронтом;\
- смешанные варианты.

Асинхронное управление характеризуется тем, что изменение сигналов на информационных входах непосредственно влияет на состояние. Единственный нетривиальный триггер с асинхронным управлением- (исключая смешанное управление) это RS-триггер. Список чувствительности включает оба входа r и s. Тогда можно записать для выхода q

```
always @ (r,s)
   if (r && s)  q='bx;// при одновременной подаче  активных (единичных)
                      // уровней на оба входа состояние не определено
      else if (r) q='b0;
           else  q= s ? 'b1:q;// при одновременной подаче  нулей на оба входа
                      // состояние сохраняется
```

Триггер со статическим управлением

```
always @ (clock,data)
  if (clock)  q=data;
```

описание J-K с динамическим входом и асинхронным сбросом может быть таким

```
always @ (posedge clock or posedge reset)
  if (reset)  q='b0;
  else case({j,k)
        2'b00; q<=q
        2'b01: q<='b0;
         2'b10: q<='b1;
         2'b11: q<=~q;
         Default: g<=bx;// если входные сигналв не определены
```

Определение UDP последовательностного типа (фактически автомата с двумя устойчивыми состояниями, т. е. триггера) отличается, прежде всего, тем, что для выхода вводится дополнительная спецификация, определяющая его как регистровую переменную в соответствии с общими правилами языка Verilog:

```
reg <имя выхода>;
```

Возможно определение начального состояния автомата за счет включения оператора вида:

```
initial <имя выхода> = <значение>;
```

Синтаксис представления таблицы для последовательностного UDP определен следующим образом:

```
<таблица для последовательностного UDP> ::=
  table
 « <список значений входов>:<исходное состояние>:<состояние перехода>; »
  endtable
<состояние перехода> ::= 0 | 1 | x | -
```

Состояние и выход имеют одно и то же значение.

Для последовательностных UDP, управляемых уровнем синхронизирующего сигнала, набор допустимых значений входов и исходных состояний такой же, как набор допустимых значений входов комбинационных UDP. Для определения состояния перехода введен дополнительный символ –, означающий сохранение состояния.

Для примера в листинге 3.57 приведено определение примитива, соответствующего синхронному RS-триггеру с потенциальным управлением (единичный уровень разрешающий). При нулевом сигнале на входе `clk`, а также если на обоих информационных входах нулевые сигналы, состояние триггера не изменяется. Если `clk=1`, а на информационных входах присутствует запрещенная комбинация сигналов или неопределенные сигналы, то состояние элемента считается неопределенным.

Листинг 3.57

```
primitive  rsff_sync(q,clk,r,s);
input clk,r,s;
output q;  reg q;
table
//  clk   r    s    :    q(t)   :    q(t+1)
     0    ?    ?    :      ?    :       —;
     ?    0    0    :      ?    :       —;
     1    0    1    :      ?    :       1;
     1    1    0    :      ?    :       0;
```

```
    1    1    1    :    ?    :    x;
    1    ?    x    :    ?    :    x;
    1    x    ?    :    ?    :    x;

endtable
endprimitive
```

Для последовательностных UDP с динамическим управлением для одного из входов, определенного как синхронизирующий, записывается два значения сигнала в форме `(v,w)`, где `v≠w` и `v,w ∈ {0,1,x,b,?}`. Это означает, что изменение выхода происходит после изменения соответствующего сигнала из состояния `v` в состояние `w`.

Программа (листинг 3.58) иллюстрирует такую запись. Представлен J-K-триггер с асинхронным сбросом по входу `r` (активный низкий уровень) и переключением нарастающим фронтом сигнала `clk`.

