Noé LENOIR
Gaël Ousset

SEI 2A

Compte rendu TP
Modélisation de circuits numériques

Lors de ce TP, nous avons réalisé la conception d'un certain nombre de circuits numériques. Pour cela, nous avons commencé par analyser le cahier des charges expliqué dans l'énoncé, puis nous avons codé en Verilog les portes et opérations nécessaires, pour enfin réaliser des tests et simuler le résultat.

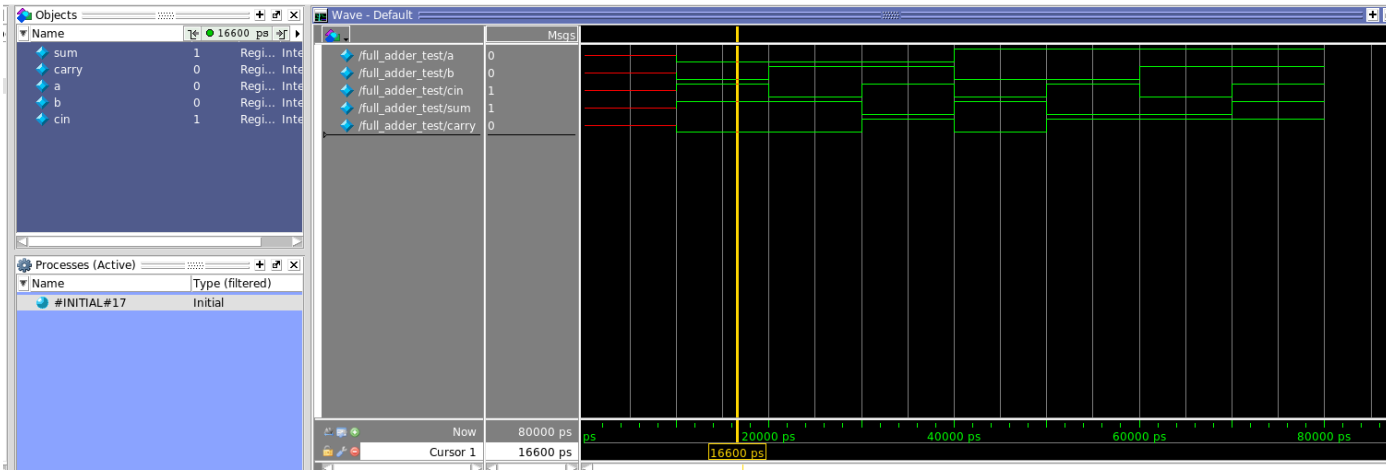
Tout d'abord, nous avons commencé par comprendre et tester le code d'un half adder. Pour cela, nous avons compilé le fichier test donné dans le dossier, et nous avons lancé la simulation sur modelsim. On constate que la simulation est juste pour les différentes valeurs de a, b et que la retenue fonctionne correctement (cf 1) du compte rendu n°1).

Ensuite, nous avons essayé de réfléchir à deux solutions pour coder un full adder. La première à laquelle nous avons pensé et que nous avons codée est une version plus mathématique. Nous avons commencé par définir les entrées et sorties, et le temps d'exécution. Nous n'avons cependant pas gardé la dernière version du code en photo. En effet, il y a une erreur que nous avons corrigé avant la simulation : assign{carry,sum} = a+b+cin;
Nous avons complété le test bench correspondant

L'objectif de ce test bench est donc de tester les différentes possibilités des signaux d'entrées du full adder afin de vérifier graphiquement si la sortie correspond bien aux valeurs attendues.

```
timeunit 1ns;
timeprecision 100ps;


logic    sum ;
logic    carry;
logic    a  ;
logic    b  ;
logic   cin;

// INSTANCE HA
 full_adder HA (.a(a), .b(b), .carry(carry), .sum(sum), .cin(cin));
// Monitor Results

  initial
    begin
        $timeformat ( -9, 1, " ns", 9 );
//$timeformat [ ( units_number , precision_number , suffix_string , minimum_field_width ) ] ;
//Units number suggests the simulation precision time as specified by timescale.
//Precision number is the number of integers displayed after decimal point.\
//Suffix string is the string suffixed at the end of time.\
//Minimum field width suggests the minimum number of characters used to display the time.

        $monitor ( "time=%t a=%b a=%b cin=%cin carry=%b sum=%b", $time, a, b, cin, carry, sum );

 #10 a=1'b0; b=1'b0; cin =1'b1;
 #10 a=1'b0; b=1'b1; cin=1'b0;
 #10 a=1'b0; b=1'b1; cin=1'b1;
 #10 a=1'b1; b=1'b0; cin=1'b0;
 #10 a=1'b1; b=1'b0; cin=1'b1;
 #10 a=1'b1; b=1'b1; cin=1'b0;
 #10 a=1'b1; b=1'b1; cin=1'b1;
 #10
        $stop;
 $display ( "HA TEST TIMEOUT" );
end
endmodule
```

Pour tester notre composant, nous avons dû modifier le fichier de compilation, en précisant quel full adder nous utilisons (cf photo n°3 2) du résumé de la première séance).



