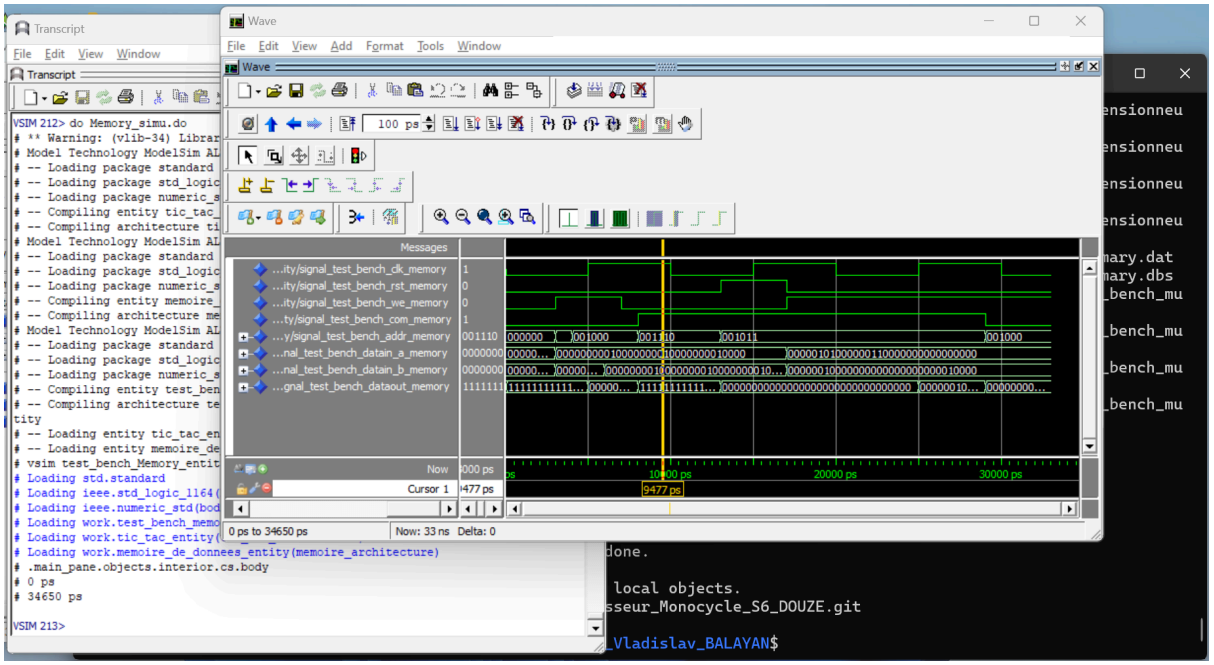
Rapport

Projet d'Électronique Numérique - VHDL Processeur Monocycle



Membre : Vladislav Levovitch BALAYAN
Soutien et aide : Ayoub LADJICI
Aide à la rédaction et aide au débogage : ChatGPT + Monsieur DOUZE Yann

Remerciement

Mes remerciements vont évidemment tout d'abord à mon ami et soutien le plus fidèle, Ayoub LADJICI, qui, malgré les rumeurs, ragots et ses propres difficultés, a su m'épauler tout au long de cette année, m'encourager et me donner tout l'espoir qu'il a pu.

Un très grand merci aussi à Monsieur MARTIN qui m'a beaucoup appris pendant nos séances encadrées depuis le cinquième semestre. Avec son collègue Monsieur PINNA, qui par le cours d'électronique numérique 1, m'ont fait découvrir une nouvelle passion pour ce langage étant l'amour parfait entre l'informatique et l'électronique.

Mes remerciements vont aussi à Monsieur DOUZE qui a toujours fait preuve d'une grande pertinence dans ses réponses même lorsque mes questions étaient des plus basiques. Merci à lui et à notre très bien aimé Thibault (alias Papa des Ei) qui ont beaucoup participé à mon épanouissement personnel et qui m'ont appris à bien structurer mes codes.

Pour finir, j'aimerais remercier toutes ces personnes extraordinaires que j'ai rencontrées cette année dans la formation de mes rêves depuis de nombreuses années, et qui, par leur simple présence, me rendaient heureux : les copains du TPA, Inessa, Nicolas, Maxime, Ayman, Victor, Quentin, Papa, Ilias, Nadir, et ceux du TPF, nos incroyables alternants avec une mention toute particulière pour ces perles qui m'ont accompagné tout au long de l'année dans tous nos différents travaux en traitement du signal : Liza, Jihen, Amel, Salma, Farah et Ioana, les alternants de mon groupe, Asdjad, Fatou, Enes, Hakan, Vedat, Abde, Richard, Amine, les deux Tarek, Michel, Emmanuel, Théo, et les alternants qu'on ne pouvait voir que trop rarement grâce aux restructurations mais qui ont tout de même su rayonner : Arthur, Léa, Gloire A Dieu, Gabriel, Rémy, Ali, Elsa, Karlitou, Sékouba, Tom et pour finir Mon Pitit Chou.

Introduction

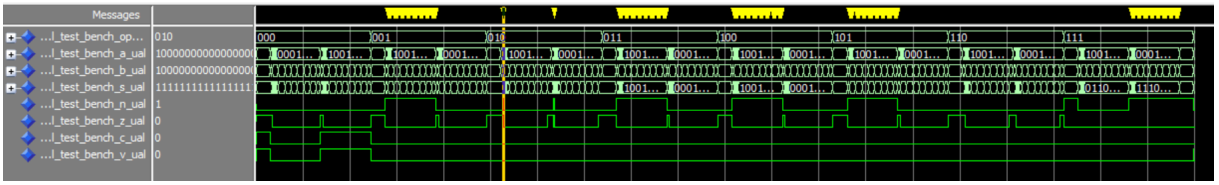
Dans le cadre de notre parcours universitaire en 3ème année d'école, nous sommes amenés, en Electronique Numérique 3, à réaliser et simuler un cœur de processeur mono-cycle. Pour une meilleure gestion de notre projet, nous avons, sur les conseils de Thibault, codé dans un environnement de travail pertinent avec l'IDE VScode, un suivi avec GitHub et une remise en ligne via Google Doc.

Ce projet ambitieux implique la conception et la simulation de différents blocs principaux tels que l'unité de traitement, l'unité de gestion des instructions et l'unité de contrôle. L'objectif est de diviser le projet en petits composants tels que des multiplexeurs, des registres, des bancs de mémoire, ainsi qu'une Unité Arithmétique et Logique (UAL). Cela nous permettra de simuler, d'assembler et de tester notre modèle sur une carte FPGA.

Partie 1 : Unite de traitement

1.1 Unité Arithmétique et Logique

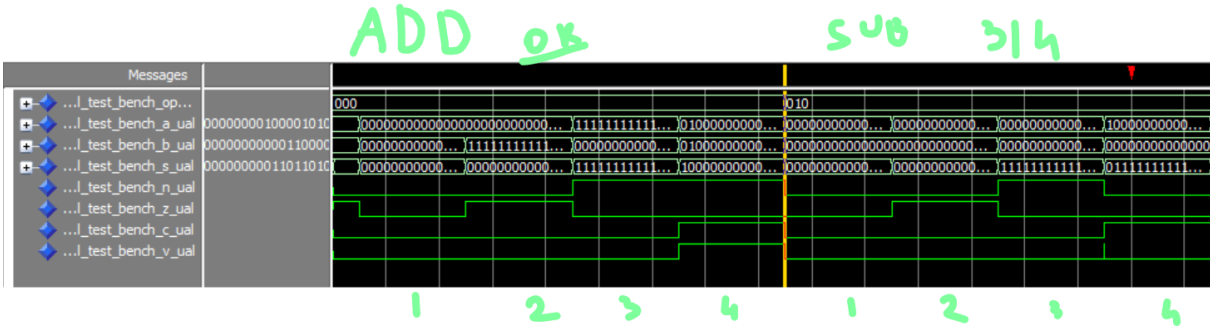
Figure 1.1.1 : Chronogramme de l'UAL avec 2 boucles for testant toutes les opérations de l'UAL.



