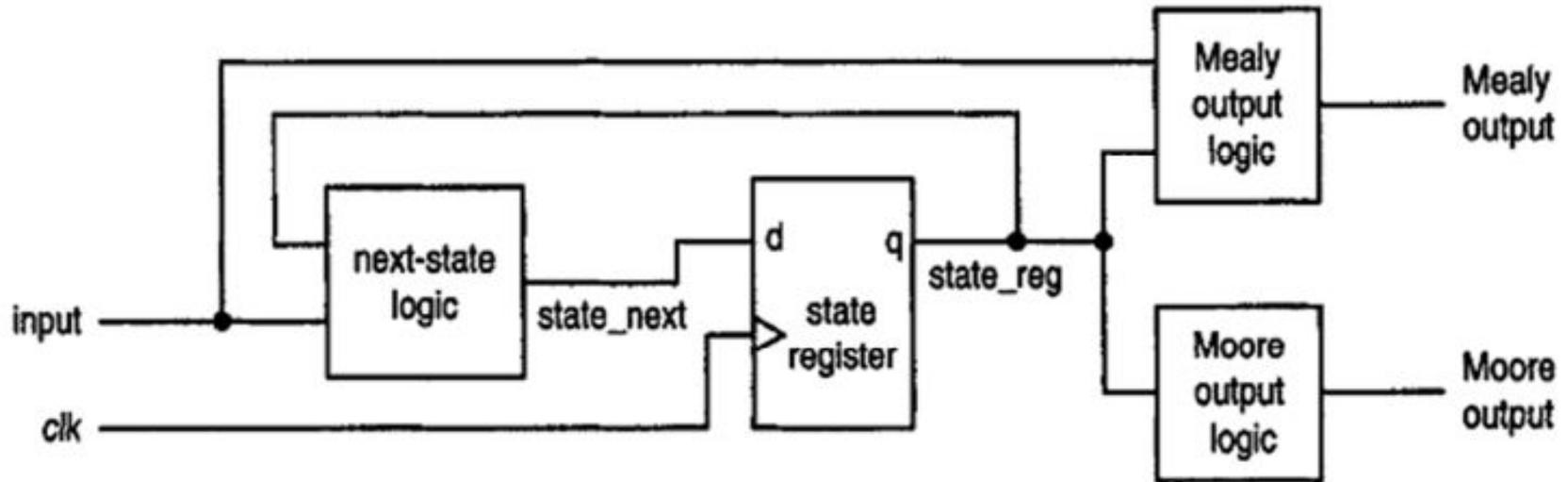
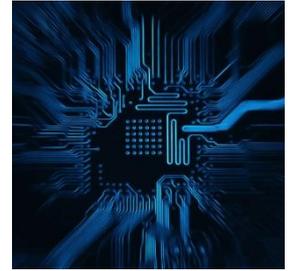


Finite state machines and VHDL

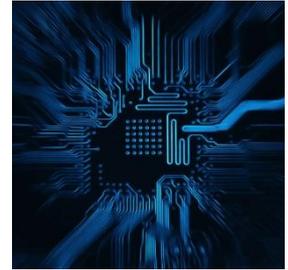
Types of finite state machines



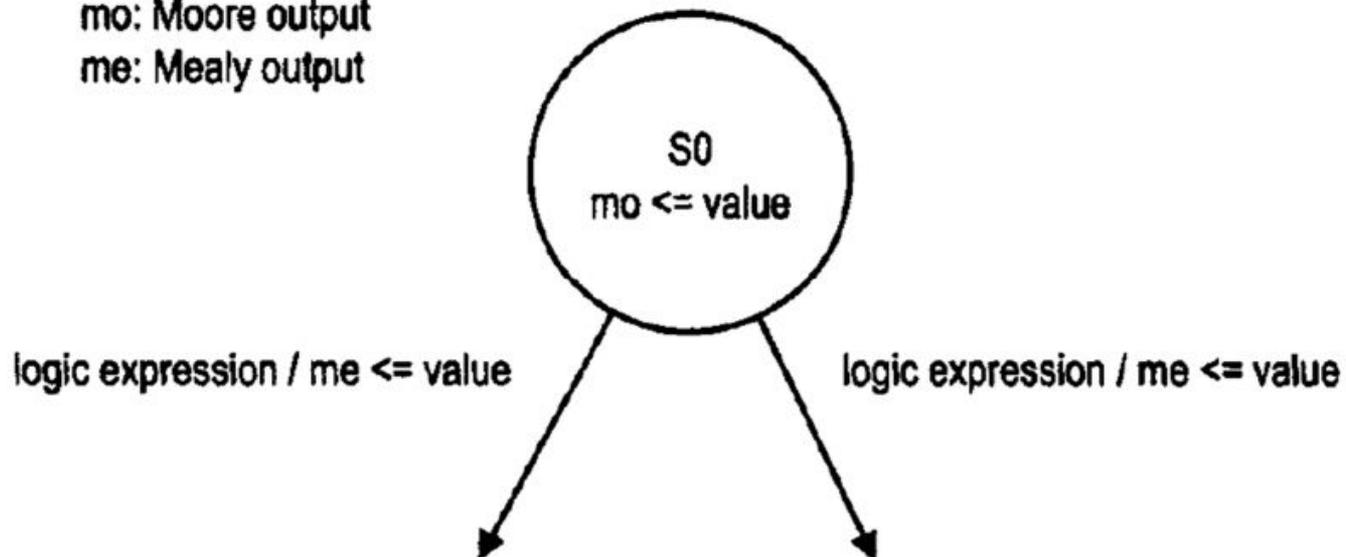
Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs.

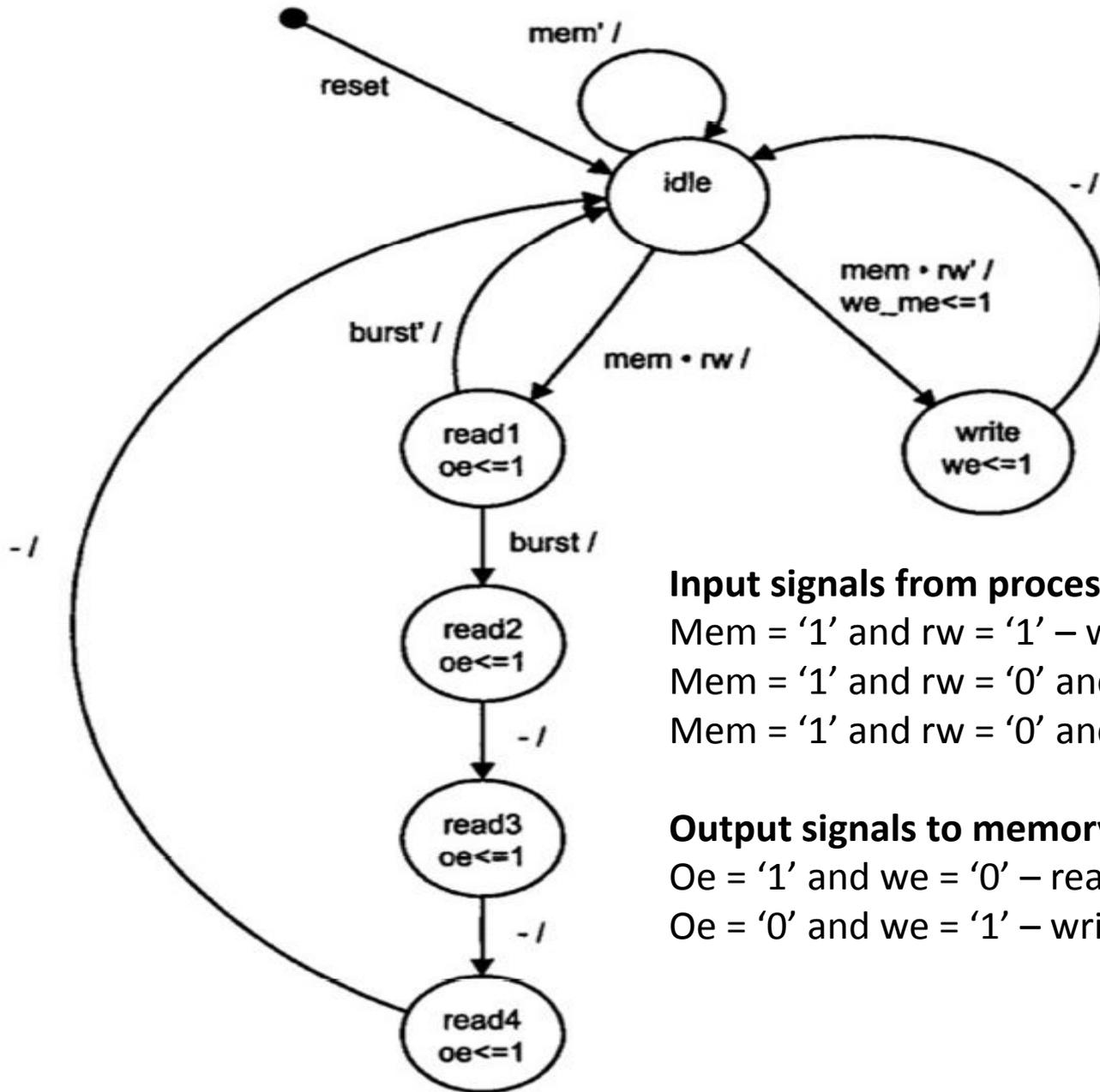
Moore machine is a finite-state machine whose output values are determined only by its current state.

