Compte rendu du BE de VHDL

Conception d'un RISCv en VHDL

LESPIAUCQ Denis

CARVALHO Flavien

4 AE SE 3

28 novembre 2024

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Sommaire

Objectifs du BE

L'objectif de ce BE est de concevoir et d'implémenter un processeur RISC V à 5 niveaux de pipelines en VHDL. Ce dernier sera équipé d'une unité arithmétique pour les calculs arithmétiques simples, d'un banc de mémoire de données et d'instructions ainsi que d'un banc de registre à double port de lecture afin de pouvoir correctement faire fonctionner l'ensemble. Ainsi, ce bureau d'étude nous permettra de concevoir et simuler un processeur tout en réfléchissant sur les aléas, les composants synchrones et asynchrones ainsi que les moyens à mettre en œuvre pour rendre le processeur le plus rapide possible.

Matériel à notre disposition

Afin de mener à bien ce Bureau d'Etude en VHDL, nous avions à notre disposition le matériel suivant :

- Ordinateur (Windows / Linux)
- Logiciel Xilinx
- Une carte FPGA BASYS 3 Diligent

Il peut être important de noter que nous n'avons implanté sur la carte FPGA que notre Unité Arithmétique et Logique dont l'interface Homme/Machine a pu se faire à l'aide des différents boutons de la carte ainsi que de son bandeau de 16 LEDs.

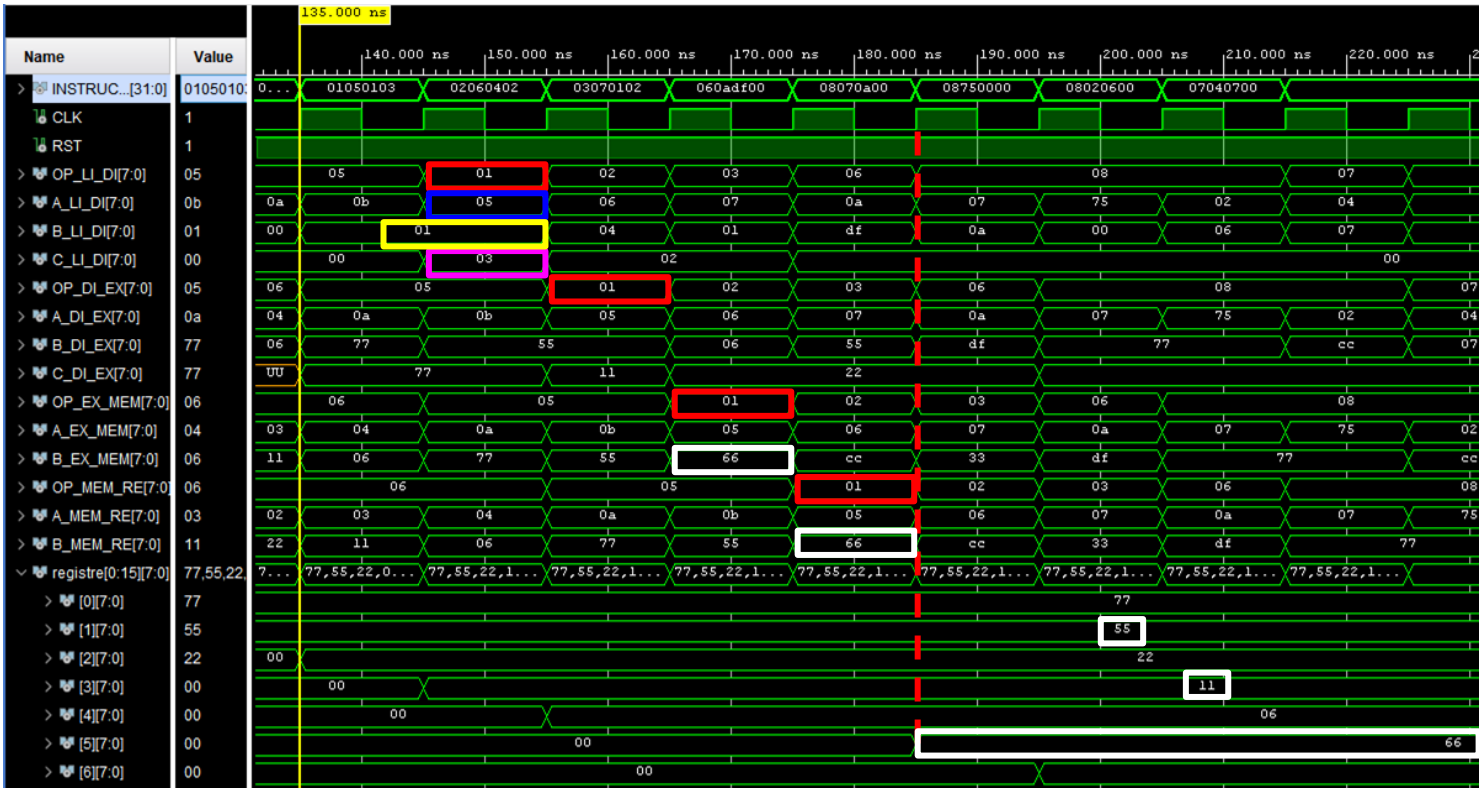
Unité Arithmétique et Logique (ALU)

Pour commencer ce BE, nous nous sommes occupés de l'ALU. Ce composant est capable de prendre en entrée deux signaux sur **8 bits** correspondants aux valeurs A et B ainsi que d'un signal de contrôle pour la sélection de l'opération à effectuer. En sortie, nous devons retrouver le signal correspondant au résultat sur **8 bits**, ainsi qu'un bit de **Carry** (addition), **Overflow** (multiplication) et **Négatif** (soustraction).