Nous remarquons bien que les drapeaux nuls et négatifs visibles tout au long du test bench sont cohérents avec les parties où les résultats sont négatifs ou nuls. Par ailleurs, on remarque que les seuls cas de débordement et de retenue se produisent au niveau de la zone d'addition.

Dans un second temps, le test ne vérifiant pas la zone de soustraction ici, nous allons la tester pour vérifier sa validité plus en détail.

Figure 1.1.2 : Chronogramme de l'UAL avec des valeurs précises pour les opérations ADD et SUB.



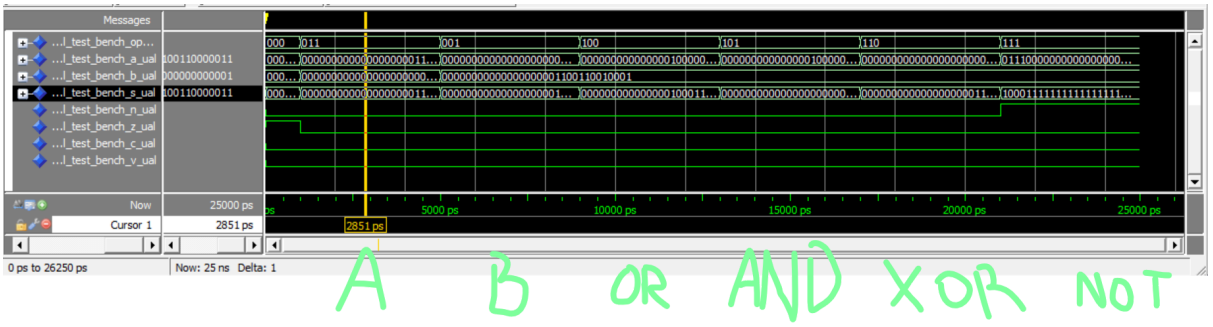
Dans cette sous-partie, nous avons testé en détail les opérations ADD et SUB dans un deuxième banc de test. Tout d'abord avec un test d'opération simple en zone 1, puis un test de nullité cherchant à lever seulement le drapeau Z en zone 2, ensuite un test de négativité cherchant à lever seulement le drapeau N en zone 3, et enfin un test en zone 4 pour vérifier les drapeaux de débordement et de retenue. Tous nos tests ont validé nos assertions de résultats et de drapeaux, hormis le test de retenue en zone 4 pour l'opération de soustraction.

Figure 1.1.3 : Extrait de code montrant la partie posant problème au niveau de la retenue.

```
129     -- Test Sub soustraction avec Debordement_Retenue
130     SIGNAL_Test_Bench_A_UAL <= x"8000_0000";
131     SIGNAL_Test_Bench_B_UAL <= x"0000_0001";
132     wait for 1 ns;
133
134     assert (SIGNAL_Test_Bench_S_UAL = x"7FFF_FFFF") report "Test SUB Debordement : Erreur resutat" severity error;
135     assert (SIGNAL_Test_Bench_Z_UAL = '0') report "Test SUB Debordement : Z_UAL incorrect" severity error;
136     assert (SIGNAL_Test_Bench_N_UAL = '0') report "Test SUB Debordement : N_UAL incorrect" severity error;
137     assert (SIGNAL_Test_Bench_C_UAL = '1') report "Test SUB Debordement : C_UAL incorrect" severity error;
138     assert (SIGNAL_Test_Bench_V_UAL = '1') report "Test SUB Debordement : V_UAL incorrect" severity error;
139     wait for 3 ns;
```

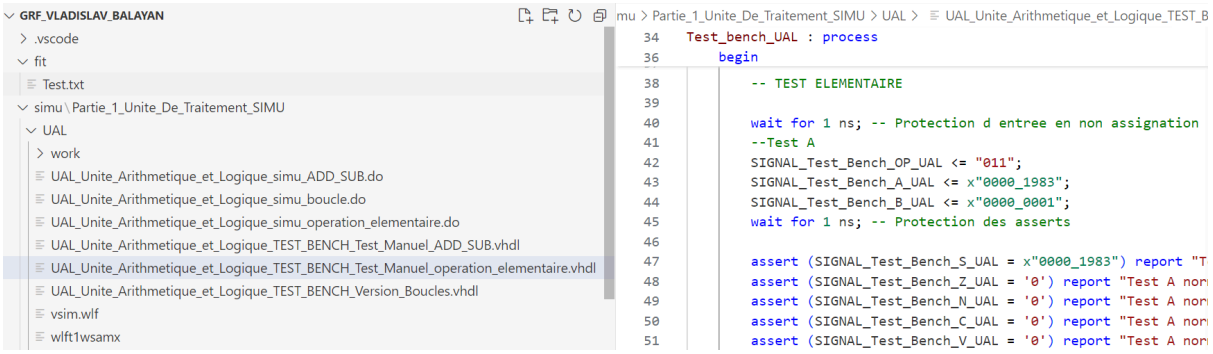
Malheureusement, après plusieurs essais et vérifications, nous avons effectué de multiples corrections et modifications, retrouvables dans l'annexe 1.1 Debogage UAL. Cependant, nous avons rencontré un problème persistant avec la retenue négative que nous n'avons pas pu résoudre malgré un long acharnement, hormis des résultats pour la soustraction tout de même corrects. Nous avons cherché aide et conseil, mais résoudre cela aurait nécessité de reprendre entièrement la structure d'un collègue, ce qui aurait perdu de son intérêt. Pour cette raison, nous allons continuer le projet en sachant que la retenue négative reste ambiguë et non résolue.

Figure 1.1.4 : Chronogramme de l'UAL avec des valeurs précises pour les opérations A, B, OR, AND, XOR et NOT.



Nous avons réitéré les tests dans un troisième banc de test pour vérifier les opérateurs A, B, OR, AND, XOR et NOT. Ceux-ci n'ont levé aucun rapport d'erreur et se sont donc avérés concluants.

Figure 1.1.5 : Extrait de code montrant en particularité la partie A.



Voici un extrait de code pour la Figure 1.1.4 afin de vérifier en détail que nos valeurs de sortie sont correctes de manières plus lisible que directement sur le chronogramme.

Ici, nous obtenons en sortie 1983 au lieu de 0001 ; aucun des drapeaux ne doit être levé et c'est bien le cas. La vérification est bonne.

1.2 Banc de Registres

Tout d'abord, nous avons pensé à ajouter un nouveau module s'occupant exclusivement de l'horloge pour une meilleure compartimentation des fonctions des modules. Celui-ci a été directement testé en condition d'utilisation normale, ici dans un banc de test. À terme, il servira à toutes les fonctions et modules synchrones.

Figure 1.2.0 : Chronogramme de l'horloge générée depuis un module extérieur.