листинг3.58

```
primitive  jkff_r(q, r, clk, j, k);
input clk,r,j,k;
output q;  reg q;
initial q=1'b0;
table
//  r    clk      j    k    :    q(t)   :    q(t+1)
    0      ?      ?    ?    :      ?    :    0;
    1    (1?)     ?    ?    :      ?    :    —;
    1    (01)     0    0    :      ?    :    —;
    1    (01)     1    0    :      ?    :    1;
    1    (01)     0    1    :      ?    :    0;
    1    (01)     1    1    :      1    :    0;
    1    (01)     1    1    :      0    :    1;
endtable
endprimitive
```

Описание цифровых автоматов

```
Module moor (reset, clock,x,y)
// декларации сигналов
 F1: assign     //  комбинационная логика  следующего состояния
         b<=F1(x,S);
F2:  assign   //  комбинационная логика выходов
              Y<=F2(S);
 MEM:   always @(posedge clock or posedge reset)-//схема памяти
         If (reset ) s<= s0          //начальная установка
           Else S<=b;// присваивание значения очередному состоянию
endmodule;


Module mealey_sinc (reset, clock,x)
// декларации сигналов
MEM:   always @(posedge clock or posedge reset) -//схема памяти
         If (reset ) s<= s0          //начальная установка
           Else S<= F1(S, x);; //определение следующего состояния
                Y<= F2(S,x); //  определение выхода для следующего такта
endmodule;
```

Таблица переходов

Вход	Исходное состояние			
	S0	S1	S2	S3
X0	S0	S1	S2	S0
X1	S1	S2	S3	S0
X2	S3	S0	S1	S0