L'architecture comportementale repose sur un multiplexeur qui sélectionne dynamiquement l'opération à effectuer. Des signaux intermédiaires sont utilisés pour simplifier la détection des cas particuliers, notamment lors des calculs arithmétiques. Étant donné que ce composant est asynchrone, nous n'avons pas besoin d'utiliser un **process** déclenché par une horloge. Ainsi, la sortie retourne toujours le dernier résultat dépendant de la valeur du vecteur de contrôle. Afin de faciliter la lecture, de rendre plus rapide l'exécution et éviter les aléas, nous avons fait le choix d'utiliser un multiplexeur grâce à la condition **when**, où le signal intermédiaire **result** réceptionne le résultat sur **16 bits** avant de le convertir et de l'envoyer en sortie, sur **8 bits** (permet d'identifier la présence du bit de Carry ou Overflow).

Durant la phase de conception de ce composant, nous nous sommes heurtés à une difficulté. En effet, différentes opérations élémentaires, comme la **multiplication** ou le **xor** n'étaient pas reconnues. Nous avons fait de nombreuses recherches avant de s'apercevoir qu'il était indispensable d'inclure la bibliothèque **STD_LOGIC_ARITH**, utile pour ces opérations.

Dans notre cas, l'ensemble des opérations demandées ont été implémentées et ce composant ne présente aucun aléa de données. Il est possible de mettre en évidence le bon fonctionnement de quelques fonctions arithmétiques de l'ALU, comme par exemple l'addition dont le chronogramme est visible ci-dessous :



Dans ce cas, le code OP de l'addition est **x"01"**, et on souhaite effectuer la somme des registres **R1** et **R3** dont le résultat est directement stocké dans le registre **R5**. L'instruction est donc **x"01050103"**. Il est possible d'observer qu'au premier coup d'horloge (à 145 ns) le code OP ainsi que les signaux A, B et C se mettent à jour avec les nouvelles valeurs utiles et que le code (*rouge*) est transféré sur le canal OP des différents pipelines au cours du temps. Les signaux **B_DI_EX** et **C_DI_EX** se chargent quant à eux des valeurs des registres **R1** et **R3**, et le signal de sortie

B_EX_MEM se charge de la valeur correspondant à la somme, qui sera ensuite acheminée en sortie du pipeline pour être stocké dans le registre **R5** (*large encédré blanc*).

Banc de Registre

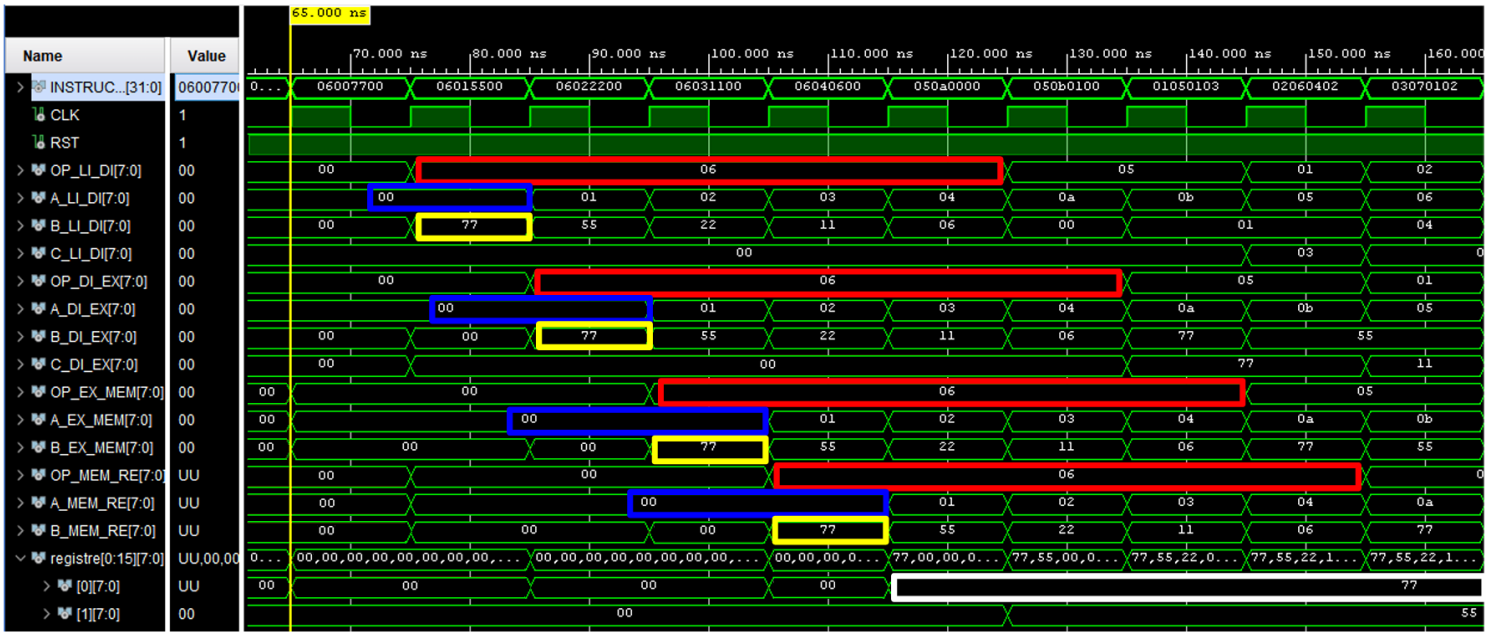
Le banc de registres est un composant synchrone nécessitant donc cette fois-ci un **process** cadencé sur l'horloge du processeur, d'où la présence de la ligne **if rising_edge(CLK) then** permettant le déclenchement d'une lecture ou d'une écriture à chaque front montant.