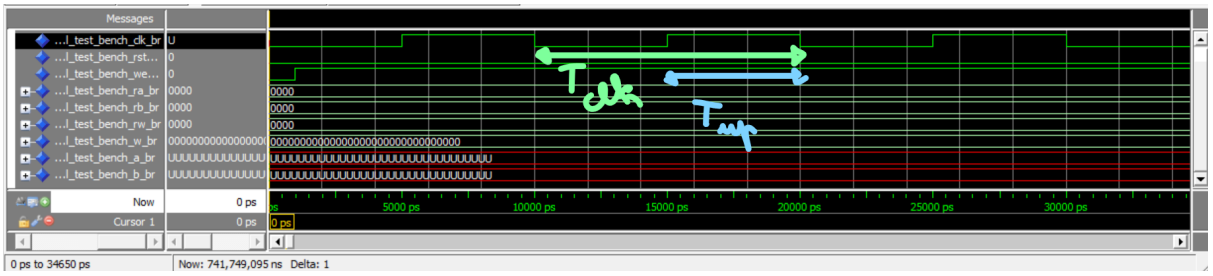


Figure 1.2.1 : Chronogramme du Banc de Registres indiquant la mise à jour des registres en fonction de l'adresse donnée et demandée en lecture ou écriture.

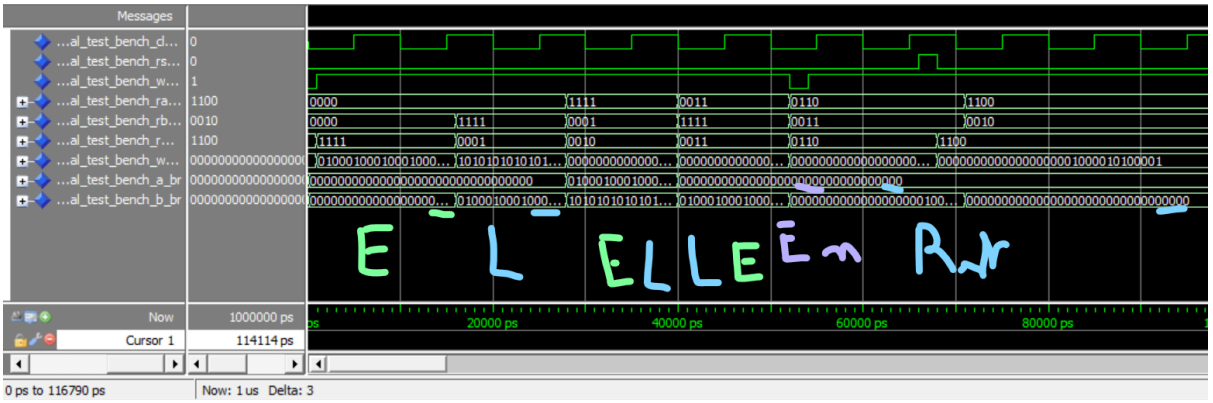
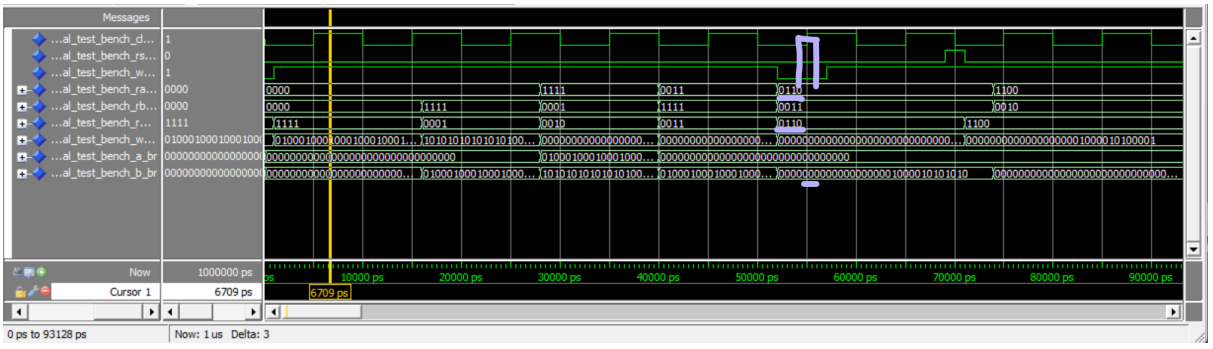


Figure 1.2.2 : Chronogramme du Banc de Registres indiquant le fonctionnement de l'enable.



Nous avons testé l'écriture simple, la lecture simple, l'écriture simultanée, l'écriture avant lecture, l'écriture après lecture, le reset et l'enable. Tous les tests avec assert ont été

réussis. Nous constatons une remise à zéro après le reset et que les sorties A et B sont correctement assignées en fonction des adresses de registres demandées. Enfin, nous constatons de manière plus claire sur la Figure 1.2.2 que lorsque l'écriture est désactivée, bien que les registres soient identiques entre celui en écriture et en lecture, au coup d'horloge les valeurs A et B ne sont pas mises à jour.

Figure 1.2.3 : Extrait de code montrant les modifications à apporter pour avoir la convention tableau de vecteurs, bien que celle-ci soit plus opaque.

```
case Ra_BR is
    when "0000" => A_BR  <= Registre_0_RB;  -- Registre 0
    when "0001" => A_BR  <= Registre_1_RB;  -- Registre 1
    when "0010" => A_BR  <= Registre_2_RB;  -- Registre 2
```

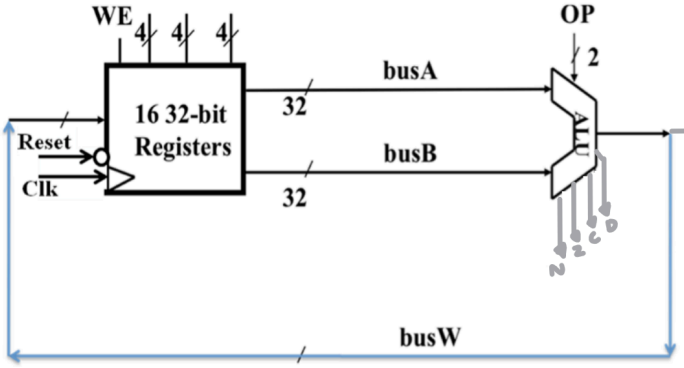
```
--type t_Registre is array (0 to 15) of std_logic_vector(31 downto 0); -- Ca existe mais je ne suis pa
--signal Registres_RB : t_Registre := (others => (others => '0'));
-- Pour le registre je ne suis pas fan de mettre un tableau de 32 bits je prefere les avoir en seul me

signal Registre_0_RB, Registre_1_RB, Registre_2_RB ,Registre_3_RB ,Registre_4_RB ,Registre_5_RB ,Regis
-- La version avec le banc consisterai a
-- remplacer les lignes when "0000" => A_BR  <= Registre_0_RB;  -- Registre 0 par
-- when "0000" => A_BR  <= Banc (0);  -- Registre 0 et ainsi de suite et la ligne pour l ecriture :
-- when "0000" => Registre_0_RB <= W_BR;  -- Registre 0
-- when "0000" => Banc(0) <= W_BR;

-- Content
```

Par ailleurs, nous avons délibérément choisi d'utiliser des vecteurs distincts pour chaque registre plutôt qu'un tableau de vecteurs pour des raisons de lisibilité et de facilité de débogage. Cette approche permet d'identifier et de tester chaque registre individuellement, améliorant la compréhension du code et de garder une cohérence avec le reste du code utilisant des vecteur et non des tableaux de vecteurs. Bien que l'utilisation d'un tableau soit une méthode valide, nous avons préféré une structure plus simple et claire pour gérer les registres.

