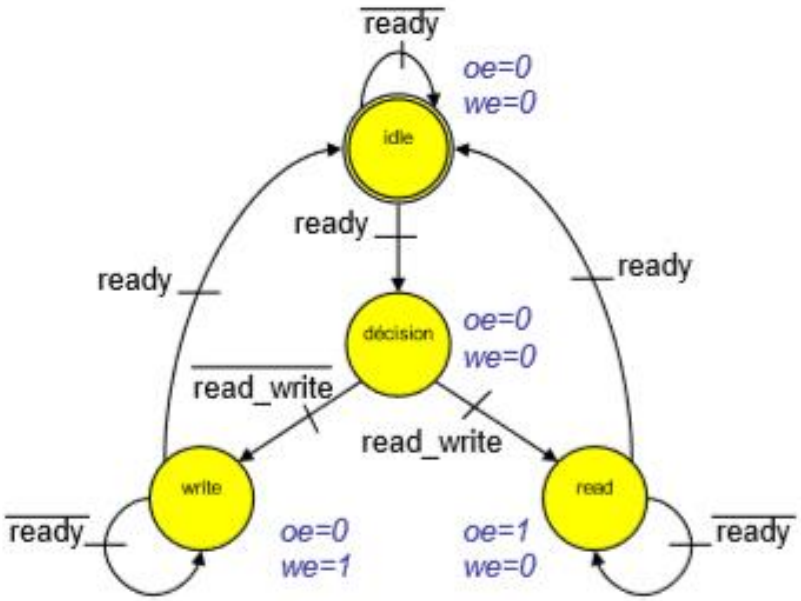
COMPTE RENDU TP 2 VHDL

L'objectif de ce travail est d'illustrer la description comportementale en langage VHDL de machines à états finis autonomes de type *Moore* ou *Mealy*.

CONTRÔLEUR DE MEMOIRE

PREPARATION

Nous réalisons en VHDL une entité et une architecture correspondant au Diagramme suivant, décrivant le fonctionnement d'un contrôleur de mémoire.

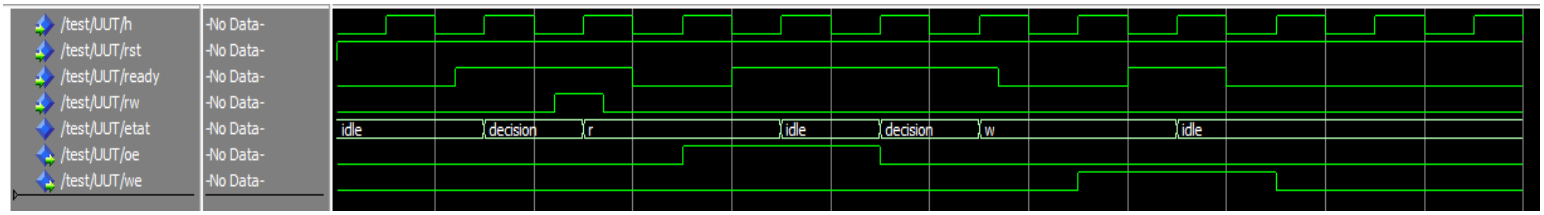


```
1   entity mem_control is
2           port (
3                   ready, rw, h, rst:      bit;
4                   oe,we:                  out bit
5           );
6   end entity;
9   architecture beh_1 of mem_control is
10
11          type defetat is (idle,decision,r,w);
12          signal etat: defetat;
13  begin
14          process (h,rst)
15          begin
16                  if rst = '0' then etat <= idle;
17
18                  elsif h = '1' and h'event then
19
20                          case etat is
21                                  when idle =>
22                                          if ready = '1'  then etat <= decision; end if;
23                                  when decision =>
24                                          if rw = '1'     then etat <= r;
25                                          else etat <= w; end if;
26                                  when r =>
27                                          oe <= '1';
28                                          if ready = '1'  then etat <= idle;
29                                          else etat <= r; end if;
30                                  when w =>
31                                          we <= '1';
32                                          if ready = '1'  then etat <= idle;
33                                          else etat <= w; end if;
34                          end case;
35
36                  end if;
37          end process;
38  end architecture;
39
```