Pour la conception de ce composant, nous avons été confrontés à deux difficultés ; la première étant la mise en place d'un signal **registre**, car sa déclaration nous était encore peu familière. En effet, il était nécessaire de créer un tableau (**array**) de 16 emplacements de type **STD_LOGIC_VECTOR** sur 16 bits ainsi que le signal **registre** rattaché permettant la manipulation de ce tableau. La deuxième résidait dans le fait que nous souhaitions convertir la valeur du signal **addr_W** sur 4 bits en un **integer**. La fonction **TO_INTEGER** ne semblait pas fonctionner. Nous avons trouvé la solution en incluant au fichier VHDL la bibliothèque **NUMERIC_STD** qui permettait l'utilisation de cette fonction. Le problème lors du débogage était l'absence d'indications spécifiques sur l'erreur de la part du logiciel. Un simple message indiquant l'ignorance de la fonction était affiché.

Dans notre cas, l'ensemble des opérations demandées ont été implémentées et ce composant ne présente aucun aléa de données.

De même, nous avons fait attention à utiliser, lors de la conversion, la fonction **unsigned** afin d'indiquer que toutes les adresses fournies ne sont pas signées et restent positives. Enfin, il sera possible de remarquer que nous avons privilégié, dans la condition de début, la vérification du signal de reset **RST** permettant la mise à zéro de l'ensemble des registres du banc.

Pour l'affectation du registre **R0** avec la valeur **x"77"** (soit 119), nous utilisons alors l'encodage "**06007700**" (x"06" pour l'AFC, x"00" pour le registre 0 puis x"77" pour la valeur à stocker). Ainsi, par simulation, nous obtenons le chronogramme suivant :



Ainsi, comme visible sur ce chronogramme, il est possible d'observer qu'au front d'horloge suivant la mise à jour du code OP, le registre **R0** est affecté de la valeur **x"77"**. Il existe bien un petit délai de **4 coups d'horloge**, ce qui est tout à fait cohérent avec notre structure du processeur. À nouveau, il est possible de voir la transmission de l'OP CODE en cascade.

Banc de Mémoire

Le banc de mémoire est un composant fondamental dans notre processeur RISC-V, permettant de stocker les données ou les instructions sur 256 emplacements de 8 bits chacun (similaire au banc de registre, mais avec une place plus importante). Son fonctionnement repose sur un tableau de mémoire initialisé à zéro et synchronisé avec le signal d'horloge, car ce composant est aussi synchrone. Lorsqu'un signal de réinitialisation **RST** est actif, tout le contenu de la mémoire est remis à zéro. En mode écriture (**RW = 0**), la donnée fournie sur l'entrée **Data_in** est stockée à l'adresse spécifiée par le signal d'entrée **addr**, après conversion de cette dernière en entier. En mode lecture (**RW = 1**), la donnée correspondant à l'adresse est transférée sur la sortie **data_out**. Ce banc de mémoire garantit une gestion fiable et flexible des données nécessaires au processeur, tout en assurant un accès rapide aux données ; ce composant pourrait être assimilé à de la mémoire *cache* ou *RAM*.

La conception de ce banc de mémoire ne nous a posé quasiment aucune difficulté particulière puisque la structure même de ce composant est quasi identique à celle du banc de registre, à la différence qu'il existe **256 emplacements** mémoires au lieu de 16, et que les données sont formatées sur **32 bits** au lieu de 8. Ce banc de mémoire est utilisé pour le stockage de l'ensemble de "programme" du microcontrôleur avec l'ensemble des instructions. On peut éventuellement citer la difficulté liée à la syntaxe pour la définition des signaux. Il nous a fallu plusieurs tentatives avant de réussir à créer le signal contenant les vecteurs, et de trouver la bonne syntaxe pour écrire dans ce signal.

```
signal data_mem : mem_type := (others => X"00");
```

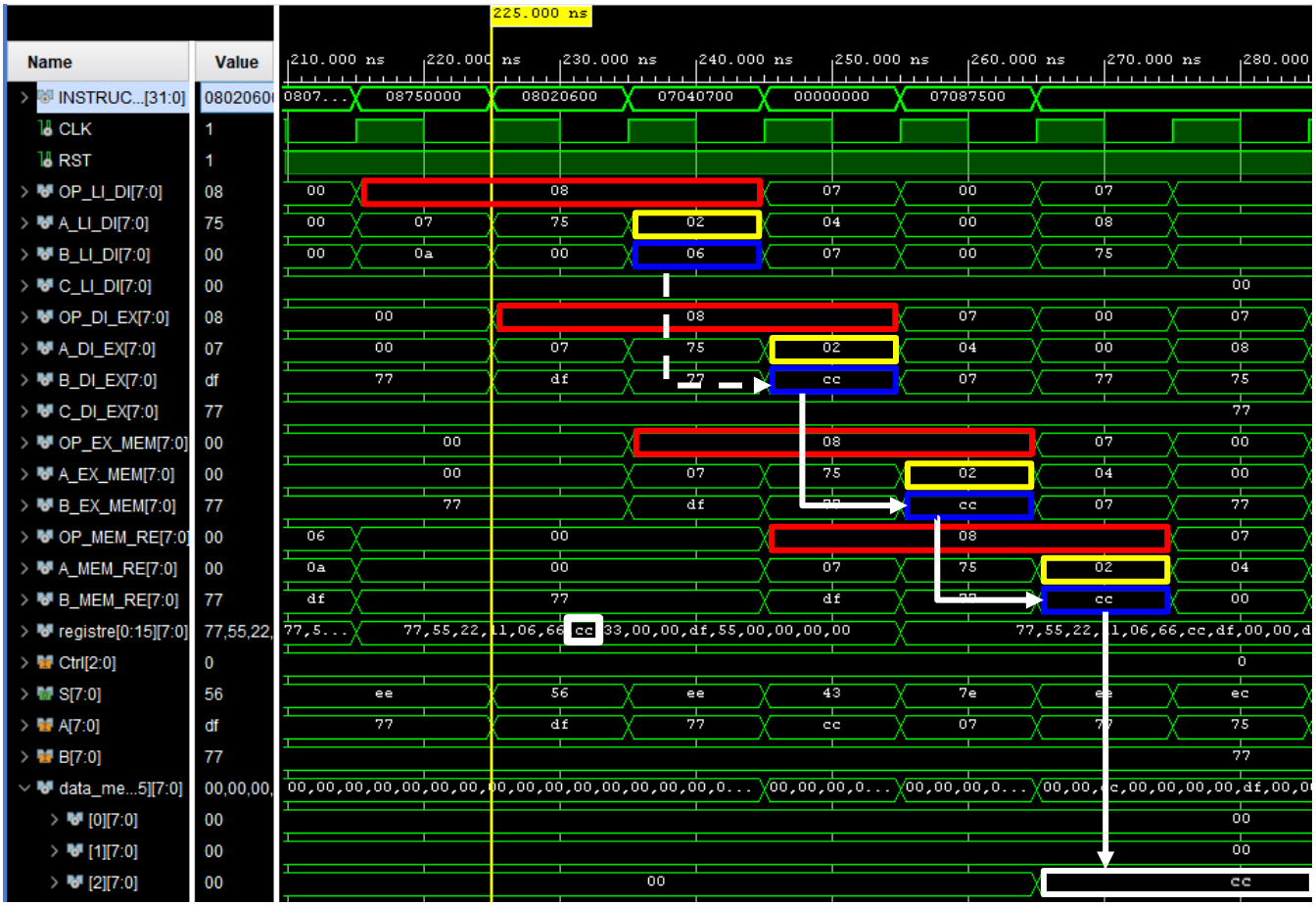
data_mem est un signal de type mem_type (défini comme un tableau de vecteurs sur 8 bits), il ne contient que des 0.

```
data_mem(TO_INTEGER (unsigned (addr))) <= data_in;
```