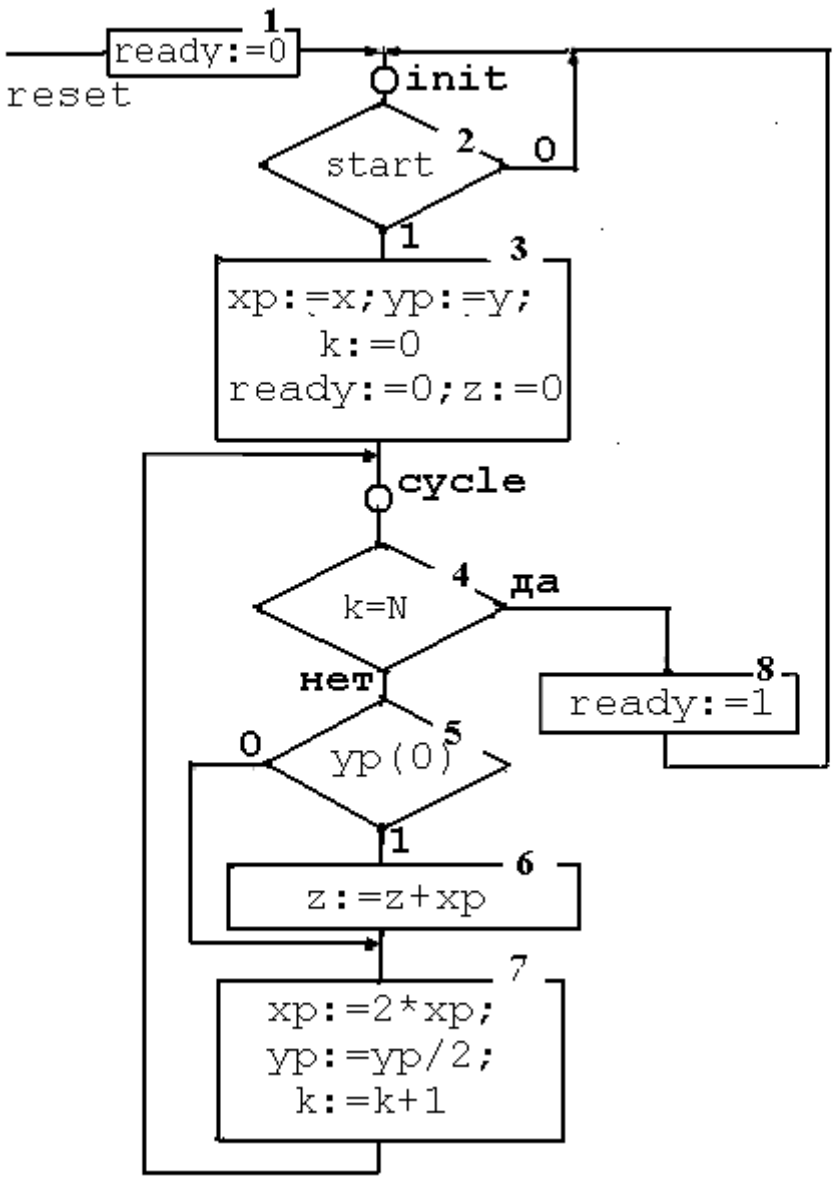
Таблица выходов

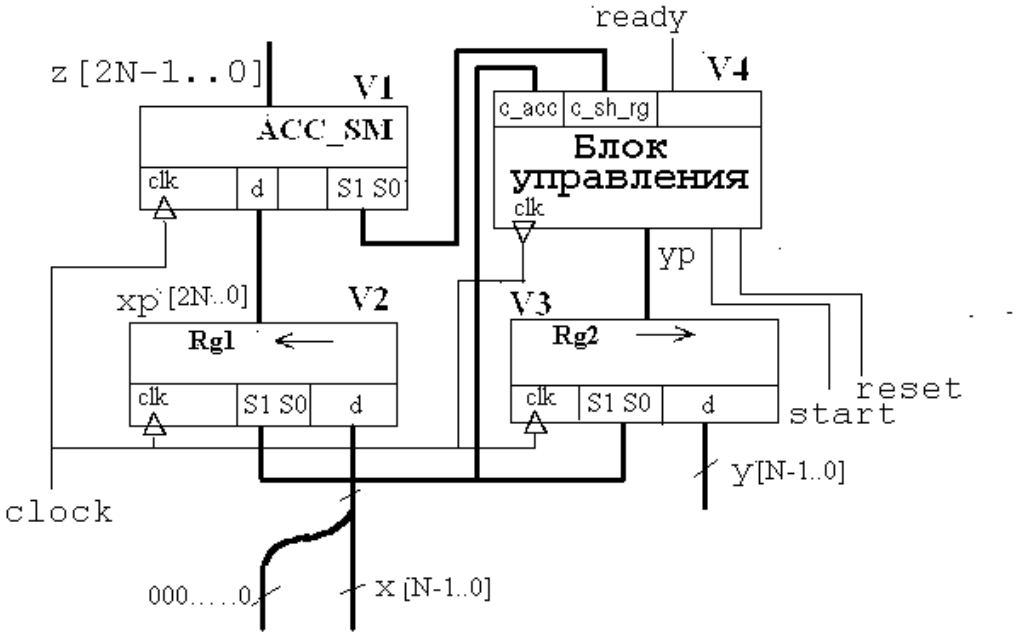
Вход	Исходное состояние			
	S0	S1	S2	S3
X0	Y0	Y0	Y0	Y0
X1	Y0	Y0	Y0	Y1
X2	Y2	Y0	Y0	Y0

```
module st_m (in_data,out_data,reset, clock);
  parameter inp_len=2;
  parameter out_len=2;
  parameter state_len=2;
  parameter[inp_len-1:0]
        x0=2'b00,//кодированиек входов
        x1=2'b01,
        x2=2'b10;

  parameter [out_len-1:0]
        y0=2'b00, //кодирование выходов
         y1=2'b01,
          y2=2'b10;

   parameter [state_len1:0]
        s0=2'b00,//кодирование состояний
         s1=2'b01,
          s2=2'b10,
          s3=2'b11;
  input in_data;  // декларации интерфейса
  output out_data;
  input reset,clock;
wire [inp_len:0]in_data; //декларации
                  //типов(форматов) данных
wire reset, clock;
reg  [state_len-1:0] state;
reg [out_len-1:0] out_data;
```

```
always @ (posedge clock or posedge reset)
begin
out_data=y0;
case(state)
s0 : if (in_data == x0)
                              state <= s0;
          else if (in_data == x2)
                                        state<= s3;
                    else          begin
                          state <=s1; out_data=y2;
                                end
  s1 : if (in_data == x0)
                                          state <= s1;
     else if (in_data==x1)
                                        state = s2;
          else      state <= s0;
s2: if (in_data == x0)
                                        state <= s2;
    else if (in_data == x1)
                                        state <= s3;
          else state <= s1;
default : begin state <= s0;
            out_data = (in_data==x1) ? y0:y1;
        end
endcase
end //always
endmodule
```