1.3 UAL et Banc de Registres assemblés

Figure 1.3.1 : Schéma d'ensemble de l'UAL et du banc de registres (à gauche).

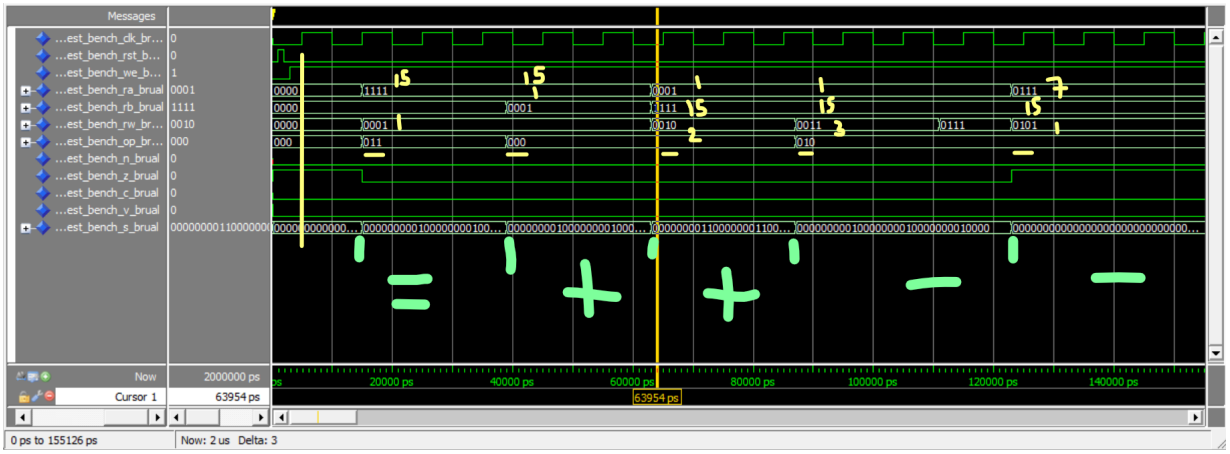
Figure 1.3.2 : Opérations effectuées avec les valeurs de registres de données utilisées et le résultat attendu (à droite).



x"0040_2010" = R(15)
x"0040_2010" - R(1) = R(15)
x"0080_4020" - R(1) = R(1) + R(15)
x"00C0_6030" - R(2) = R(1) + R(15)
x"0040_2010" - R(3) = R(1) – R(15)
x"0000_0000" - R(5) = R(7) – R(15)

Après avoir assemblé et débogué longuement les modules (1.2 Debogage Assemblage UAL - Banc de Registres) l'unité arithmétique et logique (UAL) avec le banc de registres comme indiqué sur la Figure 1.3.1, nous avons réalisé le banc de test des opérations élémentaires de la Figure 1.3.2. Avec des valeurs suffisamment simple pour être plus aisèment repérable. Les valeurs choisies sont suffisamment simples pour être plus aisément repérables. Par ailleurs, R7 est égal à R15 afin de vérifier également sur cette simulation la validité des drapeaux, déjà testée par les précédents bancs de test.

Figure 1.3.3 : Chronogramme de l'assemblage Banc de Registres et UAL indiquant les 5 équations demandées, les bus d'adresses des équations et les résultats photocopiés dans la variable S.



Suite au reset au démarrage et à l'activation de l'écriture, nous remarquons aisément les 5 zones de calcul avec les valeurs de OP, la donnée de contrôle de l'UAL : d'abord l'égalité, puis les deux sommes, et enfin les deux différences. Nous remarquons que les bus d'adresses de recopie RW et de lecture RA et RB sont parfaitement cohérents.

Figure 1.3.4 : Extrait de code montrant un assert d'addition et un de soustraction utilisés avec les valeurs précédemment calculées.

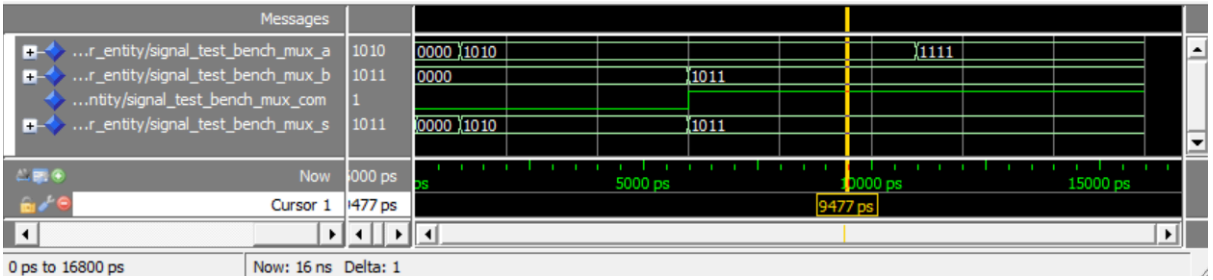
```
-- Verification R2 +
assert SIGNAL_Test_Bench_S_BRUAL = x"00C0_6030" report "Test +2 incorrect" severity failure;
wait for 12 ns;

-- R3 = R1 - R15
SIGNAL_Test_Bench_Ra_BRUAL <= "0001"; -- R1
SIGNAL_Test_Bench_Rb_BRUAL <= "1111"; -- R15
SIGNAL_Test_Bench_OP_BRUAL <= "010"; -- S = A - B
SIGNAL_Test_Bench_Rw_BRUAL <= "0011"; -- Ecriture dans R3
wait for 12 ns;

-- Verification R3 -
assert SIGNAL_Test_Bench_S_BRUAL = x"0040_2010" report "Test -3 incorrect" severity failure;
wait for 12 ns;

-- R5 = R7 - R15
SIGNAL_Test_Bench_Rw_BRUAL <= "0111"; -- Ecriture dans R7
wait for 12 ns;
```

Enfin, pour vérifier les valeurs les plus importantes, par l'intermédiaire de la variable bonus S, qui recopie les valeurs intermédiaires de retour à l'écriture de la sortie de l'UAL, nous avons repris les calculs précédemment énoncés en Figure 1.3.2. La simulation s'étant déroulée jusqu'à la fin sans s'interrompre à cause d'un échec de résultat, nous concluons à l'exactitude de notre assemblage.

1.4 Composants utiles à l'assemblage d'une unité de traitement

1.4.1 Multiplexeur 2 vers 1 à N bits

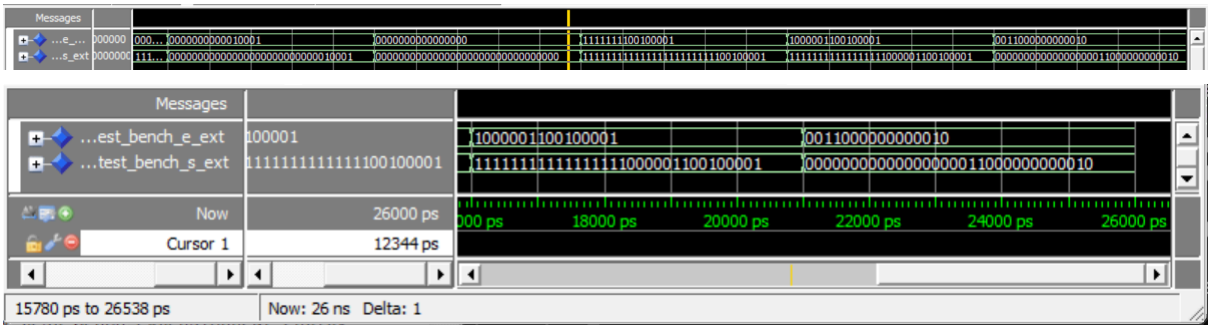
Figure 1.4.1.1 : Chronogrammes du multiplexeur 2 vers 1



Nous venons de coder et simuler l'un des composants les plus élémentaires en numérique, après la bascule et les opérateurs logiques simples. Ainsi, nous pouvons voir que dans le 1er tiers, il y a une première assignation à A, puis à B, suivie d'une modification de A sans changement sur la sortie, la donnée de commande n'ayant pas été modifiée. Ainsi, nous validons notre premier composant utile à l'élaboration de l'assemblage de l'unité de traitement.