data_mem décalé de addr reçoit la donnée entrante.

Ces deux lignes sont les lignes qui ont posé un problème en termes de syntaxe. La conversion de type **<TO_INTEGER(unsigned(addr))>** permet d'utiliser un signal **STD_LOGIC_VECTOR** comme un indice de tableau, elle est obligatoire.

Enfin, il nous pouvons observer le chronogramme issu de la simulation du Banc de Mémoire, visible en figure ci-dessous. Ce chronogramme met en évidence le fonctionnement du **STORE** par notre Banc de Mémoire. En effet, il est possible d'observer qu'au moment de l'appel par le code OP **x"08"** pour enregistrer le contenu de **R6** à l'adresse **n°2** (instruction **x"08020600"**), ce dernier suit à nouveau le pipeline dans sa branche OP, la branche A gardant la valeur de l'adresse et la B la valeur du registre **R6** à partir du **DI/EX** (exécution). Finalement, au 4ème coup d'horloge, à l'adresse n°2, on retrouve bien la valeur **x"CC"** correspondant à la valeur du registre **R6**.



Banc d'Instructions

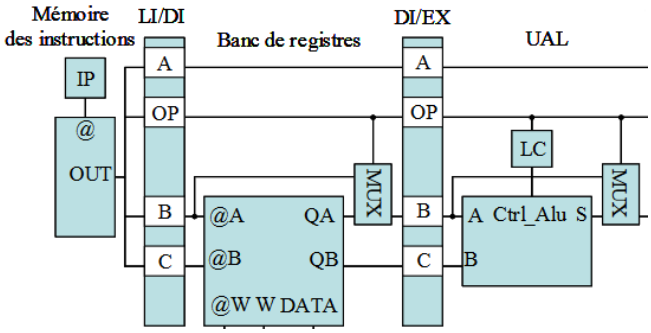
Le banc d'instructions est chargé de stocker les instructions nécessaires au fonctionnement de notre processeur RISC V. Dans notre cas, nous avons choisi qu'il s'agirait d'une mémoire de type ROM (Read-Only Memory) comportant 256 emplacements, chacun pouvant contenir une instruction encodée sur 32 bits. Ainsi, il suffit d'utiliser le composant "Banc Mémoire" comme "Banc d'Instructions" en y ajouter l'ensemble des instructions en dure directement dans le code interne : les instructions, préchargées dans la mémoire lors de la synthèse, sont codées de manière compacte en plusieurs champs : un code opération **OP** qui identifie l'opération à réaliser (par exemple, **AFC** pour l'affectation immédiate, **COPY** pour la copie de registres, ou des opérations arithmétiques comme **ADD**, **MULT**, **SOUS**), ainsi que des paramètres tels que les numéros de registres ou des valeurs immédiates. Ces instructions permettent d'initialiser des registres, d'effectuer des calculs ou de réaliser des accès mémoire (lecture avec **LOAD** ou écriture avec **STORE**). À chaque front montant du signal d'horloge (**CLK**), l'instruction située à l'adresse spécifiée par le signal d'entrée **addr** est extraite de la mémoire et envoyée sur la sortie **data_out**. L'adresse, exprimée sous forme de vecteur logique binaire (**STD_LOGIC_VECTOR**), est convertie en entier avec la fonction **TO_INTEGER(unsigned(addr))** pour accéder au tableau mémoire. Cette mémoire est initialisée avec un programme exemple qui couvre une variété d'opérations typiques, permettant ainsi de simuler et tester le comportement du processeur.

```
-- AFC R0,#119
data_mem(1) <= x"06007700";          -- R0 = 119            0x77
-- COP R10,R0    (R10 <- R0)
data_mem(2) <= x"050A0000";          -- R10 = 119 = R0      0x77
-- ADD R5,R0,R10   (R5 <- R0 + R10)
data_mem(3) <= x"0105000A";          -- R5 = 119 + 119      0xEE
-- SOU R7,R5,R0 (R7 <- R5 - R0)
data_mem(4) <= x"03070500";          -- R7 = 238 - 119      0x77
-- STORE @7,R5
data_mem(5) <= x"08070500";          -- @7 <- 238 = R5      0xEE
-- LOAD R4,@7
data_mem(6) <= x"07040700";          -- R4 = 238 <- @7      0xEE
```

Chemin des Données

Finalement, il ne nous reste plus qu'à implémenter directement notre composant RISC_V. Il s'agit du système complet intégrant plusieurs blocs fonctionnels interconnectés pour assurer les différentes étapes du cycle d'exécution des instructions. Les principaux composants incluent un banc d'instructions, un banc de registres, une unité arithmétique et logique (ALU), et un banc mémoire, reliés via des signaux qui véhiculent les données et les commandes.