On constate que sur notre simulation, les valeurs de sum et carry varient comme il faut. Par exemple lorsque a=0,b=0 cin=1, on a bien sum=1 et carry=0. De même pour a=1,b=1, cin=0 on a sum=0 et carry=1 (cf ci dessus).

On peut comparer avec la simulation du full adder structural avec 2 half adder, on constate que les résultats sont les mêmes (cf résumé 2)).
Ainsi on constate que les 2 solutions sont viables pour simuler un full adder.

De plus, nous nous sommes intéressés au composant MUX. Après avoir lu le code sur 8 bit, nous comprenons que quand l'entrée sel_a vaut 1, le résultat prend en compte la valeur de a, au contraire lorsque sel_a vaut 0 la valeur en sortie est celle de b.
Pour tester si le composant a été bien codé, nous réalisons toutes les possibilités de test. En effet, on réalise toutes les combinaisons de valeur de a, b, et sel_a, et nous vérifions qu'en sortie il s'agit de la bonne valeur :
Extrait du test bench associé au MUX (cf fiche résumé séance 1)

```
  // Apply Stimulus
  initial
    begin
      in_a='0; in_b='0; sel_a=0; #1ns xpect('0);
      in_a='0; in_b='0; sel_a=1; #1ns xpect('0);
      in_a='0; in_b='1; sel_a=0; #1ns xpect('1);
      in_a='0; in_b='1; sel_a=1; #1ns xpect('0);
      in_a='1; in_b='0; sel_a=0; #1ns xpect('0);
      in_a='1; in_b='0; sel_a=1; #1ns xpect('1);
      in_a='1; in_b='1; sel_a=0; #1ns xpect('1);
      in_a='1; in_b='1; sel_a=1; #1ns xpect('1);
      $display ( "MUX TEST PASSED" ) ;
      $finish(0) ;
    end

endmodule
```

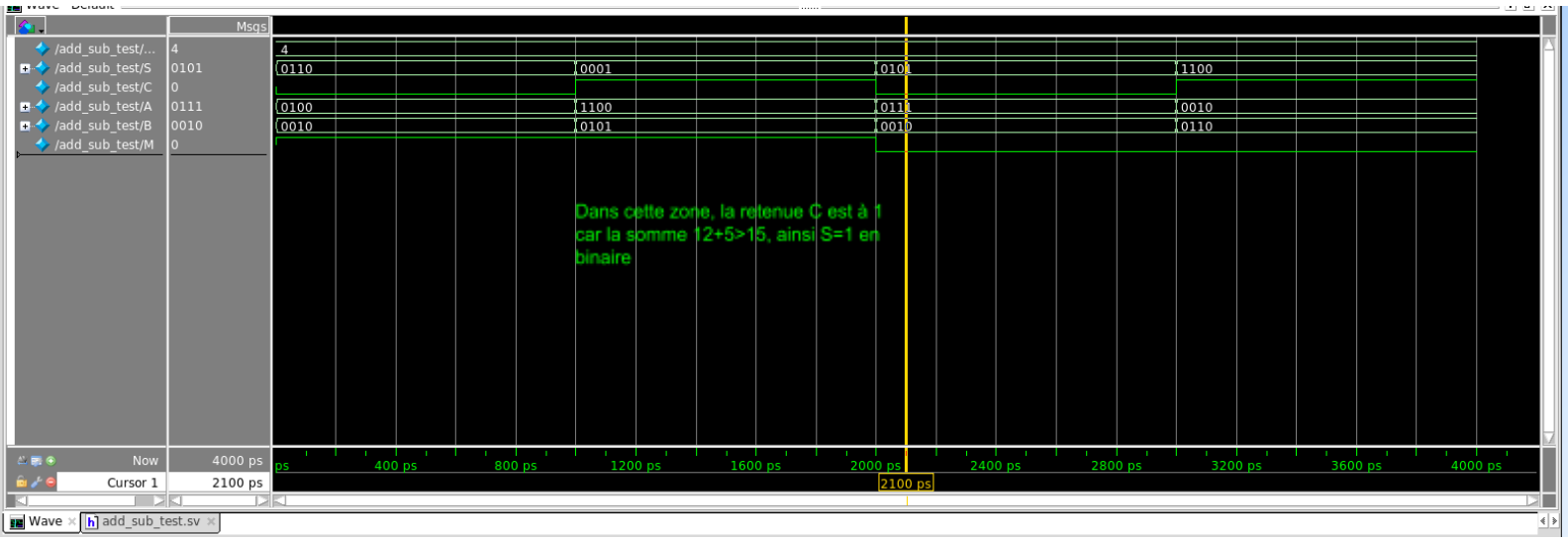
En plus de constater que le test est validé, nous pouvons observer les valeurs sur les Wave. Ainsi, la valeur de sel_a varie périodiquement, et la sortie est bien en fonction de la valeur de sel_a (cf compte rendu 1) dernière image).