Determination of FSM



mo: Moore output
me: Mealy output





Input signals from processor:

Mem = '1' and rw = '1' – write;

Mem = '1' and rw = '0' and burst = '0' – short read;

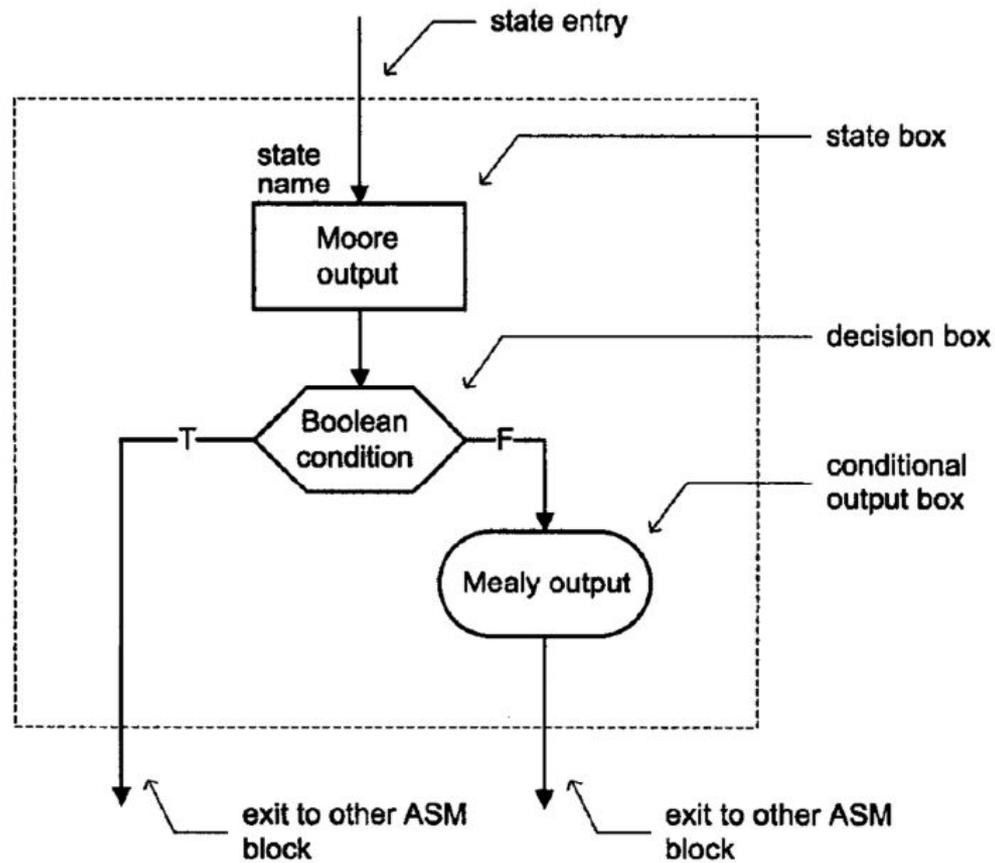
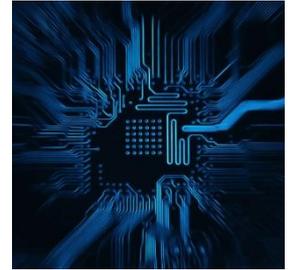
Mem = '1' and rw = '0' and burst = '1' – long read;

Output signals to memory:

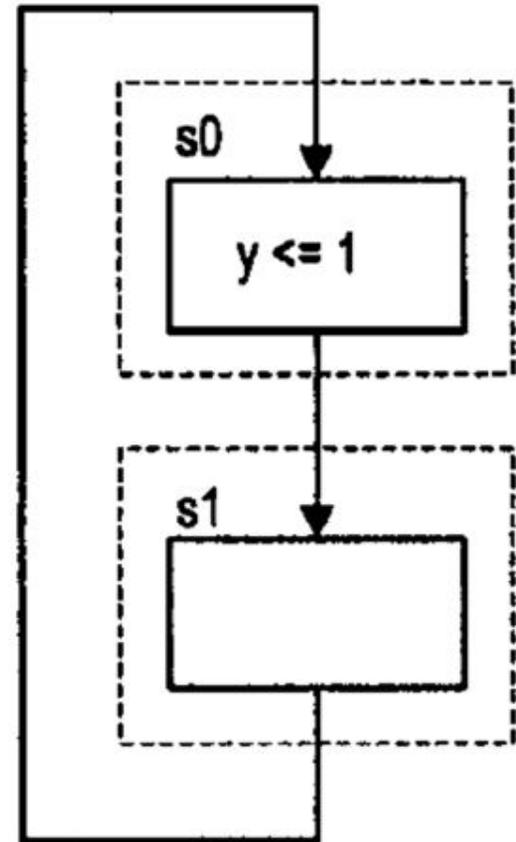
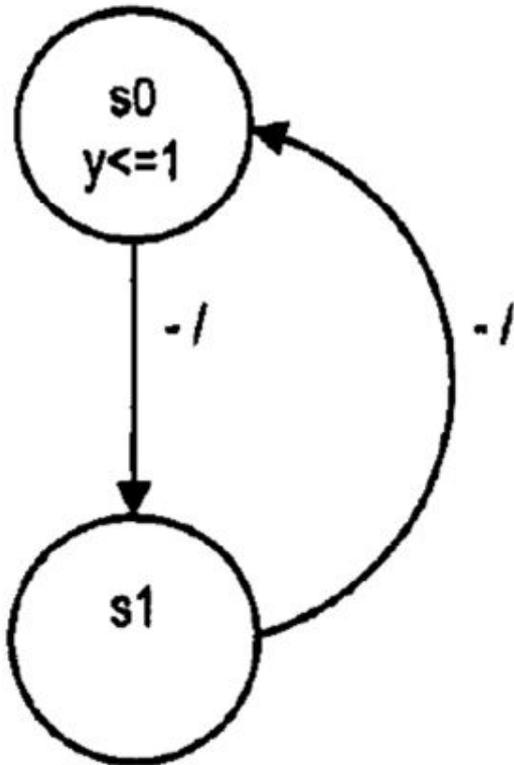
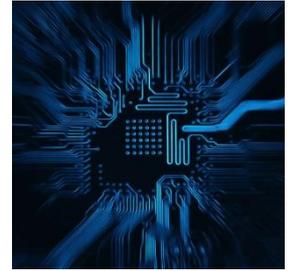
Oe = '1' and we = '0' – read