Le banc d'instructions possède l'ensemble des instructions encodées sur 32 bits. Chaque instruction est ensuite décodée, divisée en différents champs (opération, opérande, valeurs immédiates), et transmise au niveau suivant. La phase de décodage (**LI/DI**) analyse l'opération et sélectionne les données pertinentes à extraire ou à manipuler dans le banc de registres ou directement depuis les instructions.



Pendant la phase d'exécution (**DI/EX**), si l'instruction correspond à une opération arithmétique (**ADD**, **SUB**, **MUL**) ou logique, elle est traitée par l'ALU. Cette unité utilise les opérandes extraits pour produire un résultat (**S**) en fonction du contrôle (**Ctrl**). Les signaux de dépassement (**Overflow**), de retenue (**Carry**) et de résultat négatif (**Negatif**) sont également générés pour certains cas.

Nous utilisons une boucle d'incrémentation automatique pour appeler chaque instruction dans l'ordre :

```
IP <= std_logic_vector(unsigned(IP) + 1); -- Incrémentation de IP
```

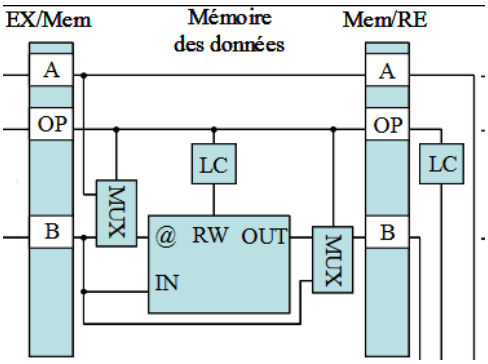
Si nous avions eu le temps de concevoir l'instruction **JUMP**, il nous aurait été possible d'effectuer des sauts d'instructions pour pouvoir effectuer des opérations beaucoup plus complexes.

Pour chaque niveau de pipeline, il est nécessaire de vérifier le type d'opérateur **OP** réceptionné afin de pouvoir effectuer les différentes affectations nécessaires. Par exemple, pour le premier étage **LI/DI**, nous avons :

```
case OP_LI_DI is
    when  x"01" => ALU_CONTROL <= "000";
    when  x"02" => ALU_CONTROL <= "010";
    when  x"03" => ALU_CONTROL <= "001";
    when  others => ALU_CONTROL <= "000";
end case;
```

Pour le cas de l'addition (**x"01"**), la multiplication (**x"02"**) et la soustraction (**x"03"**), nous affectons le bon signal correspondant à l'opération à effectuer à l'entrée de contrôle de l'**ALU**.

La phase de mémoire (**EX/MEM**) gère les interactions avec le banc mémoire. Pour les instructions de type **LOAD** ou **STORE**, les données sont lues ou écrites dans la mémoire à des adresses spécifiées. Les autres instructions conservent les résultats directement depuis l'**ALU**.



Enfin, lors de la phase d'écriture (**MEM/RE**), les données calculées ou extraites de la mémoire sont renvoyées vers le banc de registres pour être stockées dans le registre cible. Grâce à cette étape, on garantit que le résultat de l'instruction est accessible pour les opérations suivantes.

Le design utilise une approche de type **pipeline**, c'est-à-dire que les données circulent entre les étapes via des signaux intermédiaires (**OP_LI_DI**, **A_DI_EX**, etc.), ce qui permet une exécution continue et optimisée des instructions. Le composant offre une structure flexible et modulaire, représentative des principes de conception des processeurs RISC-V, et met en œuvre un flot cohérent des données et des contrôles à chaque étape du cycle d'exécution.

Synthèse

Enfin, une fois que l'ensemble de notre processeur RISC V fut programmé, testé et validé, nous avons pu exécuter la synthèse complète de ce système et obtenir les performances qui lui sont liées. Ainsi, nous nous sommes concentré sur les informations du "**Max Delay Paths**" afin de connaître le temps maximal que met les signaux à traverser notre processeur pour le pire des cas. La synthèse nous permet d'obtenir le rapport suivant :