Методы построения операционных устройств


1
ready:=0
reset
init
start
2
0
1
3
xp:=x;yp:=y;
k:=0
ready:=0;z:=0
cycle
4
k=N
да
нет
8
ready:=1
0
yp(0)
5
1
6
z:=z+xp
7
xp:=2*xp;
yp:=yp/2;
k:=k+1


ready
z[2N-1..0]
V1
ACC_SM
clk
d
S1 S0
V4
c_acc
c_sh_rg
Блок управления
clk
yp
xp[2N..0]
V2
Rg1
clk
S1 S0
d
V3
Rg2
clk
S1 S0
d
reset
start
y[N-1..0]
clock
000.....0
x[N-1..0]

```
module control_unit(clock, reset, start,y,c_sh_rg, c_acc,
ready);
input clock, reset, start;
input y;
output c_sh_rg, c_acc, ready;
parameter n=8;
enum {init,cycle} state;
wire clock, reset, start,y;
reg     ready;
reg  [1:0]c_sh_rg, c_acc;
int k;
always_comb // комбинат. логика сигналов управления
  case (state)
   init: if (start)  begin
                     c_sh_rg=2'b11; c_acc=2'b01;
                    end
         else    begin
                     c_sh_rg=2'b0; c_acc=2'b0;
                    end
    cycle: if (k==n )   begin
                  c_sh_rg=2'b00; c_acc=2'b00;
                 end
           else  begin c_acc={y,1'b0};
                 c_sh_rg=2'b10;
                 end
   endcase;
//
always_ff @ (posedge clock, posedge reset)
 // begin
     if (reset)
        begin state<=init; ready=0; end
   else if (clock)
       case (state)
          init: if (start) begin
                          state=cycle; ready='b0; k=0;
                        end
          cycle: if (k==n)
                  begin
                       ready=1; k=0; state=init;
                  end
                else  begin
                       k=k+1;
                  end

     endcase;

//END  // always
endmodule;
```

```
interface sys_bus () ;
parameter m=8;
logic clock,reset,start,ready;
logic [m-1:0]x,y;
 logic[2*m-1:0] q;
 modport  monit (output x,y,clock,start,reset,
              input ready,q);
 modport device (input x,y,clock,start,reset,
              output ready,q);
endinterfac
module mp_mult (sys_bus.device a);
parameter m=8;

reg [2*m-1 : 0] xp;
reg [m-1: 0] yp;
reg [1:0] contr_sh_reg, contr_acc_sm;
wire [2*m-1 : 0] x_wide;

assign x_wide= {8'b0,a.x};
acc_sm #(2*m)
v1( .clk(a.clock),.d(xp),.q(a.q),.s(contr_acc_sm));

shift_register #(2*m,1,"left")
v2(.clk(a.clock),
.dl(1'b0),.dr(1'b0),.par_inp(x_wide),.result(xp),.s(contr_sh_reg));

 shift_register #(m,1,"right")
v3(.clk(a.clock),                    .dl(1'b0),.dr(1'b0),.par_inp(a.y),
.result(yp),.s(contr_sh_reg)) ;

  control_unit #(m)
v4 (a.clock,a.reset,a.start, yp[0],
 contr_sh_reg, contr_acc_sm, a.ready);
endmodule;
```