Oe = '0' and we = '1' – write

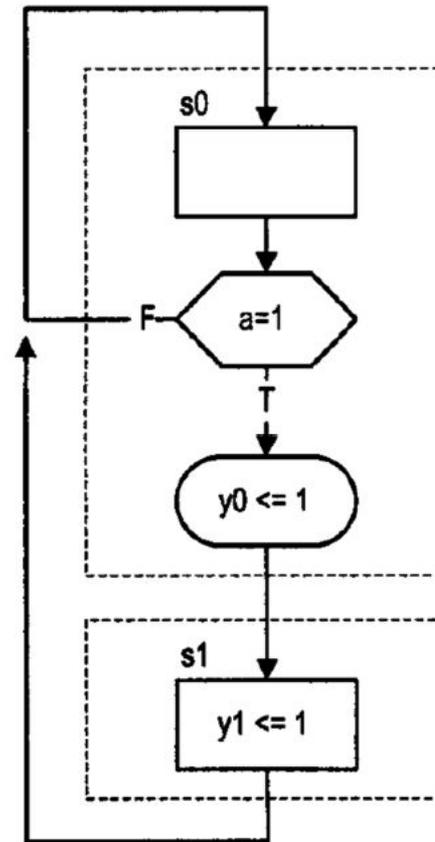
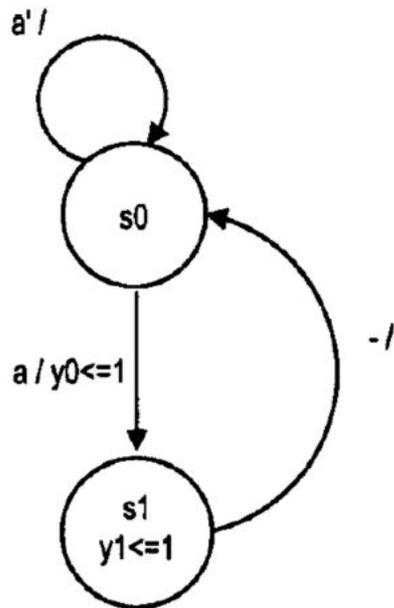
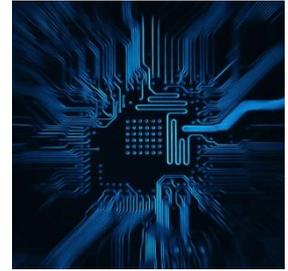
Determination of FSM



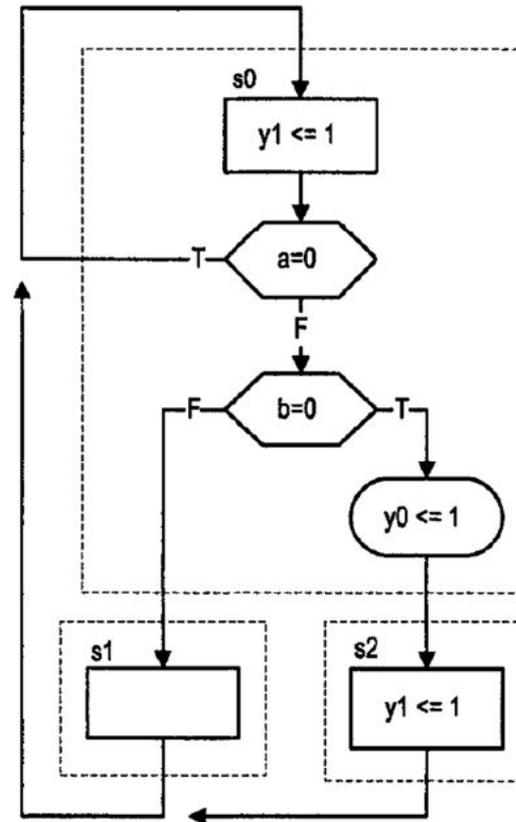
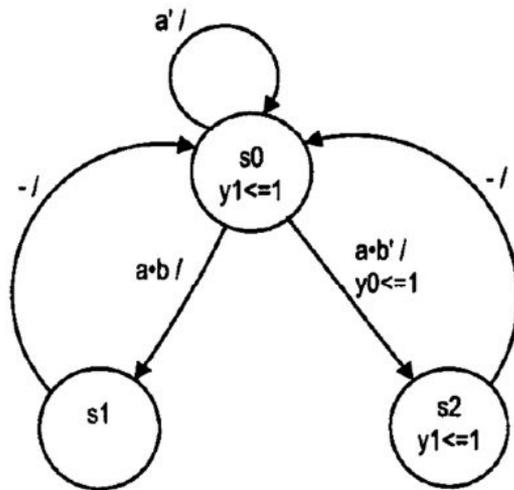
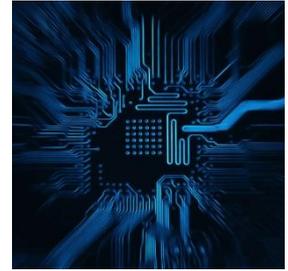
Equality of the representations



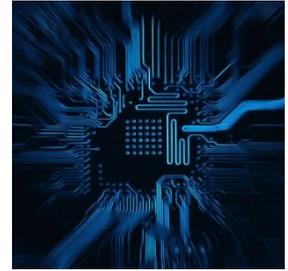
Equality of the representations



Equality of the representations

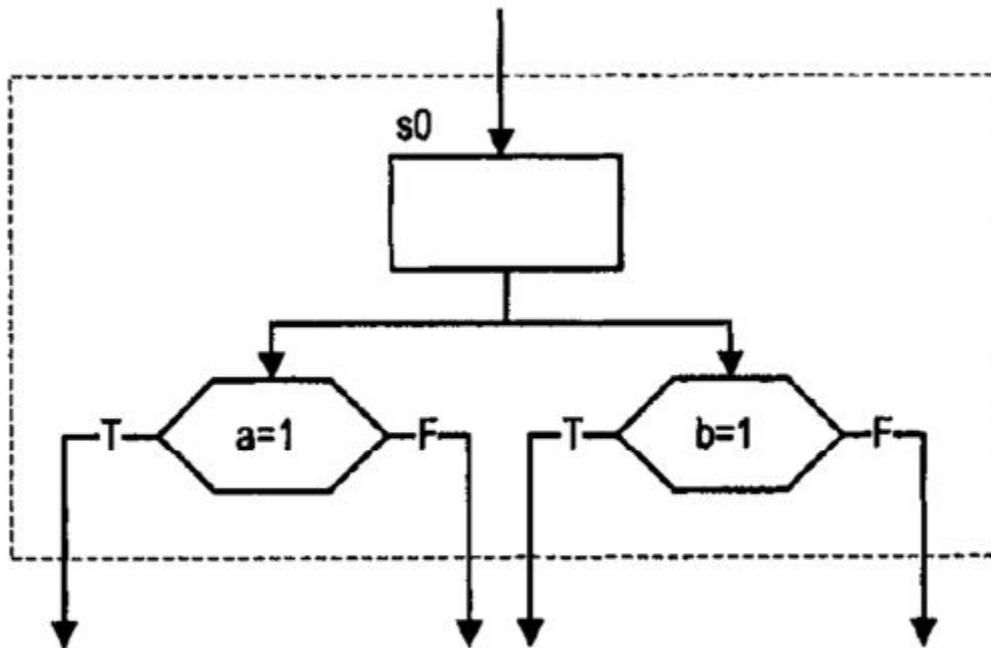
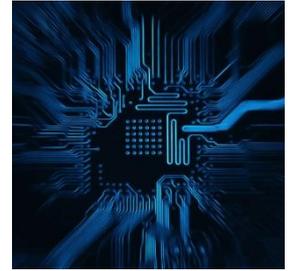


Main rules

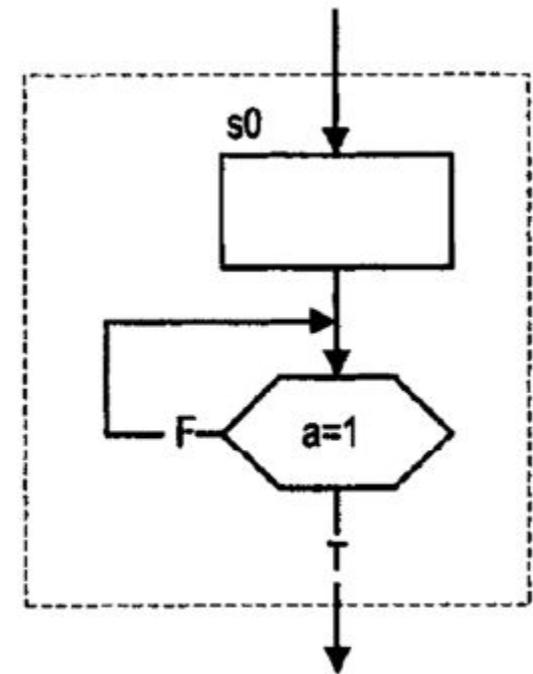


1. Each input combination
2. Каждой входной комбинации должен соответствовать единственный выходной путь из данного узла ГСА. Допускается использование одного выходного пути для нескольких входных комбинаций, но не наоборот.
3. Выходной путь из узла ГСА обязательно должен вести к блоку состояния (либо другого, либо этого же узла ГСА).