Nous avons modélisé une porte adder/soustracteur. Pour cela, nous avons analysé les entrées et les sorties de ce bloc, et nous avons choisi que si sign=1 alors l'opération est une addition, et au contraire si sign=0 alors c'est une soustraction. Nous avons utilisé un always@(*) pour que toutes les valeurs lues déclenchent l'exécution. (cf compte rendu photo °1). Par la suite, nous

réalisons un certain nombre de tests, avec les différentes possibilités. Une somme classique, une somme avec une retenue C, une soustraction classique et une qui retourne un nombre négatif.

```
initial
  begin
    A=4'd4;  B=4'd2; M=1; #1ns xpect(4'd6,  1'd0);
    A=4'd12; B=4'd5; M=1; #1ns xpect(4'd1,  1'd1);
    A=4'd7;  B=4'd2; M=0; #1ns xpect(4'd5,  1'd0);
    A=4'd2;  B=4'd6; M=0; #1ns xpect(4'd12, 1'd1);
    $display ( "MUX TEST PASSED" ) ;
    $finish(0) ;
  end
```

On a l'extrait de code du compte rendu intermédiaire. Pour la seconde ligne, on teste avec des valeurs sur 4 bits en décimal, la somme devrait atteindre 17, ce qui dépasse la capacité maximale, d'où le fait de la retenue et de la valeur 1 en sortie.



En outre, nous avons réalisé un ALU, une fonction arithmétique et logique. En analysant le sujet, on comprend qu'il faut le paramétrer sur 8 bits avec des opérandes synchrones. C'est pour cela que nous utilisons un always @(posedge clk). Nous comprenons qu'il va falloir utiliser les opérateurs pour changer les opérations en fonction de la valeur de la concaténation de carry_out et S.
On utilise un always comb pour donner au flag zéro la valeur 1 si tous les bits de A sont à 0.

```
always @(posedge clk)
  unique case (opcode)
      3'd0:{carry_out,S} = A ^ B;
      3'd1:{carry_out,S} = A + B;
      3'd2:{carry_out,S} = A - B;
      3'd3:{carry_out,S} = A + 1;
      3'd4:{carry_out,S} = A - 1;
      3'd5:{carry_out,S} = A;
      3'd6:{carry_out,S} = A & B;
      3'd7:{carry_out,S} = A | B;
      default:{carry_out,S} = 9'd0;
  endcase

// SystemVerilog: always_comb synthesis intent specification
  always_comb
    zero = ~(|A);
```

Pour le test bench, on a conservé la forme donnée dans l'énoncé, mais nous nous sommes vite rendu compte qu'il fallait modifier les valeurs dans le checkit qui ne correspondaient pas (cf compte rendu intermédiaire n°2 pour voir les nouvelles valeurs).