1.4.2 Extension de signe de N à 32 bits

Figure 1.4.2.1 : Chronogrammes du module nécessaire à l'extension de signe.



Pour vérifier la validité du module, nous avons effectué 5 tests simples : 1 vérification de l'extension simple avec un vecteur positif, 2 un vecteur nul, 3 un vecteur négatif suivant un vecteur à zéro, 4 un vecteur négatif à la suite d'un autre de même signe et enfin 5 un vecteur de signe différent, donc positif, à la suite d'un vecteur négatif. Les 5 asserts se sont révélés concluants, le module étend bien les 5 nombres signés par des 0 ou des 1. Le sous-module est donc validé.

Figure 1.4.2.2 : Extrait de code montrant un des cinq asserts validés.

```
-- Verification negatif a la suite
SIGNAL_Test_Bench_E_Ext <= x"8321";
wait for 1 ns;
assert SIGNAL_Test_Bench_S_Ext = x"FFFF_8321" report "Test - incorrect" severity failure;
wait for 4 ns;
```

1.4.3 Mémoire de données 64x32 bits

Figure 1.4.3.1 : Extrait de code indiquant la nouvelle gestion de données par l'utilisation d'un tableau, structure plus opaque.

```
architecture Memoire_architecture of Memoire_de_donnees_entity is
-- Signal
    type mem is array(0 to 63) of std_logic_vector(31 downto 0);
    signal Memory : mem := (others => (others => '1')); -- La memoire etant plus opaque que le Banc de Registre
    -- Le tableau a le desavantage d etre plus opaque comme strucuture de donnee mais c est plus coherent avec l
```

Pour ce troisième sous-élément additionnel, nous avons cette fois directement opté pour l'utilisation d'un tableau de vecteurs pour stocker les mots. En effet, partant de la considération que la mémoire est un module assez obscur, une représentation du même niveau d'opacité semblait être le choix le plus judicieux. Nous avons donc réadapté la structure du banc de registres afin que celui-ci puisse gérer en plus un multiplexage interne et réorganisé le code afin qu'il soit plus clair malgré son opacité.

Figure 1.4.3.2 : Extrait de code montrant l'exploitation de la structure de tableau plus lisible.

```
27  begin
28  --Memory_Reading_Process : process (Addr_Memory)
29  --    begin
30      DataOut_Memory <= Memory(to_integer(unsigned(Addr_Memory)));
31  --    end process Memory_Reading_Process;
32
33  Memory_Writing_Process : process (Clk_Memory,Rst_Memory) -- Reset asynchrone
34      begin
35          if Rst_Memory = '0' then
36              if rising_edge(Clk_Memory) then
37                  if WE_Memory = '1' then -- Sucre syntaxique pour la lisibilite
38                      -- Local Mux
39                      case COM_Memory is
40                          when '0' => Memory(to_integer(unsigned(Addr_Memory))) <= DataIn_A_Memory;
41                          when '1' => Memory(to_integer(unsigned(Addr_Memory))) <= DataIn_B_Memory;
42                          when others => null; -- Par securite meme si ne devrait pas arriver
43                      end case;
44                  end if;
45              end if;
46          elsif Rst_Memory = '1' then
47              -- Redemarrage de la memoire
48              for i in 64-1 downto 0 loop
49                  Memory(i) <= (others => '0');
50              end loop;
51          end if;
52      end process Memory_Writing_Process;
53  end Memoire_architecture;
54
```

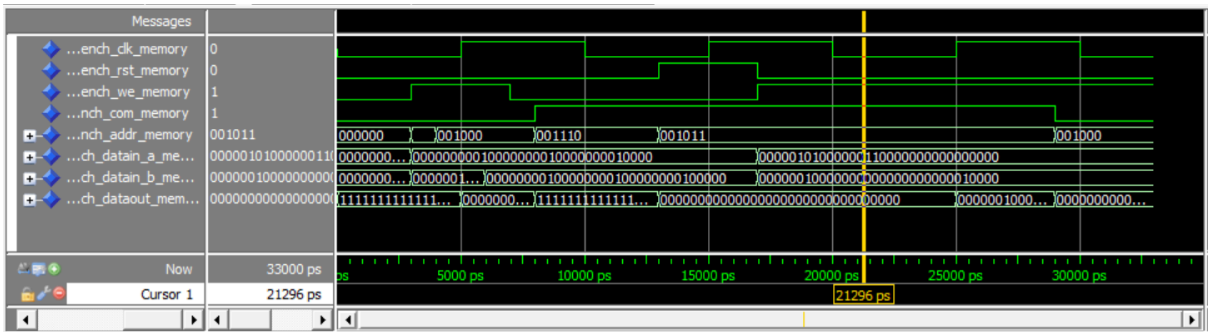
Pour mieux détecter dans le projet si le reset a été démarré, nous avons mis comme valeurs d'initialisation tous les bits à la valeur numérique haute '1'. Ainsi, lorsque l'on enclenche le reset, les valeurs deviennent très distinctement nulles et repérable par la suite sur le chronogramme.

Figure 1.4.3.3 : Extrait de code présentant les asserts vérifiant la bonne remise à zéro, le stockage de valeurs après une réinitialisation et la conservation des précédentes valeurs après changement d'adresse de registre.

```
-- Test reset
assert SIGNAL_Test_Bench_DataOut_Memory = x"FFFF_FFFF" report "Test valeurs blanches" severity failure;
SIGNAL_Test_Bench_Rst_Memory <= '1';
wait for 4 ns;
SIGNAL_Test_Bench_Rst_Memory <= '0';
SIGNAL_Test_Bench_WE_Memory <= '1';

-- Test post reset et initialialisation des valeurs a 0
SIGNAL_Test_Bench_Addr_Memory <= "001011"; -- Adresse 11
SIGNAL_Test_Bench_DataIn_B_Memory <= x"0200_0010"; -- Current selection
SIGNAL_Test_Bench_DataIn_A_Memory <= x"0503_0000";
assert SIGNAL_Test_Bench_DataOut_Memory = x"0000_0000" report "Test initialisation avant ecriture" sever
wait for 12 ns;

-- Test changement registre et conservation memoire
assert SIGNAL_Test_Bench_DataOut_Memory = x"0200_0010" report "Test lecture post reset" severity failure
SIGNAL_Test_Bench_COM_Memory <= '0'; -- Selection valeurs de A
SIGNAL_Test_Bench_Addr_Memory <= "001000"; -- Adresse 8 nouvellement pour A
wait for 11 ns;
assert SIGNAL_Test_Bench_DataOut_Memory = x"0503_0000" report "Test lecture memoir" severity failure;
SIGNAL_Test_Bench_Addr_Memory <= "001011"; -- Adresse 11 precedemment ecrite par B
assert SIGNAL_Test_Bench_DataOut_Memory = x"0200_0010" report "Test lecture post reset" severity failure
```

Après de multiples tests, notamment lors de la lourde phase de 1.3 Debogage Memoire de données, nous présentons ici les tests les plus pertinents permettant une validation très complète du module.