Ошибки при составлении ГСА.

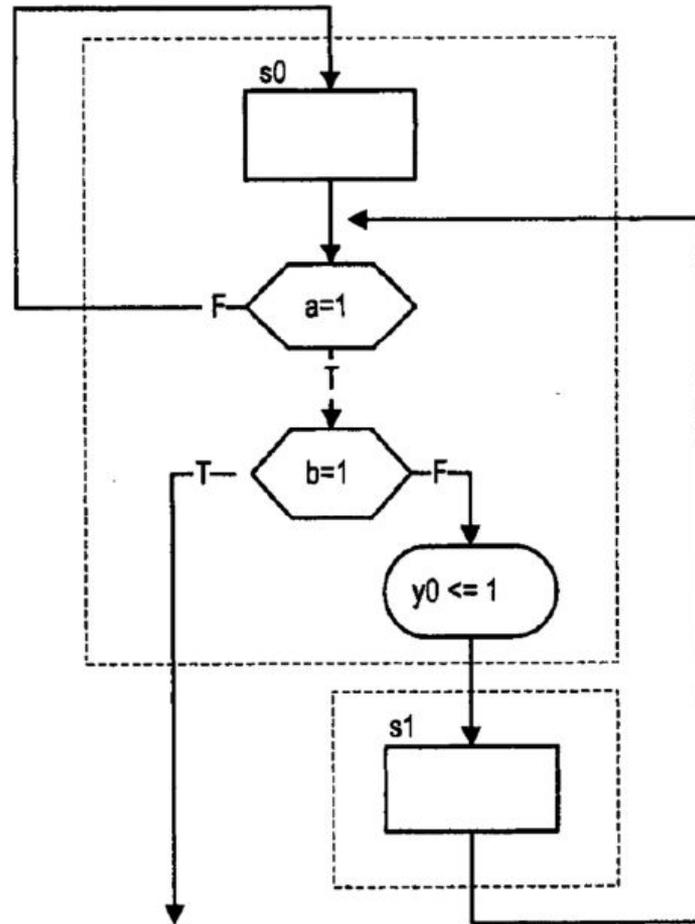
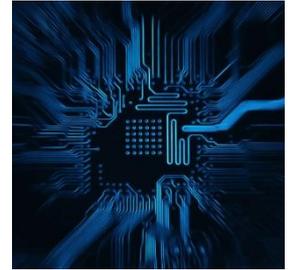


(a)



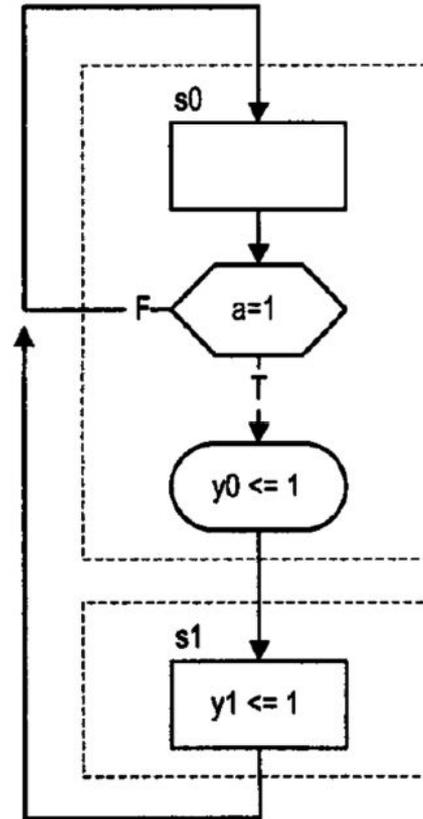
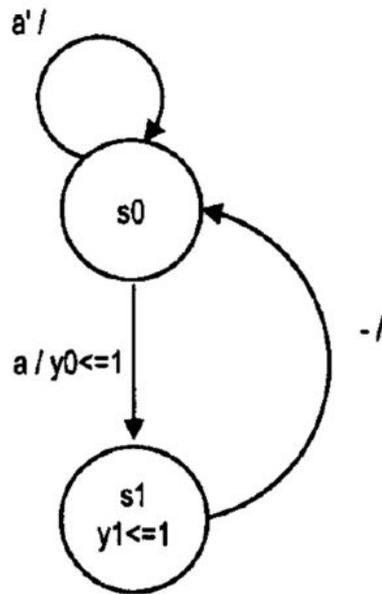
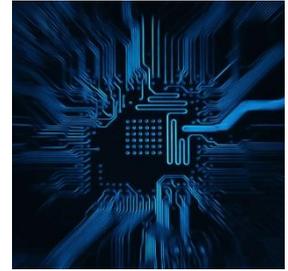
(b)

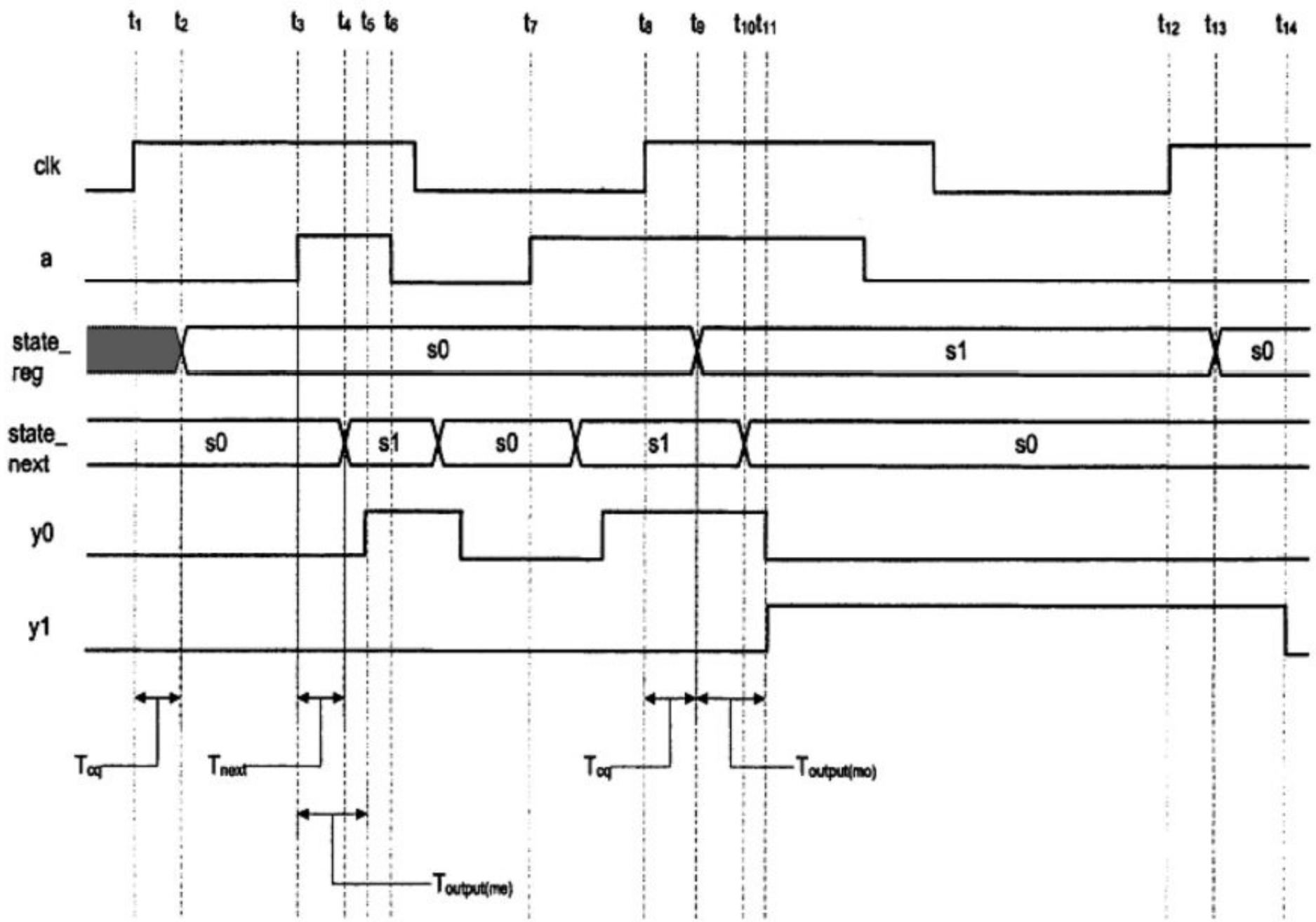
Ошибки при составлении ГСА.