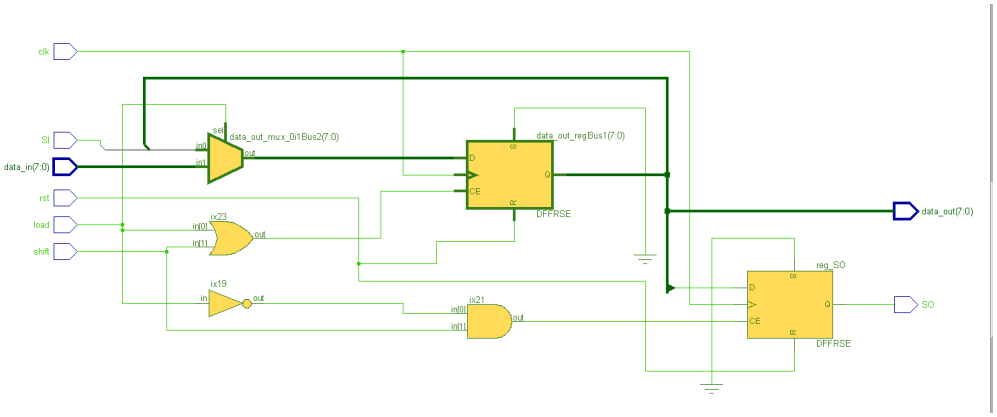
Après avoir lancé notre simulation sur Modelsim et avec ces modifications, on constate que le résultat est satisfaisant et que nous arrivons jusqu'à la fin des tests, avec les valeurs correspondantes (cf dernière image compte rendu intermédiaire n°2)

Ensuite, nous avons modélisé un counteur up/down. Le cahier des charges nous impose un compteur avec un reset asynchrone, et qui suit la période clk. On comprend alors qu'on va utiliser un always_ff @(posedge clk||rst). On commence par tester le cas où rst=1 avec un if pour réinitialiser le compteur. Avec un else if, donc si rst=0, nous écrivons la condition sur load qui permet d'assigner la valeur data dans le count si load=1. Puis enfin, les conditions sur le up et down qui permettent de faire +-1 à la valeur count.
Pour le testbench, nous avons décidé de commencer par vérifier que la valeur est bien recopiée, puis que les cas ou up=0 et up=1, et pour finir que reset=1 permet bien de remettre à 0. On obtient donc le code suivant :

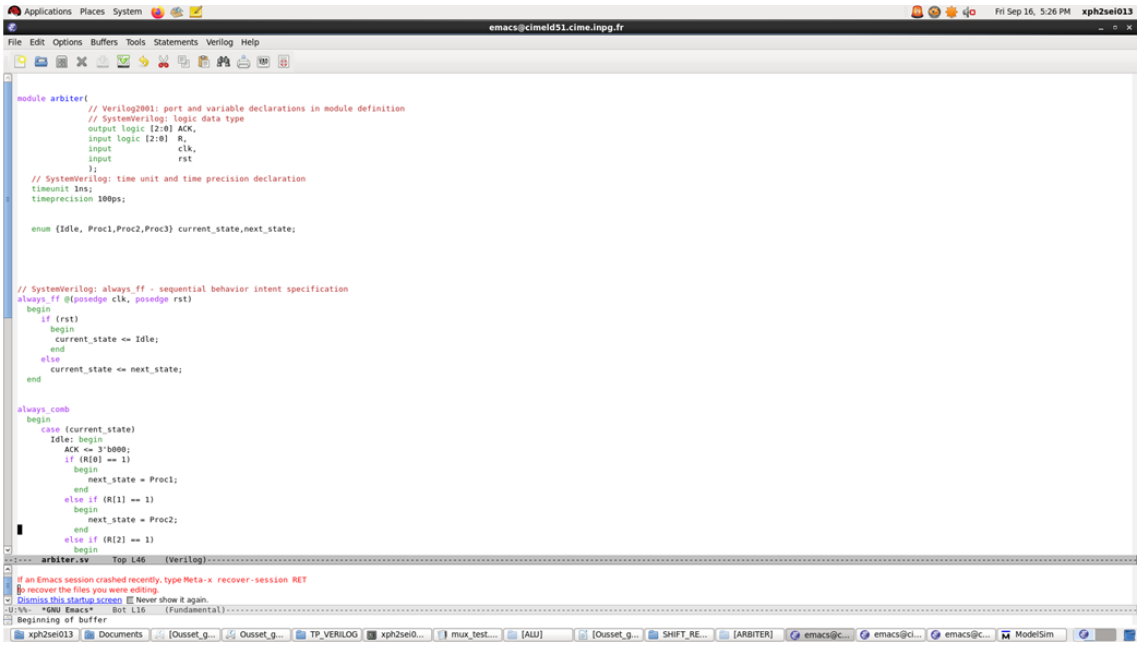
```
initial
        begin
                rst = 0; load = 1; up = 1; data = 5'd45; #10 xpect(5'd45);
                rst = 0; load = 0; up = 1; data = 5'd45; #10 xpect(5'd46);
                rst = 0; load = 0; up = 0; data = 5'd45; #10 xpect(5'd45);
                rst = 1; load = 0; up = 0; data = 5'd45; #5  xpect(5'd0);
                $display ( "COUNTER TEST PASSED" ) ;
                $finish(0) ;
    end
```