Nous définissons ensuite le *testbench* qui va nous permettre de tester cette architecture.

```
6   architecture bench_1 of test is
7
8           component mem_control is
9                   port (
10                          ready, rw, h, rst:      bit;
11                          oe,we:                  out bit
12                  );
13          end component;
14
15          signal ready, rw, h, rst, oe, we:       bit;
16
17          for UUT: mem_control use entity work.mem_control(beh_1);
18
19  begin
20
21          UUT:    mem_control port map (ready=>ready, rw=>rw, h=>h, rst=>rst, oe=>oe, we=>we);
22
23          rst     <=      '1';
24          h       <=      not h after 5 ns;
25          ready   <=      '1' after 12 ns, '0' after 30 ns, '1' after 40 ns, '0' after 67 ns, '1' after 80 ns, '0' after 90 ns;
26          rw      <=      '1' after 22 ns, '0' after 27 ns;
27
28  end architecture;
```

En lançant la simulation, nous obtenons le chronogramme suivant.



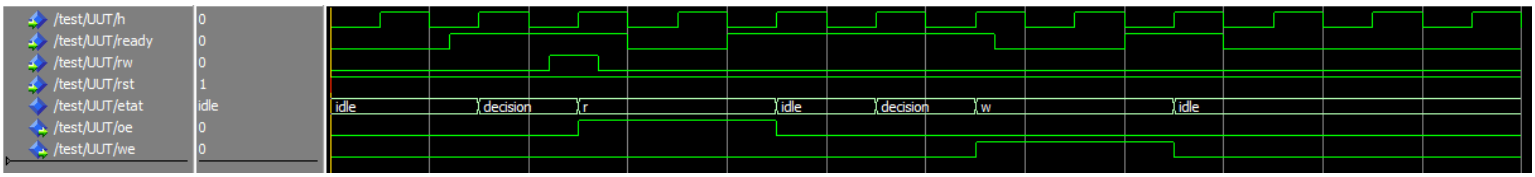
Nous observons que le chronogramme correspond bien au Diagramme des états. Seulement, on voit une latence d'un coup d'horloge entre les entrés et les sorties. Cela vient du fait qu'il faille d'abord calculer l'état interne pour ensuite, au prochain tour du process, modifier les sorties.

Ce problème devrait être réglé en créant deux process différents, un pour l'horloge et le *reset*, et un autre pour les autres signaux.

Voici la nouvelle architecture implémentant les deux process.

```
43  architecture beh_2 of mem_control is
44
45          type defetat is (idle,decision,r,w);
46          signal etat, netat: defetat;
47  begin
48          m: process (h,rst)
49          begin
50                  if rst='0' then etat <= idle;
51                  elsif h='1' and h'event then
52                          etat <= netat;
53                  end if;
54          end process;
55
56          mem: process (ready, rw, etat)
57          begin
58                  netat <= etat;
59                  oe <= '0';
60                  we <= '0';
61
62                  case etat is
63                          when idle =>
64                                  if ready = '1' then netat <= decision;  end if;
65                          when decision =>
66                                  if rw = '1' then netat <= r;
67                                  else netat <= w; end if;
68                          when r =>
69                                  oe <= '1';
70                                  if ready = '1' then netat <= idle;
71                                  else netat <= r; end if;
72                          when w =>
73                                  we <= '1';
74                                  if ready = '1' then netat <= idle;
75                                  else netat <= w; end if;
76                  end case;
77          end process;
78  end architecture;
79
```

Pour la simulation, nous gardons le même *testbench* que précédemment, en utilisant l'architecture *beh_2* à la place de *beh_1*. Voici le chronogramme correspondant.