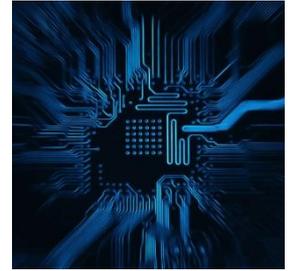
(c)

Временная диаграмма конечного автомата.





Описание конечных автоматов на VHDL.



1. Для описания состояний конечного автомата на VHDL используется перечислимый тип данных.

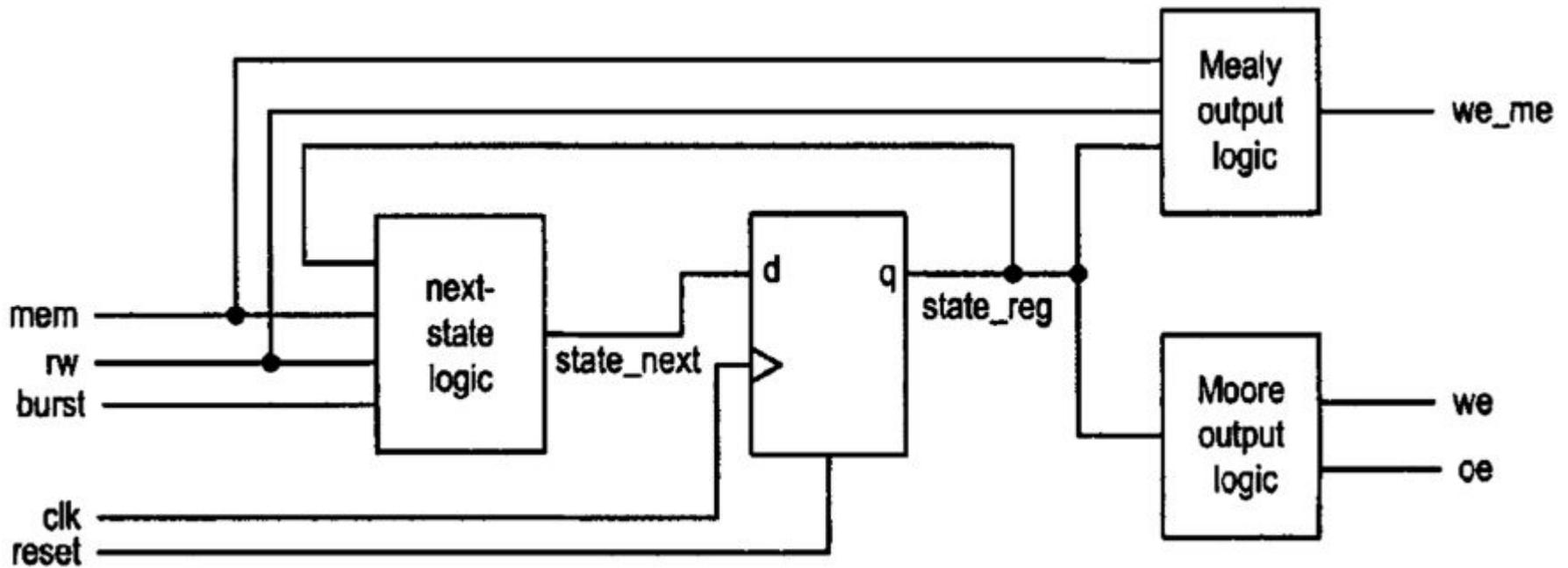
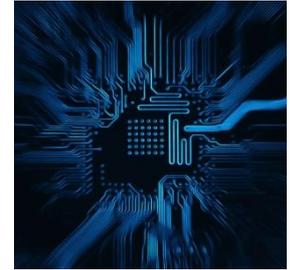
```
type state_type is (state_0, state_1, ..., state_n);
```

2. Необходимо отделить описание элементов памяти от описания логики вычисления следующего состояния и логики вычисления значений выходных сигналов.

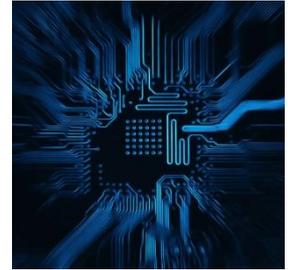
Возможные способы описания:

1. Мульти-сегментный тип программирования;
2. Двух-сегментный тип программирования;
3. Одно-сегментный тип программирования.

Мульти-сегментный тип программирования.



Мульти-сегментный тип программирования.



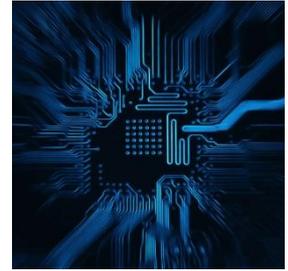
```
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
  port(
    clk, reset: in std_logic;
    mem, rw, burst: in std_logic;
    oe, we, we_me: out std_logic
  );
end mem_ctrl ;
```

```
architecture mult_seg_arch of mem_ctrl is
  type mc_state_type is
    (idle, read1, read2, read3, read4, write);
  signal state_reg, state_next: mc_state_type;
begin
```

Описание входных и
выходных сигналов

Задание типа
mc_state_type перечисляя
все состояния.

Мульти-сегментный тип программирования.

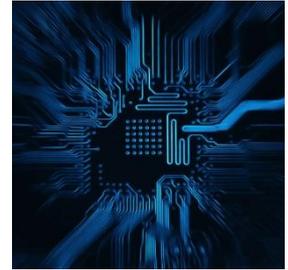


```
-- state register
process (clk, reset)
begin
    if (reset='1') then
        state_reg <= idle;

    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
```

Создание регистра state_reg с асинхронным обнулением.

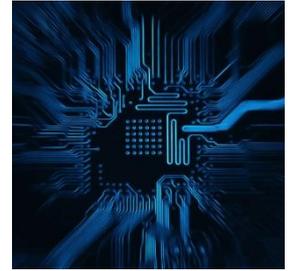
Мульти-сегментный тип программирования.



```
process (state_reg, mem, rw, burst)
begin
  case state_reg is
    when idle =>
      if mem='1' then
        if rw='1' then
          state_next <= read1;
        else
          state_next <= write;
        end if;
      else
        state_next <= idle;
      end if;
    when write =>
      state_next <= idle;
    when read1 =>
      if (burst='1') then
        state_next <= read2;
      else
        state_next <= idle;
      end if;
    when read2 =>
      state_next <= read3;
    when read3 =>
      state_next <= read4;
    when read4 =>
      state_next <= idle;
  end case;
end process;
```

Определение состояния
сигнала state_next

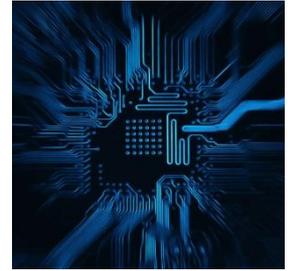
Мульти-сегментный тип программирования.



```
process (state_reg)
begin
  we <= '0'; -- default value
  oe <= '0'; -- default value
  case state_reg is
    when idle =>
    when write =>
      we <= '1';
    when read1 =>
      oe <= '1';
    when read2 =>
      oe <= '1';
    when read3 =>
      oe <= '1';
    when read4 =>
      oe <= '1';
  end case;
end process;
```

Комбинационная схема
определения выходных
сигналов Мура.