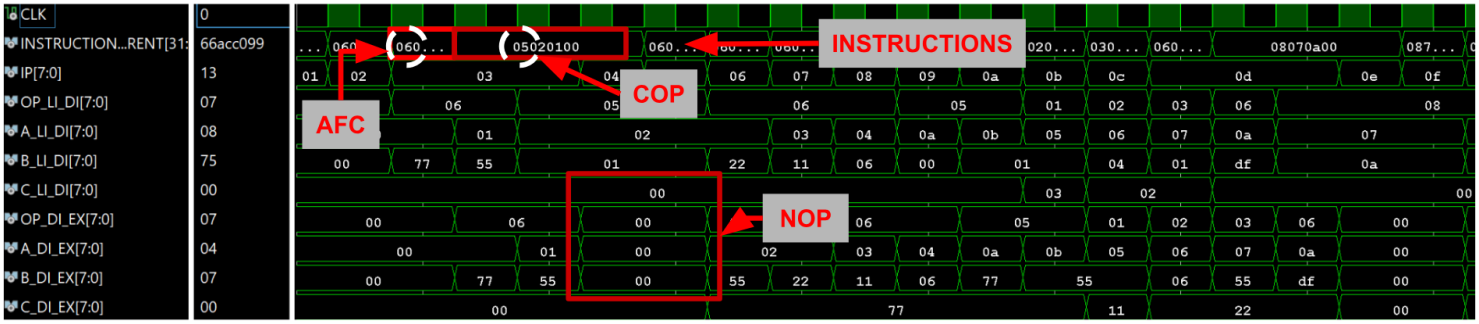
```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (VIOLATED) :        -1.904ns  (required time - arrival time)
  Source:                 C_DI_EX_reg[2]/C
                            (rising edge-triggered cell FDRE clocked by CLK  {rise@0.000ns fall@2.000ns period=4.000ns})
  Destination:            B_EX_MEM_reg[7]/D
                            (rising edge-triggered cell FDRE clocked by CLK  {rise@0.000ns fall@2.000ns period=4.000ns})
  Path Group:             CLK
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            4.000ns  (CLK rise@4.000ns - CLK rise@0.000ns)
  Data Path Delay:        5.753ns  (logic 2.390ns (41.544%)  route 3.363ns (58.456%))
  Logic Levels:           7  (CARRY4=2 LUT4=1 LUT5=2 LUT6=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.079ns = ( 6.079 - 4.000 )
    Source Clock Delay      (SCD):    2.402ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Le "**Data Path Delay**" est ici de **5,753 ns**, ce qui nous donne une fréquence maximale de notre processeur de :

$$f_{MAX} = \frac{1}{T_{MAX}} = \frac{1}{5{,}753 \times 10^{-9}} = 173{,}8\ MHz$$

Ainsi, pour garantir un bon fonctionnement de notre RISC V, il est impératif de ne pas dépasser cette fréquence. Augmenter serait néfaste car certains signaux seraient plus rapides traverser les étages que d'autres, causant alors des perturbations au sein du processeur et créant de potentiels aléas. Il est important de préciser que notre période d'horloge est de **6,5 ns**, ce qui nous permet d'obtenir une fréquence de fonctionnement de **153,85 MHz**, ce qui est légèrement plus faible que la fréquence maximale mais garantissant un bon fonctionnement et une stabilité de notre processeur. Nous considérons que cette fréquence pour notre processeur FPGA est suffisante et confortable, mais si nous souhaitions l'augmenter (jusqu'à 250 MHz par exemple), nous serions dans l'obligation d'apporter des optimisations, comme en **divisant l'étage le plus long** en ajoutant un pipeline afin de réduire les niveaux logiques et optimiser le **chemin critique**.

Dans le but d'éviter les aléas de données, nous avons protégé le fonctionnement du processeur pour éviter, dans le cas d'un **AFC** suivi d'un **COP** du même registre, que la donnée copiée soit l'ancienne valeur du registre. Nous pouvons ainsi constater le bon fonctionnement sur le chronogramme ci-après en remarquant que l'horloge du pointeur d'instruction est inhibée pendant 2 coups d'horloge et que des bulles correspondant à un **NOP** sont insérées dans le pipeline.