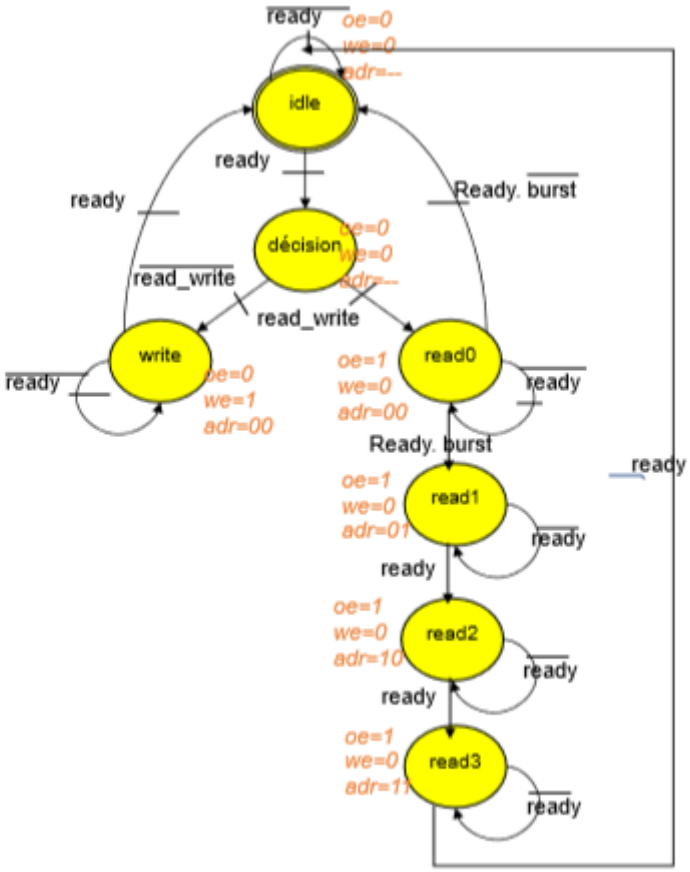


On observe bien ce qui était prévu. En effet, sur ce chronogramme on observe que les sorties sont bien synchronisées avec l'état interne.

Nous concluons que cette méthode d'implémentation est plus rapide.

Implémentation du *BURST*

Nous ajoutons la fonctionnalité *burst* dont les caractéristiques sont décrites dans le diagramme suivant.

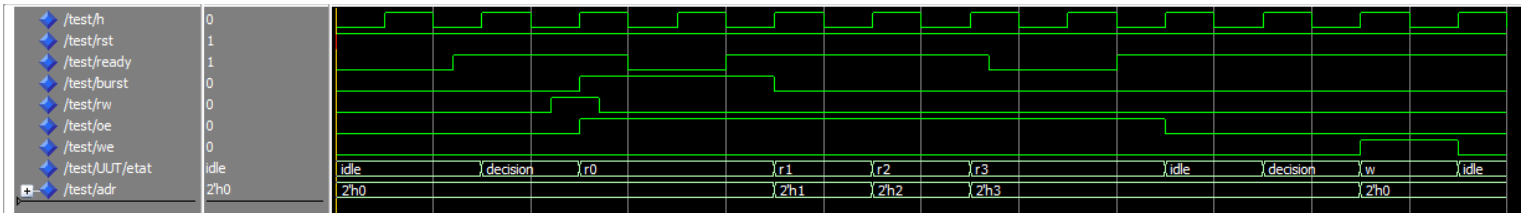


Voici le code VHDL correspondant.

```
1  entity mem_control_burst is
2          port (
3                  ready, rw, h, rst, burst :      bit;
4                  oe, we :                        out bit;
5                  adr :   out bit_vector (1 downto 0)
6          );
7  end entity;
```

```
9  architecture beh of mem_control_burst is
10
11         type defetat is (idle,decision,r0,r1,r2,r3,w);
12         signal etat, netat: defetat;
13 begin
14         m: process (h,rst)
15         begin
16                 if rst='0' then etat <= idle;
17                 elsif h='1' and h'event then
18                         etat <= netat;
19                 end if;
20         end process;
21         mem: process (ready, rw, etat)
22         begin
23                 netat <= etat;
24                 oe <= '0';
25                 we <= '0';
26                 adr <= "00";
27                 case etat is
28                         when idle =>
29                                 if ready = '1' then netat <= decision;  end if;
30                         when decision =>
31                                 if rw = '1' then netat <= r0;
32                                 else netat <= w; end if;
33                         when r0 =>
34                                 oe <= '1';
35                                 if ready = '1' then netat <= r1;
36                                 else netat <= r0; end if;
37                         when r1 =>
38                                 oe <= '1';
39                                 adr <= "01";
40                                 if ready = '1' then netat <= r2;
41                                 else netat <= r1; end if;
42                         when r2 =>
43                                 oe <= '1';
44                                 adr <= "10";
45                                 if ready = '1' then netat <= r3;
46                                 else netat <= r2; end if;
```