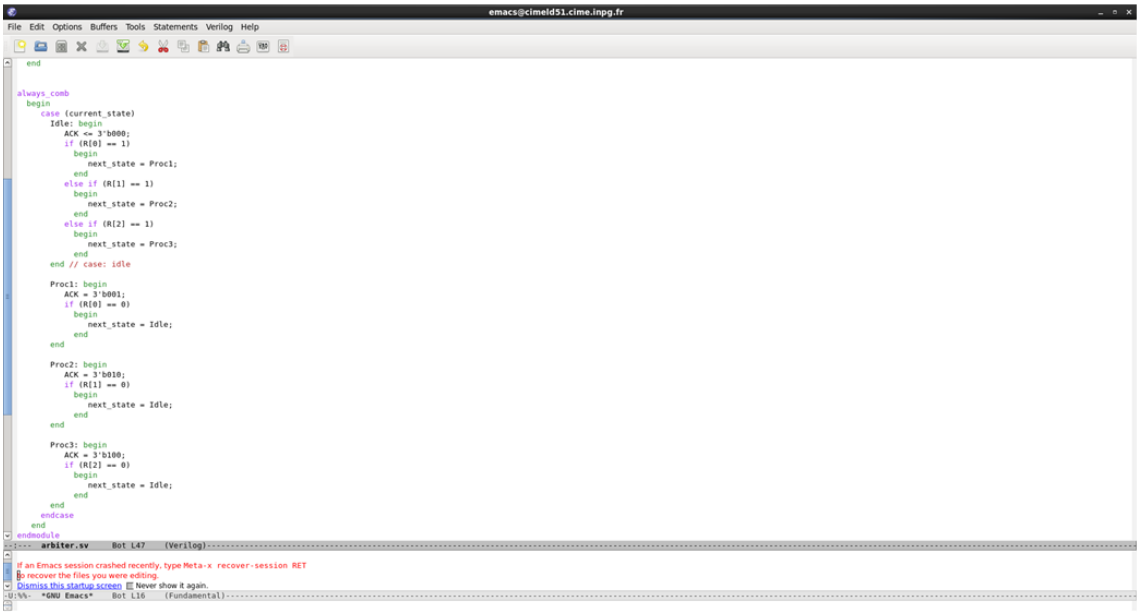
Enfin, comme le montre la dernière photo du 6) du compte rendu intermédiaire, le test est bien passé, et les valeurs sont celles que nous attendions. Notre composant est donc prêt à être utilisé.

Nous avons également modélisé un shift register. Il s'agit d'un composant qui permet de passer des données de simple à parallèle. Avec le cahier des charges, il est indiqué que load doit avoir la priorité sur shift, on comprend donc qu'on va tester le if sur le load avant celui du shift. Si shift=1, on comprend qu'il faut conserver la valeur du bit de poids fort dans S0, décaler data_out vers la gauche, d'où la ligne data_out<=data_out<<1; et mettre la valeur de SI à la place de la nouvelle valeur du bit de poids faible. Voir le screen du compte rendu.
Pour simuler le programme, on a utilisé le testbench fourni. On s'est rendu compte qu'il y avait des erreurs dans les valeurs attendues, nous avons donc recalculé et corrigé ce qu'il fallait. Après avoir lancé la simulation, on se rend compte que le test est bien passé, et que toutes valeurs coincides.
Pour finir avec ce composant, nous avons donc affiché notre composant.



Pour finir, nous avons modélisé un arbiter pour gérer les accès mémoires à différentes unités de calcul. On peut voir ci-dessous les captures d'écrans du code définissant le système. Ce dernier est basé sur une machine d'état développée dans un always comb. La synchronisation avec le signal d'horloge est gérée par un always_ff qui permet d'appliquer les nouveaux états sur le front montant de l'horloge.





Sur les deux avant dernière captures d'écrans du 3e compte rendu nous pouvons voir le test bench. Comme pour les précédents, il utilise la tâche xpect afin de vérifier si la valeur d'ACK correspond bien à la valeur attendue. Ensuite nous appliquons différents signaux de test permettant de tester tout d'abord si le système de priorité et de transition fonctionne puis le dernier test vérifie si le signal de reset fonctionne.
Pour terminer voici ci-dessous notre composant modélisé sous forme de portes logiques.