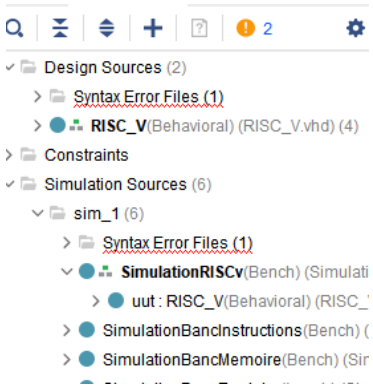
Difficultés générales

Maîtrise du logiciel, nous avons mis du temps à comprendre comment fonctionnait le logiciel. Nous faisions une synthèse avant chaque simulation, alors que cela est inutile. Nous avons perdu du temps à cause de cela, mais nous avons compris la distinction entre synthèse et simulation. Cela nous a permis ensuite de perdre moins de temps lors du debug.

Délais entre les signaux, l'une des plus grosses difficultés sur ce BE durant l'assemblage des différents composants dans le microprocesseur final. Gérer les différents délais qui peuvent exister entre les étages du pipeline était particulièrement subtile du fait qu'il existe une latence entre chaque process et qu'il faut anticiper les délais des entrées des composants et de leurs sorties. Étant donné qu'il faut un process pour actualiser les entrées d'un composant (fonctionnement synchrone), la sortie du composant ne sera mise à jour qu'à la fin de l'exécution du deuxième process.

Debug, contrairement aux autres langages de programmation classiques (C/C++, Java, Python), déboguer en VHDL est un vrai challenge et demande une fine analyse des signaux pour valider notre programmation et pour trouver les différents problèmes qu'il peut y avoir. Déboguer a été le processus le plus délicat durant tout le BE. Nous avons beaucoup appris à ce sujet grâce à ce projet.

Logiciel, il nous est arrivé plusieurs fois de faire des modifications qui n'affectent absolument pas le comportement de notre microprocesseur et pour cause, le logiciel n'affichant pas clairement qu'il ne comprend pas le code donné.



Nous avons appris que lorsque ce type d'avertissement est présent, le logiciel va utiliser l'ancienne version correcte de notre code et les modifications ne vont pas être prises en compte. Il n'affiche pas d'erreur et lance la simulation qui est décorrélée du code. Nous avons perdu beaucoup de temps avant de comprendre cela.

Conclusion

Pour conclure ce Bureau d'Étude de VHDL, nous avons été capables de mettre en application l'ensemble des notions vues en cours pour la conception des différents composants, tout en debugant correctement notre programme grâce aux différentes simulations effectuées durant nos séances. De plus, nous sommes maintenant capables d'interpréter ces chronogrammes tout en repérant les différents aléas pour pouvoir les corriger. Enfin, nous nous sommes habitués à l'utilisation du logiciel Xilinx ainsi qu'à ses différents outils, nous permettant de mener à bien ce BE de VHDL.