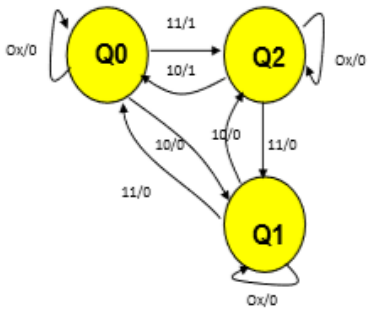
```
47                         when r3 =>
48                                 oe <= '1';
49                                 adr <= "11";
50                                 if ready = '1' then netat <= idle;
51                                 else netat <= r3; end if;
52                         when w =>
53                                 we <= '1';
54                                 if ready = '1' then netat <= idle;
55                                 else netat <= w; end if;
56                 end case;
57         end process;
58 end architecture;
59
```



Le chronogramme correspond bien au diagramme d'états. La fonction *burst* est opérationnelle.

COMPTEUR

Nous voulons réaliser un compteur d'après la description suivante.



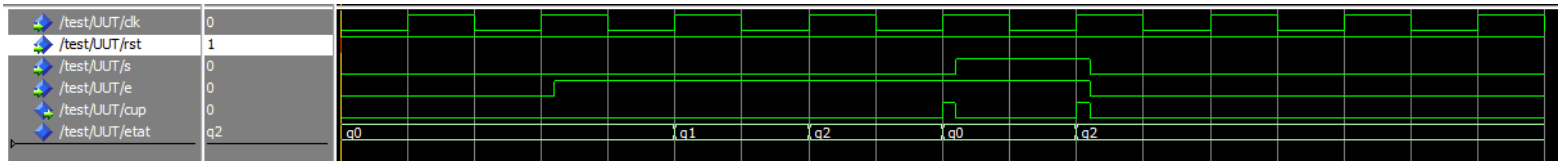
Nous commençons par décrire l'entité et l'architecture correspondante.

```
entity compteur is
        port (
                s, e, clk, rst :        bit;
                cup :   out bit
        );
end entity;

architecture beh of compteur is

        type defetat is (q0, q1, q2);
        signal etat, netat :    defetat;
        signal z:       bit;
begin
        p1: process (clk, rst)
        begin
                if rst = '0' then etat <= q0;
                elsif clk = '1' and clk'event then etat <= netat;
                end if;
        end process;

        p2: process (etat, s, e)
        begin
                netat <= etat;
                z <= '0';
                if e = '1' then
                        case etat is
                                when q0 =>
                                        if s = '1' then z <= '1'; netat <= q2;
                                        else netat <= q1;
                                        end if;
                                when q1 =>
                                        if s = '1' then netat <= q0;
                                        else netat <= q2;
                                        end if;
                                when q2 =>
                                        if s = '1' then netat <= q1;
                                        else z <= '1'; netat <= q0;
                                        end if;
                        end case;
                end if;
        cup <= z;
        end process;
end architecture;
```

Nous souhaitons comparer deux scénarios différents : le premier avec les entrées synchrones et le second avec les entrées asynchrones.

PREMIER SCENARIO

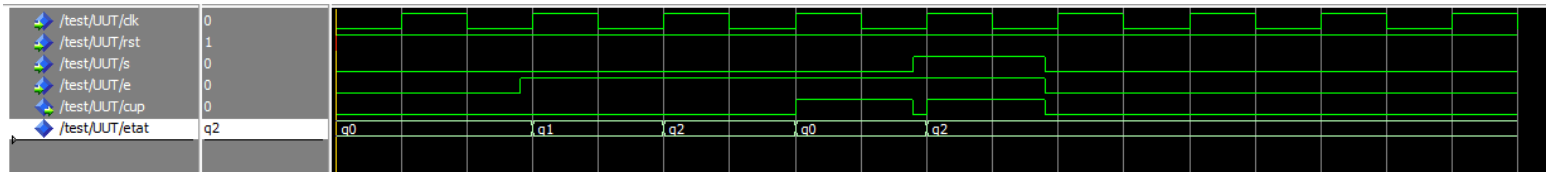
```
57  architecture bench_3 of test is
58
59          component compteur is
60                  port (
61                          s, e, clk, rst :        bit;
62                          cup :   out bit
63                  );
64          end component;
65
66          signal s, e, clk, rst, z :      bit;
67
68  begin
69
70          UUT: compteur port map (s=>s,e=>e,clk=>clk,rst=>rst,cup=>z);
71
72          clk     <=      not clk after 5 ns;
73
74          rst     <=      '1';
75          e       <=      '1' after 16 ns, '0' after 56 ns;
76          s       <=      '1' after 46 ns, '0' after 56 ns;
77
78  end architecture;
```