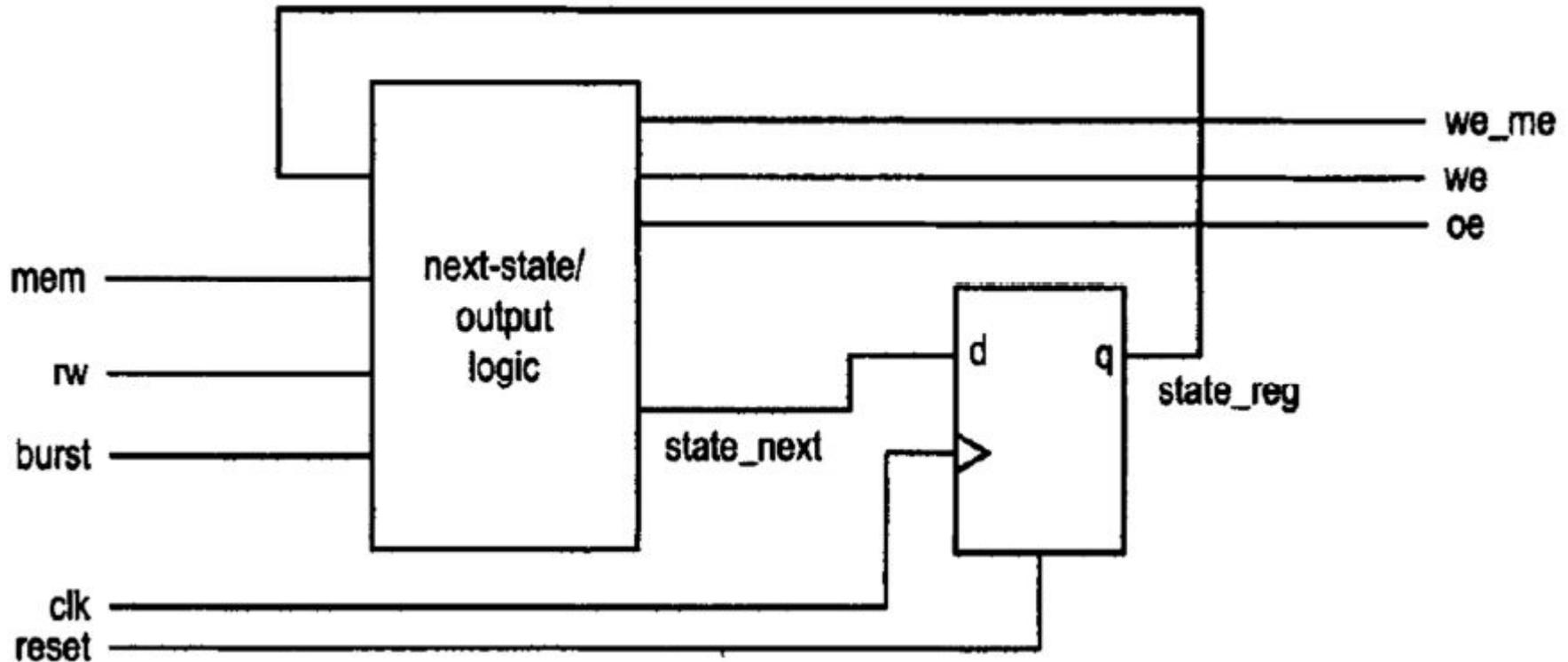
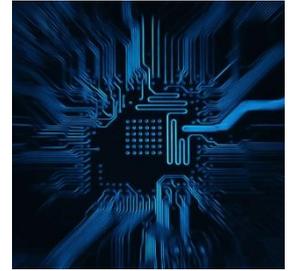
Мульти-сегментный тип программирования.



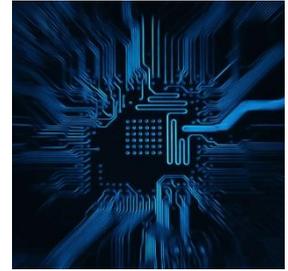
```
process (state_reg, mem, rw)
begin
  we_me <= '0'; -- default value
  case state_reg is
    when idle =>
      if (mem='1') and (rw='0') then
        we_me <= '1';
      end if;
    when write =>
    when read1 =>
    when read2 =>
    when read3 =>
    when read4 =>
  end case;
end process;
```

Комбинационная схема определения выходных сигналов Мили.

Двух-сегментный тип программирования.



Двух-сегментный тип программирования.

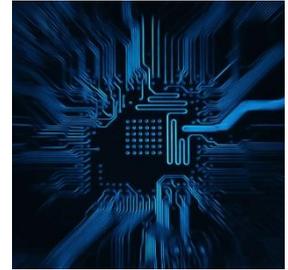


```
-- state register
process (clk, reset)
begin
    if (reset='1') then
        state_reg <= idle;

    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
```

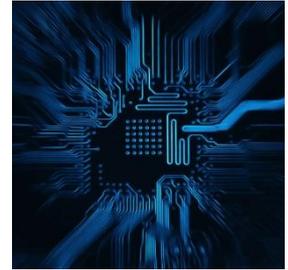
Создание регистра state_reg с асинхронным обнулением.

Двух-сегментный тип программирования.



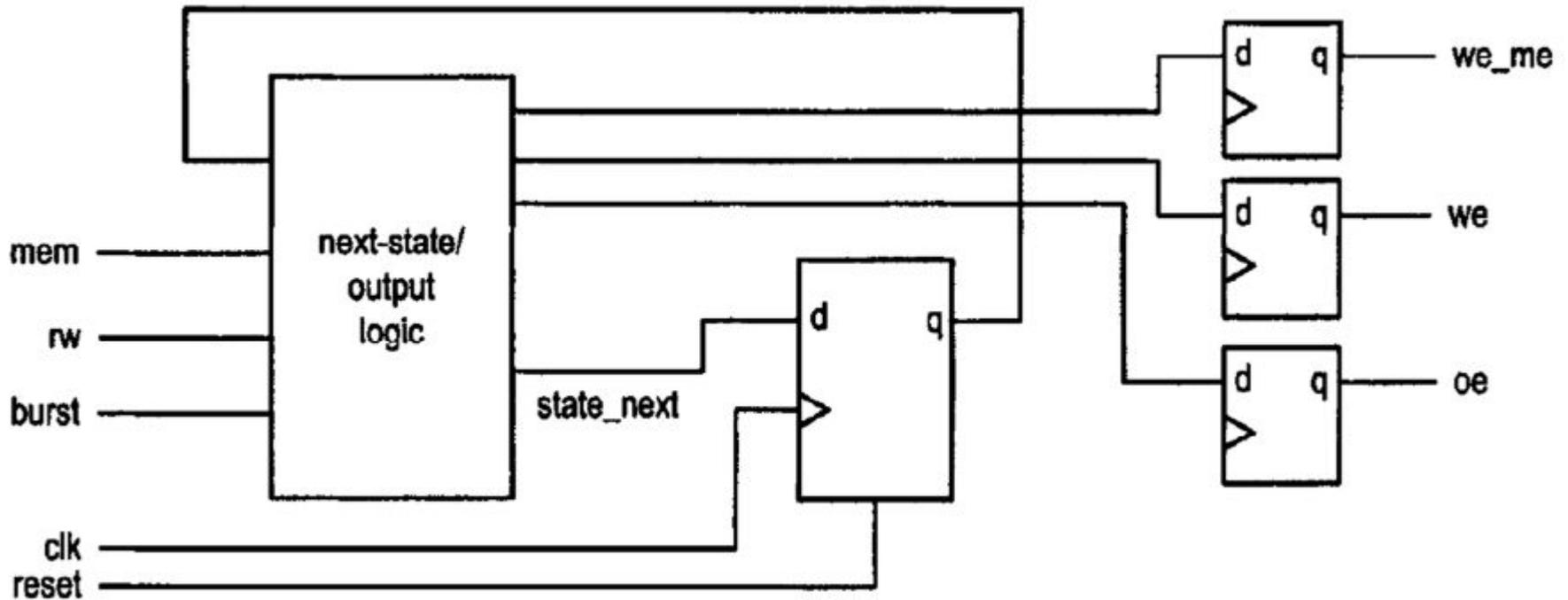
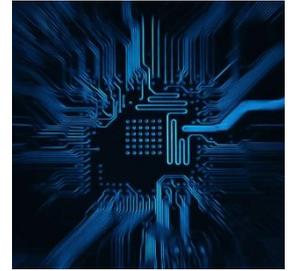
```
process (state_reg, mem, rw, burst)
begin
  oe <= '0';    -- default values
  we <= '0';
  we_me <= '0';
  case state_reg is
    when idle =>
      if mem='1' then
        if rw='1' then
          state_next <= read1;
        else
          state_next <= write;
          we_me <= '1';
        end if;
      else
        state_next <= idle;
      end if;
    when write =>
      state_next <= idle;
      we <= '1';
    when read1 =>
      if (burst='1') then
        state_next <= read2;
      else
        state_next <= idle;
      end if;
      oe <= '1';
    when read2 =>
      state_next <= read3;
      oe <= '1';
    when read3 =>
      state_next <= read4;
      oe <= '1';
    when read4 =>
      state_next <= idle;
      oe <= '1';
  end case;
end process;
end two_seg_arch;
```

Одно-сегментный тип программирования.