Figure 1.4.3.4 : Chronogramme de la simulation de la gestion du module de mémoire de données.



Vis-à-vis des contraintes demandées sur la synchronisation ou non, nous constatons au début de la simulation une écriture bien synchrone avec le front montant de l'horloge, puis, lors du changement de registre de manière asynchrone, une bonne mise à jour des valeurs vers un registre non assigné ayant encore les valeurs à blanc, ici initialisée avec tous ses bits à '1'. Nous remarquons une remise à zéro, aussi asynchrone et fonctionnelle, permise grâce encore une fois aux tests du débogage 1.3 Debogage Memoire de données. Le test, désactivant l'écriture en changeant l'adresse du registre, a en effet empêché la modification de la sortie au deuxième coup d'horloge. Enfin, nous avons vérifié par les tests visibles sur la figure Figure 1.4.3.3 la gestion correcte du changement de bus d'entrée en écriture par notre multiplexeur interne, ainsi que la tenue en mémoire de la précédente

valeur après une nouvelle écriture sur un registre différent. Toutes ces informations, ainsi que les validations de nos asserts sans interruption d'exécution, confirment notre code.

Conclusion

Bien que le projet n'ait pas été achevé et ait connu un démarrage sur les chapeaux de roues, avec un très long blocage au niveau de l'UAL et des problèmes personnels, celui-ci m'aura beaucoup apporté. J'ai finalement trouvé une forme de libération et surtout une satisfaction personnelle qui me donne encore plus de motivation pour la suite de mon parcours universitaire et humain. Ce projet m'a permis de mieux comprendre la gestion des données dans le processeur, leur déplacement via les bus de données, ainsi que la signification concrète des signaux sortant des composants. J'ai également beaucoup apprécié en savoir plus sur les composants que l'on rencontre souvent sur les schémas électroniques de nos documentations techniques et mieux les comprendre.

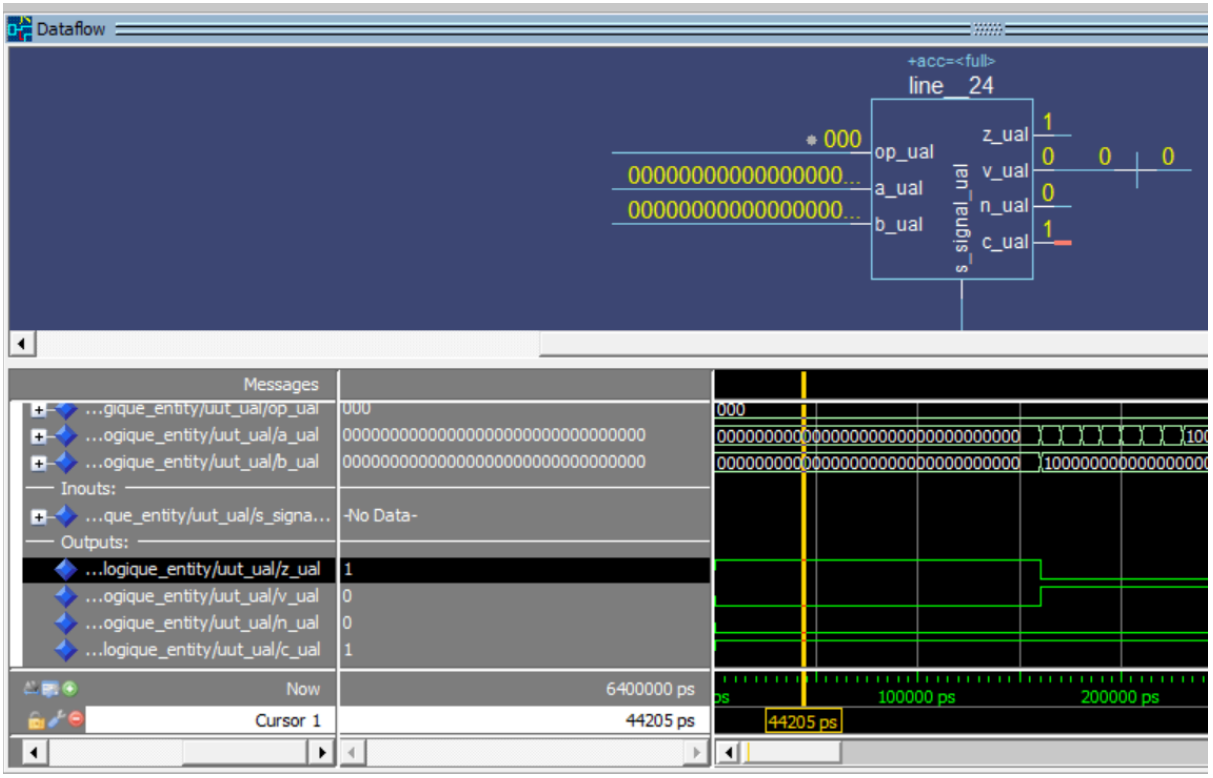
Bien que certains n'aient pas cru en moi pendant longtemps, je reste convaincu que cela était principalement dû à un manque de communication. J'ai tenté de remédier à cela trop tardivement, je peux l'admettre, notamment en exposant mes difficultés par e-mail en raison de l'absence au niveau du stage. En outre si vous souhaitez en savoir davantage, ce dernier matin, j'avais un rendez-vous en matinée voir à l'annexe 12. DOUZE, ce qui explique pourquoi je n'étais pas présent. Par ailleurs, j'avais besoin d'une bonne préparation nonobstant mon innocence avérée ou non, ce qui, comme nous l'avons vu, s'est révélé insuffisant en raison du peu de temps accordée à celle ci.

Pour conclure ce rapport, j'espère sincèrement, Monsieur DOUZE, que nos relations seront réellement apaisées à l'avenir, malgré vos récentes et très graves accusations vis-à-vis des PC, accusations qui m'ont profondément touché. J'espère également que notre belle filière Ei, EiSE, Ei2I, EiLI, EiST continuera de prospérer comme elle l'a toujours fait, grâce à Thibault HILAIRE, Dimitri GALAYKO, Yann DOUZE, Benoît FABRE, Annick ALEXANDRE, et enfin Michel REDON, et que toutes les futures promotions s'y sentiront aussi épanouies que j'aurais pu l'être, tout comme l'a été notre grand frère Monsieur VIATEUR.

Annexe de débogage

1.1 Debogage UAL

Nous venons de constater par la simulation que pour l'opération ADD, 0 + 0 a bien une comme drapeaux 1 pour zéro mais a aussi 1 pour la retenenu ce qui n'est pas correct. En effet on remarque que la retenue est activée presque par défaut en permanence.