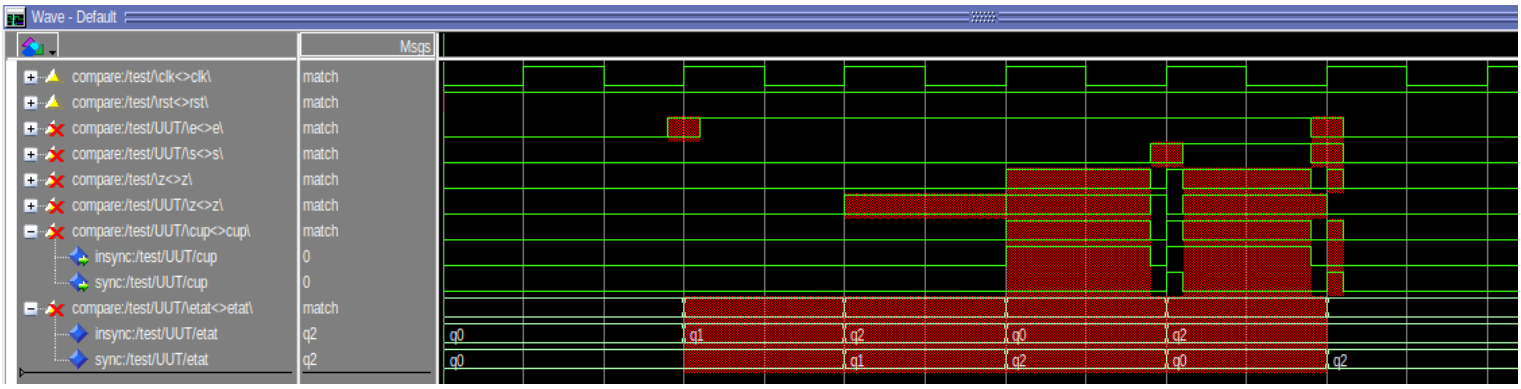
Le résultat est conforme à la description.

SECOND SCENARIO

```
80  architecture bench_4 of test is
81
82          component compteur is
83                  port (
84                          s, e, clk, rst :        bit;
85                          cup :   out bit
86                  );
87          end component;
88
89          signal s, e, clk, rst, z :      bit;
90
91  begin
92          UUT: compteur port map (s=>s,e=>e,clk=>clk,rst=>rst,cup=>z);
93
94          clk     <=      not clk after 5 ns;
95
96          rst     <=      '1';
97          e       <=      '1' after 14 ns, '0' after 54 ns;
98          s       <=      '1' after 44 ns, '0' after 54 ns;
99
100 end architecture;
```



Le résultat est aussi conforme au fonctionnement voulu. Nous allons comparer les deux scénarios avec l'outil de comparaison de *ModelSim*.



On remarque que le scénario synchrone est en retard par rapport au scénario asynchrone. En effet, dans le cas asynchrone, les états internes ont le temps de s'actualiser avant le front d'horloge, alors que dans l'autre cas, il faut attendre le front d'horloge suivant.

CONCLUSION

Nous avons étudié les machines de *Mealy* et *Moore* à travers les exemples d'un contrôleur de mémoire et d'un compteur. Nous avons remarqué que la description du système dans l'architecture ainsi que le timing des entrées influençaient la rapidité du composant. Il serait intéressant de simuler les mêmes éléments en prenant en compte les temps de propagations.