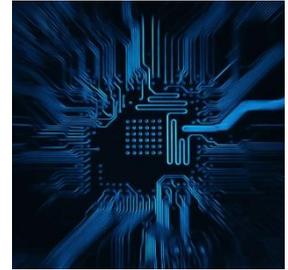


```
architecture one_seg_wrong_arch of mem_ctrl is
  type mc_state_type is
    (idle, read1, read2, read3, read4, writ
  signal state_reg: mc_state_type;
begin
  process(clk,reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      oe <= '0';      -- default values
      we <= '0';
      we_me <= '0';
      case state_reg is
        when idle =>
          if mem='1' then
            if rw='1' then
              state_reg <= read1;
            else
              state_reg <= write;
              we_me <= '1';
            end if;
          else
            state_reg <= idle;
          end if;
        when write =>
          state_reg <= idle;
          we <= '1';
        when read1 =>
          if (burst='1') then
            state_reg <= read2;
          else
            state_reg <= idle;
          end if;
          oe <= '1';
        when read2 =>
          state_reg <= read3;
          oe <= '1';
        when read3 =>
          state_reg <= read4;
          oe <= '1';
        when read4 =>
          state_reg <= idle;
          oe <= '1';
      end case;
    end if;
  end process;
end one_seg_wrong_arch;
```

Одно-сегментный тип программирования.

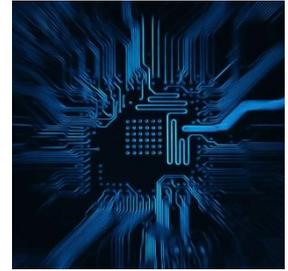


Кодирование состояний



	Binary assignment	Gray code assignment	One-hot assignment	Almost one-hot assignment
idle	000	000	000001	00000
read1	001	001	000010	00001
read2	010	011	000100	00010
read3	011	010	001000	00100
read4	100	110	010000	01000
write	101	111	100000	10000

Схема детектора фронта



Конечный автомат

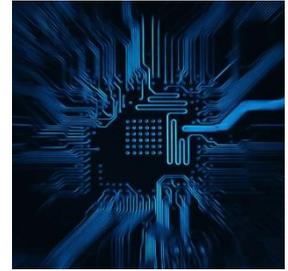
Мура

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detector1 is
port(
    clk, reset: in std_logic;
    strobe: in std_logic;
    p1: out std_logic
);
end edge_detector1;

architecture moore_arch of edge_detector1 is
type state_type is (zero, edge, one);
signal state_reg, state_next: state_type;
begin
```

```
-- state register
process(clk,reset)
begin
    if (reset='1') then
        state_reg <= zero;
    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
```

Схема детектора фронта



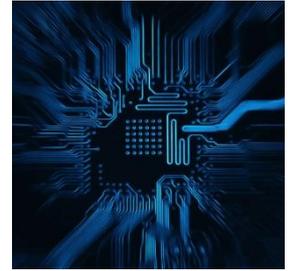
Конечный автомат

Мура

```
process(state_reg, strobe)
begin
  case state_reg is
    when zero=>
      if strobe = '1' then
        state_next <= edge;
      else
        state_next <= zero;
      end if;
    when edge =>
      if strobe = '1' then
        state_next <= one;
      else
        state_next <= zero;
      end if;
  end case;
end process;
```

```
when one =>
  if strobe = '1' then
    state_next <= one;
  else
    state_next <= zero;
  end if;
end case;
end process;
-- Moore output logic
p1 <= '1' when state_reg=edge else
'0';
end moore_arch;
```

Схема детектора фронта



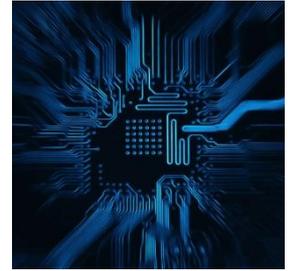
Конечный автомат

Мили

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detector2 is
port(
    clk, reset: in std_logic;
    strobe: in std_logic;
    p2: out std_logic
);
end edge_detector2;
architecture mealy_arch of edge_detector2 is
type state_type is (zero, one);
signal state_reg, state_next: state_type;
begin
```

```
-- state register
process(clk,reset)
begin
    if (reset='1') then
        state_reg <= zero;
    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
-- next-state logic
process(state_reg,strobe)
begin
```

Схема детектора фронта



Конечный автомат

```
case state_reg is
  when zero=>
    if strobe = '1' then
      state_next <= one;
    else
      state_next <= zero;
    end if;
  when one =>
    if strobe = '1' then
      state_next <= one;
    else
      state_next <= zero;
    end if;
end case;
end process;
```

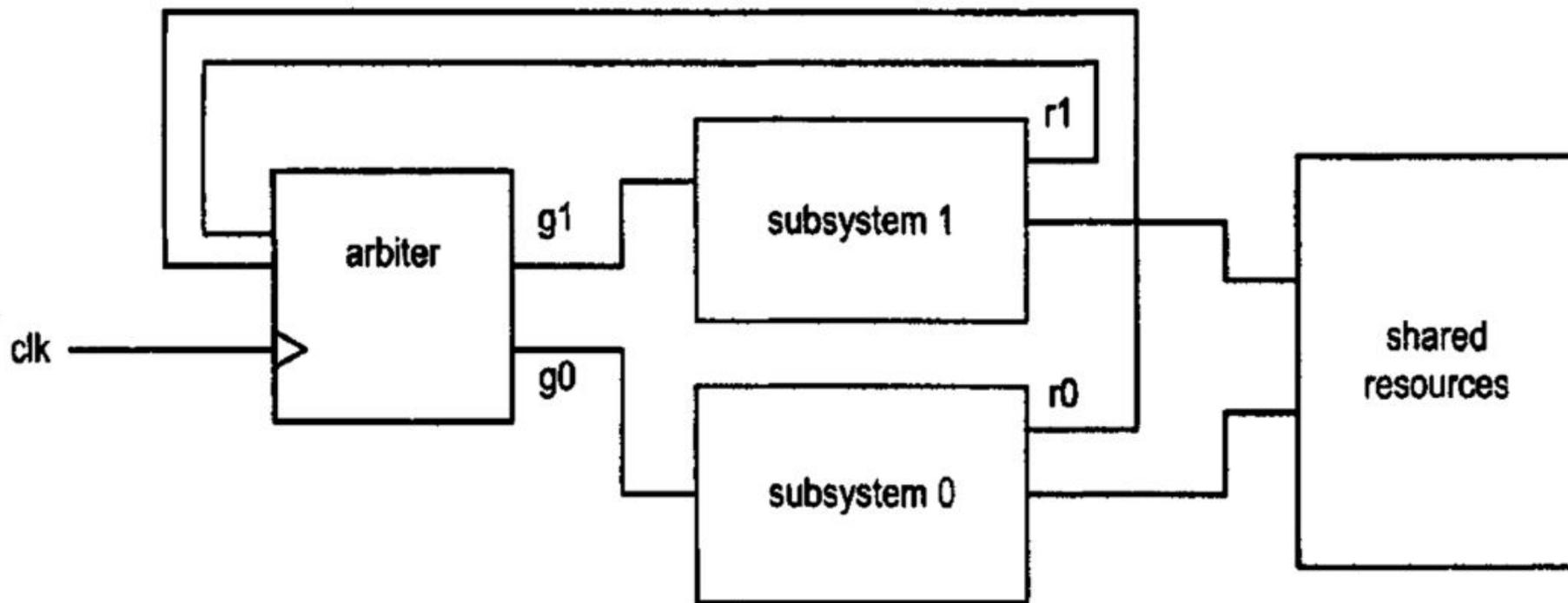
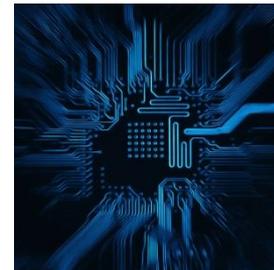
Мили