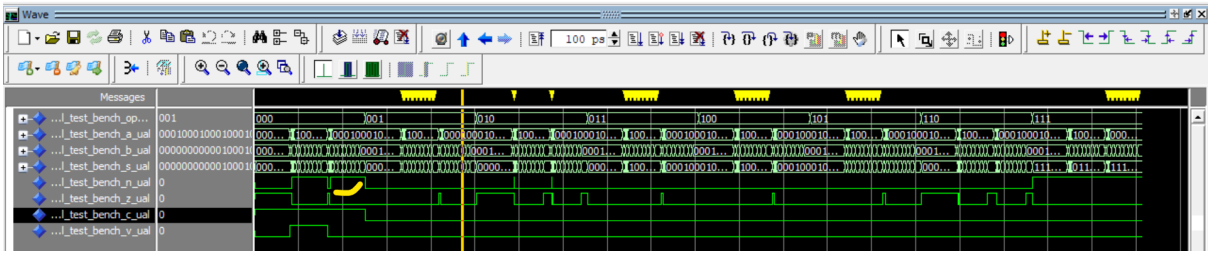

```
60   -- Retenu
61       if (((OP_UAL = "000") and (A_UAL(31) = B_UAL(31))) or ((OP_UAL = "010") and (A_UAL(31) /= B_UAL(31))))  then
62           if ((std_logic_vector (signed(A_UAL)+signed(B_UAL)) > x"7FFF_FFFF") or (std_logic_vector (signed(A_UAL)+signed(B_UAL)) < x"8000_000
63               C_UAL <= '1';
64           end if;
65       end if;
66
67       if (((OP_UAL = "010")) and (A_UAL(31) /= B_UAL(31))) and (A_UAL(31) = B_UAL(31)) then
68           if ((std_logic_vector (signed(A_UAL)+signed(B_UAL)) > x"7FFF_FFFF") or (std_logic_vector (signed(A_UAL)+signed(B_UAL)) < x"8000_000
69               C_UAL <= '1';
70           end if;
71       end if;
```

Entre temps nous avons essayé une autre approche du test d'où le nouveau paterne mais nous avons aussi remarqué que le drapeaux negatif est nous avons eu plusieurs version de la retenu :

```
-- Retenu Bis
-- if (((OP_UAL = "000") and (A_UAL(31) = B_UAL(31))) or ((OP_UAL =
"010") and (A_UAL(31) /= B_UAL(31))))  then
--         if (((std_logic_vector (signed(A_UAL)+signed(B_UAL)) >
x"7FFF_FFFF") and (B_UAL(31) = 0)) or (std_logic_vector
```

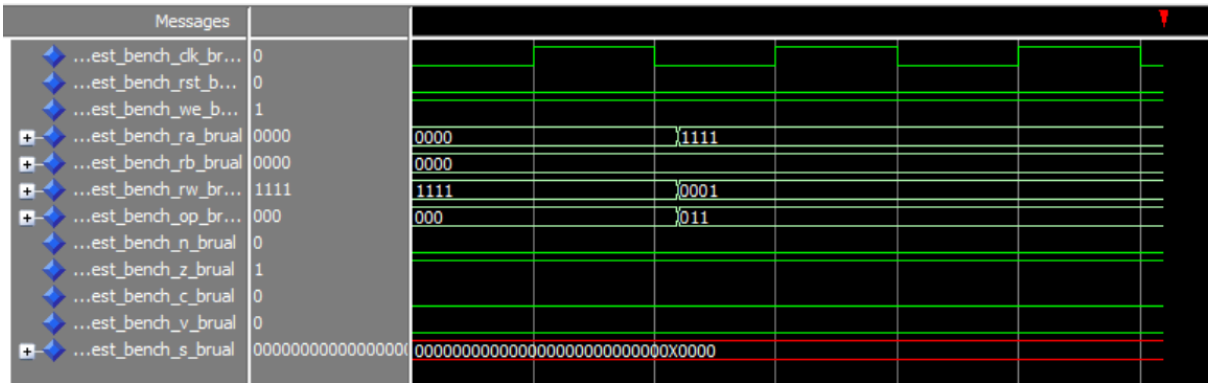
```
(signed(A_UAL)+signed(B_UAL)) < x"8000_0000")) then -- On verifie si la
valeur entiere de A + B depasse le max ou min
--            C_UAL <= '1';
--        end if;
--    end if;
--
--    if (((OP_UAL = "010")) and (A_UAL(31) /= B_UAL(31))) and
(A_UAL(31) = B_UAL(31)) then
--        if ((std_logic_vector (signed(A_UAL)+signed(B_UAL)) >
x"7FFF_FFFF") or (std_logic_vector (signed(A_UAL)+signed(B_UAL)) <
x"8000_0000")) then -- On verifie si la valeur entiere de A - B depasse
le max ou min
--            C_UAL <= '1';
--        end if;
--    end if;

-- Retenu Tris

--    if ((OP_UAL = "000"

--   if ((OP_UAL = "000" or OP_UAL = "010") and ((A_UAL(30) =
B_UAL(30)) ) -- On verfie le bit de plus gros poids

--    C_UAL <= '1';

--   end if;

-- Retenu Tetra essaie
    if (((OP_UAL = "000") and (A_UAL(31) = B_UAL(31) and
B_UAL(31)/=B_UAL(30) and B_UAL(30) = A_UAL(30)))) then -- or ((OP_UAL =
"010") and (A_UAL(31) /= B_UAL(31) and A_UAL(31)/=A_UAL(30) and
B_UAL(30) /= A_UAL(30))))  then
        C_UAL <= '1';
    end if;

-- Retenu Pinta essaie
--    if (OP_UAL = "010") and ((signed(A_UAL) - signed(B_UAL)) <=
to_signed(2**31-1,32) and (signed(A_UAL) - signed(B_UAL)) >=
to_signed(-2**31, 32))
--        C_UAL <= '1';
--    end if;
```

```
-- Retenu soustraction
    if ((OP_UAL = "010") and (A_UAL(31) /= B_UAL(31)) and
(S_SIGNAL_UAL(31) /= A_UAL(31))) then
        V_UAL <= '1';
    end if;
```



1.2 Debogage Assemblage UAL - Banc de Registres

Nous ne parvenions pas à obtenir de résultat probant après plusieurs heures nos valeurs étaient toujours incomplètes :



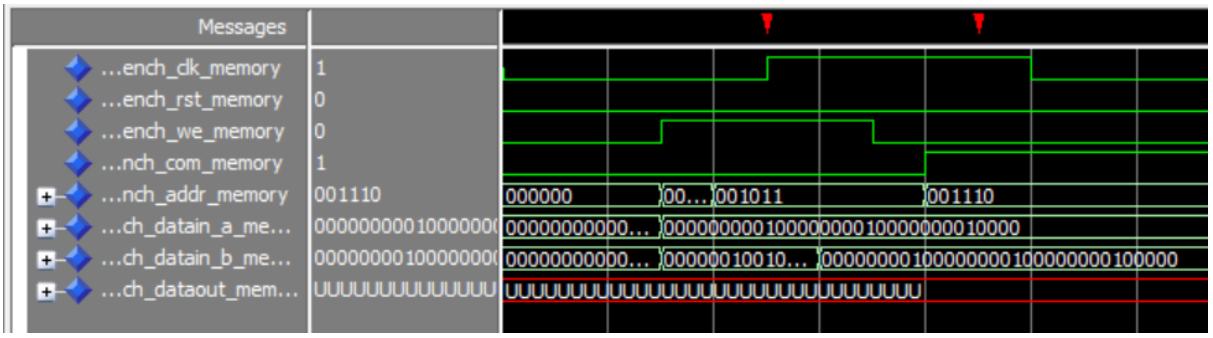
Nous avions décidé dans un premier temps de créer une variable de lecture S pour visualiser la copie de W et la sortie de l'UAL. Celle-ci étant précédemment seulement reliée par un signal sans communiquer sur sa sortie, les valeurs des drapeaux étaient plutôt cohérentes. Dans un second temps, nous avons essayé la déconnexion de W et S en déclarant dans notre entité deux variables distinctes, une pour l'entrée W et une pour la sortie S, mais cela n'a pas abouti. Enfin, nous tenterons d'essayer d'initialiser les valeurs des registres directement dans le banc de registres et vérifier seulement la bonne tenue des opérations pour valider notre assemblage.