-- Mealy output logic

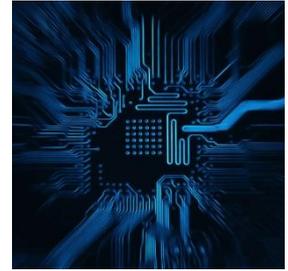
```
p2 <= '1' when (state_reg=zero) and (strobe='1') else  
'0';
```

```
end mealy_arch;
```

Арбитр



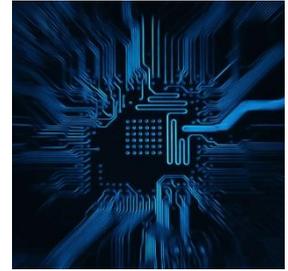
Арбитр



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity arbiter2 is
port(
    clk: in std_logic;
    reset: in std_logic;
    r: in std_logic_vector(1 downto 0);
    g: out std_logic_vector(1 downto 0)
);
end arbiter2;
architecture fixed_prio_arch of arbiter2 is
type mc_state_type is (waitr, grant1, grant0);
signal state_reg, state_next: mc_state_type;
begin
```

```
-- state register
process(clk,reset)
begin
    if (reset='1') then
        state_reg <= waitr;
    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
-- next-state and output logic
```

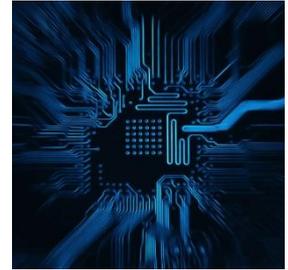
Арбитр



```
process(state_reg,r)
begin
g <= "00"; -- default values
case state_reg is
  when waitr =>
    if r(1)='1' then
      state_next <= grant1;
    elsif r(0)='1' then
      state_next <= grant0;
    else
      state_next <= waitr;
    end if;
```

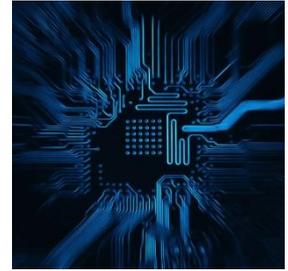
```
  when grant1 =>
    if (r(1)='1') then
      state_next <= grant1;
    else
      state_next <= waitr;
    end if;
    g(1) <= '1';
  when grant0 =>
    if (r(0)='1') then
      state_next <= grant0;
    else
      state_next <= waitr;
    end if;
    g(0) <= '1';
  end case;
end process;
end fixed_prio_arch;
```

Арбитр (синхронный доступ)



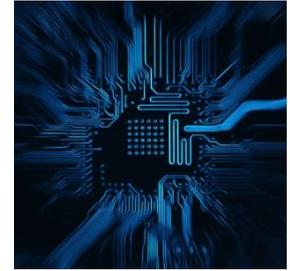
```
when waitr =>  
  if r(1)='1' then  
    state_next <= grant1;  
    g(1) <= '1'; -- newly added line  
  elsif r(0)='1' then  
    state_next <= grant0;  
    g(0) <= '1'; -- newly added line  
  else  
    state_next <= waitr;  
  end if;
```

Арбитр (циклический приоритет)



```
architecture rotated_prio_arch of arbiter2 is  
type mc_state_type is (waitr1, waitr0, grant1, grant0);  
signal state_reg, state_next: mc_state_type;  
begin  
-- state register  
process(clk,reset)  
begin  
    if (reset='1') then  
        state_reg <= waitr1;  
    elsif (clk'event and clk='1') then  
        state_reg <= state_next;  
    end if;  
end process;
```

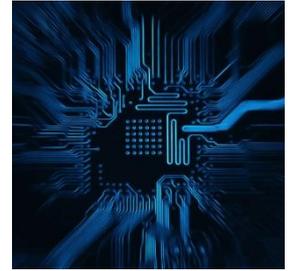
Арбитр (циклический приоритет)



```
-- next-state and output logic
process(state_reg,r)
begin
g <= "00"; -- default values
case state_reg is
  when waitr1 =>
    if r(1)='1' then
      state_next <= grant1;
    elsif r(0)='1' then
      state_next <= grant0;
    else
      state_next <= waitr1;
    end if;
  when waitr0 =>
    if r(0)='1' then
      state_next <= grant0;
```

```
    elsif r(1)='1' then
      state_next <= grant1;
    else
      state_next <= waitr0;
    end if;
  when grant1 =>
    if (r(1)='1') then
      state_next <= grant1;
    else
      state_next <= waitr0;
    end if;
    g(1) <= '1';
  when grant0 =>
    if (r(0)='1') then
      state_next <= grant0;
    else
      state_next <= waitr1;
    end if;
end case;
end process;
end rotated_prio_arc;
```

Практическое задание

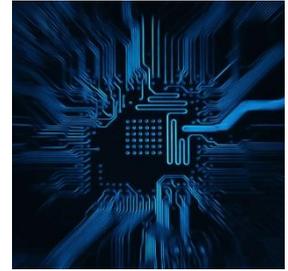


Задача 1

В задачах цифровой связи для того, чтобы обозначить начало пакета, используется специальная синхронизирующая последовательность бит – преамбула. Например, в Ethernet II преамбула включает повторяющиеся октеты "10101010". Мы хотим разработать конечный автомат, который генерирует последовательность "10101010". Схема имеет входной сигнал `start` и выход `data_out`. Когда `start` выставляется в '1', последовательность "10101010" « генерируется в течение следующих восьми периодов синхронизирующего сигнала. (Указание: реализуйте достаточно универсальную схему, чтобы ее можно было приспособить под другую преамбулу.)

- Нарисовать диаграмму состояний.
- Преобразовать диаграмму состояний в граф-схему алгоритма.
- Написать соответствующий граф-схеме алгоритма код на VHDL.

Практическое задание



Задача 2

Модифицируйте генератор преамбулы таким образом, чтобы он преобразовывал заданную в параллельном коде последовательность бит в последовательный код. Схема имеет входной сигнал `start`, входной сигнал `data_in` (8 бит) и выход `data_out`. Когда `start` выставляется в '1', генерируется последовательность, заданная сигналом `data_in`, в течение следующих восьми периодов синхронизирующего сигнала.

Домашнее задание

- 1. В режиме пакетного чтения ("burst") контроллера памяти неявно требуется, чтобы процессор сначала выставлял сигналы `rw` и `mem` на один период синхронизирующего сигнала, а затем выставлял сигнал `burst` на следующий период синхронизирующего сигнала. Упростите требования к процессору таким образом, чтобы он выставлял сигнал `burst` в течение того же периода синхронизирующего сигнала, что и сигналы `rw` и `mem`.
 - а. Нарисовать диаграмму состояний нового конечного автомата.
 - б. Преобразовать диаграмму состояний в граф-схему алгоритма.
 - в. Написать соответствующий граф-схеме алгоритма код на VHDL.
- 2. Модифицируйте детектор фронта сигнала так, чтобы он реагировал на фронт ($0 \rightarrow 1$) и спад ($1 \rightarrow 0$) входного сигнала. Т.е. схема должна генерировать короткий импульс (на один период синхронизирующего сигнала), как только величина входного сигнала `strobe` изменяется. Реализовать две архитектуры: с выводами Мура и с выводами Мили.
 - а. Нарисовать диаграмму состояний.
 - б. Преобразовать диаграмму состояний в граф-схему алгоритма.
 - в. Написать соответствующий граф-схеме алгоритма код на VHDL.

Домашнее задание

- 3. Разработайте конечный автомат, детектирующий последовательность "10101010" во входном сигнале на стороне получателя. Схема имеет входной сигнал `data_in` и выходной сигнал `match`. Сигнал `match` выставляется в '1' на один период синхронизирующего сигнала сразу после того, как обнаружена последовательность "10101010". (Указание: помните об универсальности.)
 - а. Нарисовать диаграмму состояний.
 - б. Преобразовать диаграмму состояний в граф-схему алгоритма.
 - с. Написать соответствующий граф-схеме алгоритма код на VHDL.

Спасибо за внимание