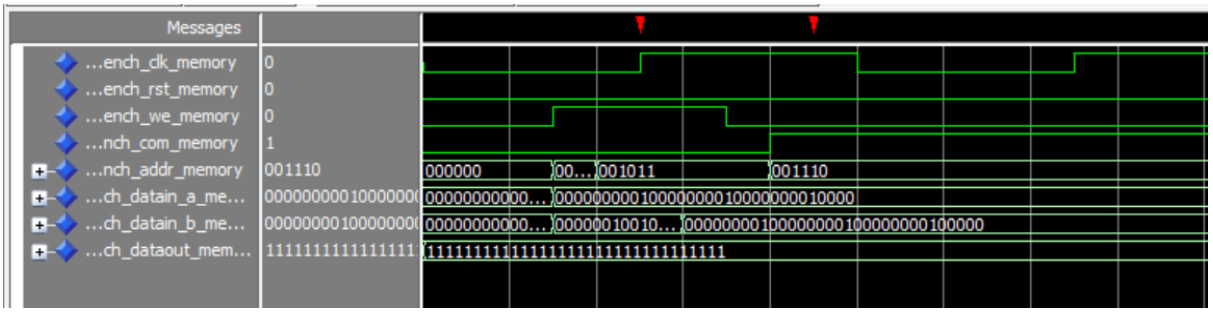
```
-- signal SIGNAL_Test_Bench_W_BRUAL : std_logic_vector(31 downto 0) :=
x"0000_0000"; -- Bit de resultat
```

1.3 Debogage Memoire de données

Nous avons tout d'abord essayé avec trois parties bien distinctes par le biais de trois process séparés : un pour le multiplexage interne des entrées A et B, un pour la lecture de la sortie et un pour son écriture.

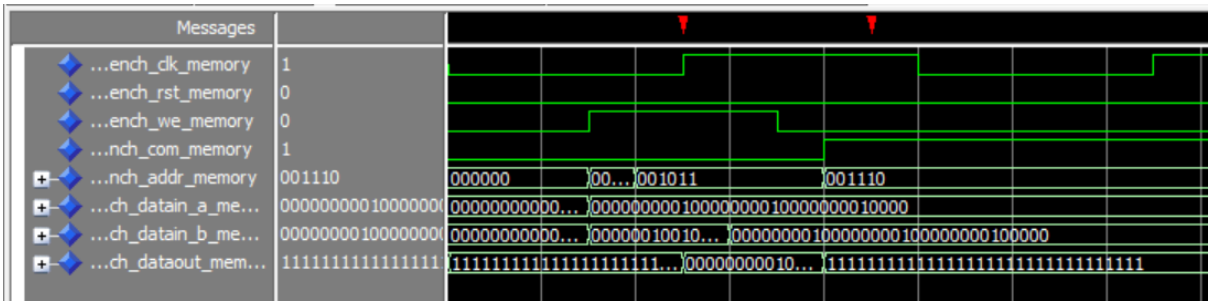
```
32  - Memory_Local_Mux_Process : process (COM_Memrory, DataIn_A_Memory, DataIn_B_Memory) -- Combinatoire assignation
33  -     begin
34  -         case COM_Memrory is
35  -             when '0' => DataIn_SIGNAL_Memory  <= DataIn_A_Memory;  -- Entree A
36  -             when '1' => DataIn_SIGNAL_Memory  <= DataIn_B_Memory;  -- Entree B
37  -             when others => DataIn_SIGNAL_Memory  <= (others => '0');   -- Dans le doute mais ne doit pas arrive
38  -         end case;
39  -     end process Memory_Local_Mux_Process;
40  -
41  - Memory_Reading_Process : process (Addr_Memory, Memory) -- Pas sur que Memory doit etre dans la liste de sensibi
      on reecrit sur le registre sans changer l adresse ...
42  -     begin
43  -         DataOut_Memory <= (Memory(to_integer(unsigned(Addr_Memory)))); -- Selection du Addr_Memory_ieme vecteur
44  -     end process Memory_Reading_Process;
45  -
46  - Memory_Writing_Process : process (Clk_Memory, Rst_Memory) -- Synchrone
47  -     begin
48  -         if Rst_Memory = '1' then -- Verification si redemarrage memoire
49  -             for i in 63 downto 0 loop
50  -                 Memory(i) <= (others => '0');
51  -             end loop;
52  -             DataOut_Memory <= (others => '0');
53  -
54  -         elsif (Rst_Memory = '0') then
55  -             if rising_edge(Clk_Memory) then
56  -                 if WE_Memory = '1' then
57  -                     Memory(to_integer(unsigned(Addr_Memory))) <= DataIn_SIGNAL_Memory;
58  -                 end if;
```



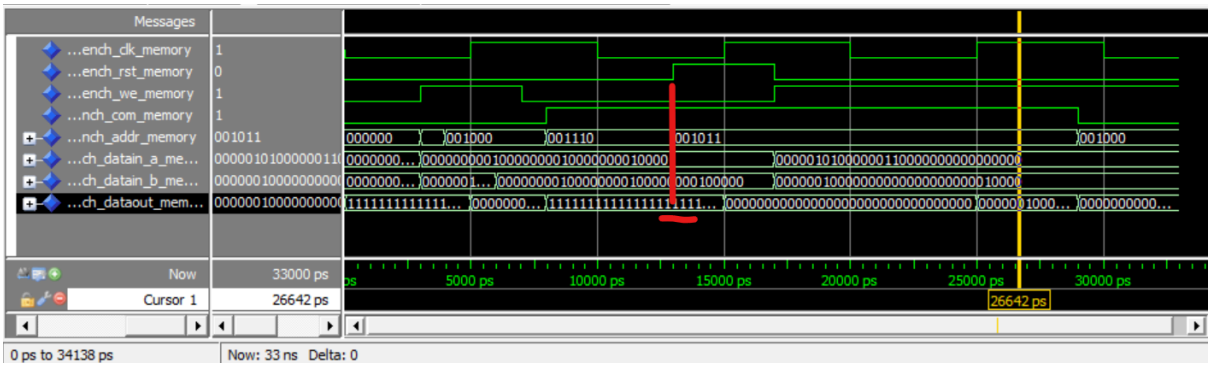


Nous avons ensuite effectué le multiplexage directement dans la partie écriture et nous nous sommes ensuite bataillé pour déterminer où placer l'affectation de la sortie. Nous l'avons tantôt mise dans le processus, tantôt après, avant de finalement trouver sa place adéquate au début.

```
31  -
32  - Memory_Writing_Process : process (Clk_Memory, Rst_Memory)
33  - begin
34  -     if Rst_Memory = '1' then -- Reset de la mémoire
35  -         Memory <= (others => (others => '0'));
36  -         DataOut_Memory <= (others => '0');
37  -     elsif rising_edge(Clk_Memory) then
38  -         if WE_Memory = '1' then
39  -             if (COM_Memrory = '0') then
40  -                 Memory(to_integer(unsigned(Addr_Memory))) <= DataIn_A_Memory;  -- Entree A
41  -             elsif (COM_Memrory = '1') then
42  -                 Memory(to_integer(unsigned(Addr_Memory))) <= DataIn_B_Memory;  -- Entree B
```



Ayant encore des erreurs, nous avons décidé de reprendre l'architecture depuis le début et avons omis que le reset était synchrone et donc il n'était plus dans la liste de sensibilité, ce qui a été corrigé dans la version suivante, le chronogramme ayant révélé cette erreur d'inattention.



12. Douze