Ce BE est la concrétisation de plusieurs années à se demander comment fonctionne un processeur. Ce projet démystifie en quelque sorte ce dernier. Nous avons souffert durant ce projet, mais l'apprentissage acquis est immense.

Nous tenions sincèrement à remercier Madame DRAGOMIRESCU et Monsieur LOUBET pour leur investissement ainsi que leur précieuse aide durant ce projet qui nous a énormément intéressé et passionné.

Annexe

Chronogrammes

Dans cette section, il vous sera possible de retrouver d'autres chronogrammes et extraits de codes VHDL permettant d'étayer nos explications tout en validant notre architecture.

Ainsi, pour l'ALU, il est possible de retrouver différents chronogrammes correspondants à différentes d'opérations :

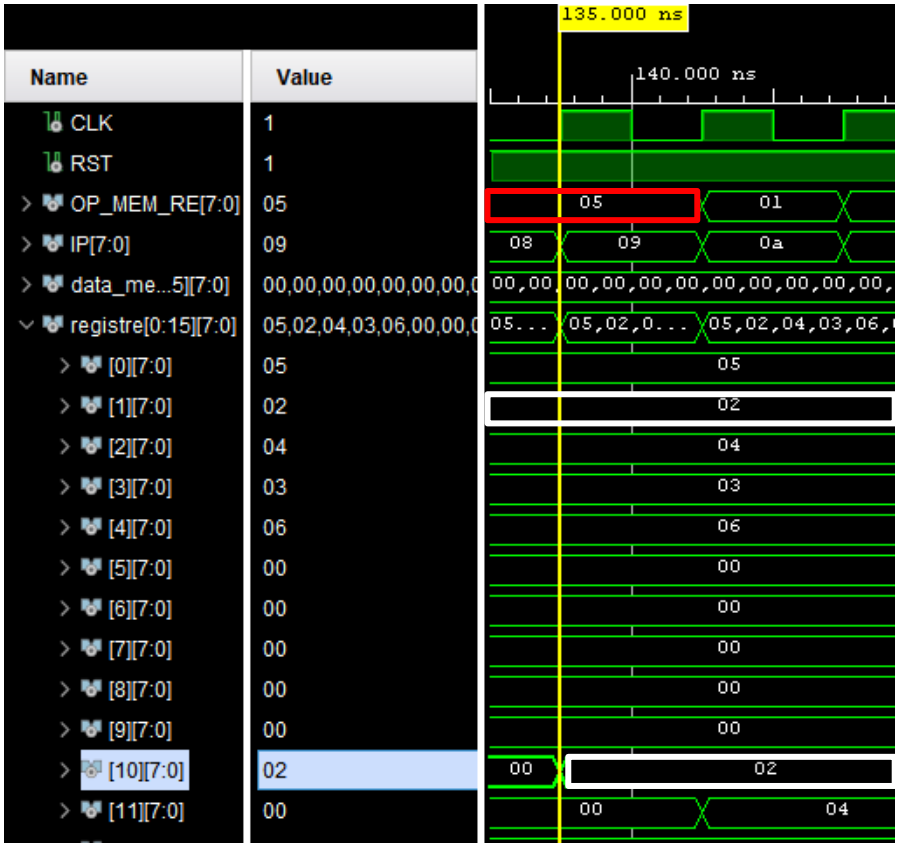


Figure 1 : Copie de R1 dans R10

Sur ce chronogramme, le registre **R1** a été au préalable affecté par la valeur **x"02"**, et se retrouve copiée directement dans le registre **R10** lors de l'exécution de l'OP **COPY**. Ici, le code OP de la copie est **x"05"**, comme il est possible de le constater à la ligne "**OP_MEM_RE**" du tableau. Nous pouvons observer qu'il n'y a aucun aléa de données sur cette fonction, et que l'instruction est bien effectuée au coup d'horloge.

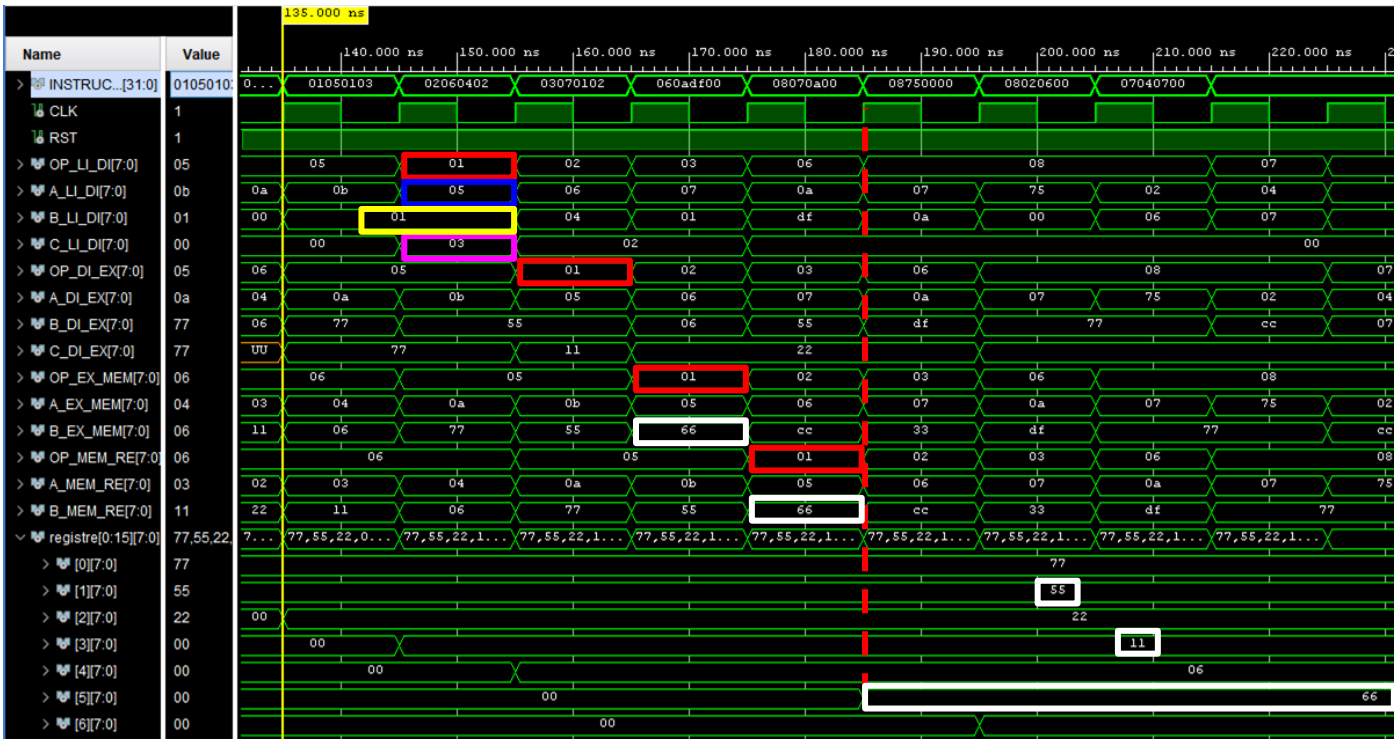


Figure 2 : Addition de R1 et R3 dans R5

Sur ce chronogramme, il est possible de retrouver le schéma traditionnel en cascade de l'information qui se propage d'un pipeline à un autre dans le temps. Ici, il s'agit d'une addition, associée au code d'exécution **x"01"** comme visible dans les encadrés en rouge. Lors de la première étape **LI/DI**, l'ensemble des signaux d'entrée (**OP**, **A**, **B** et **C**) sont affectés par les valeurs issues directement de l'instruction encodée sur **32 bits**. Ainsi, dans ce cas, il s'agit d'une **addition** (**x"01"**) entre les registres **R1** (**B = x"01"**) et **R3** (**C = x"03"**) et donc le résultat devra être stocké dans le registre **R5** (**A = x"05"**).

La valeur du registre final est transmise jusqu'au dernier pipeline pour transmettre l'information au Banc de Registre, tout comme le code OP, ce qui n'est pas le cas de la valeur de B qui prend la valeur du registre **R1** à l'étape **DI/EX** et **C** qui fait de même pour le registre **R3**. Ce n'est qu'à l'étape d'exécution **EX/MEM** que le signal B récupère le résultat de l'addition, en sortie de l'**ALU**. Finalement, au dernier coup d'horloge, le Banc de Registre stocke bien le résultat dans le registre **R5**.

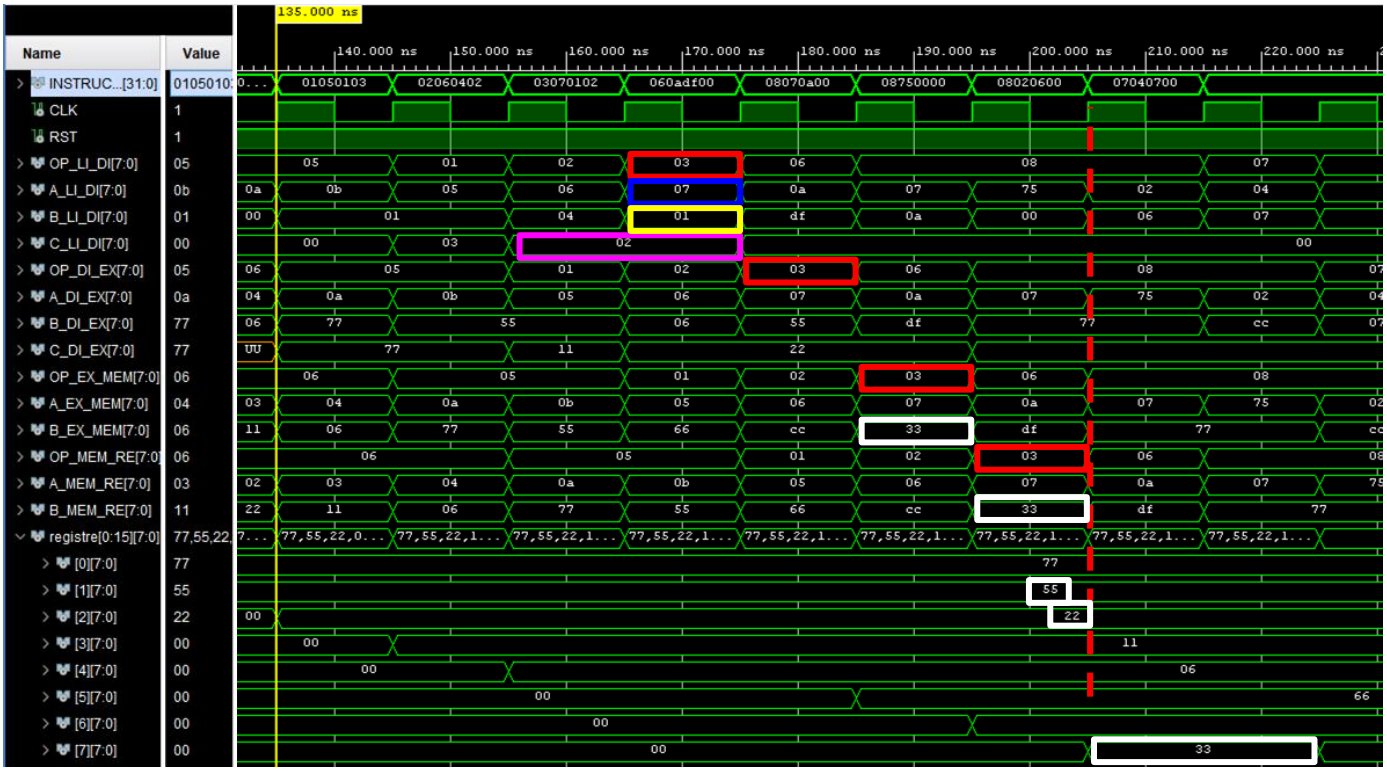


Figure 3 : Soustraction de R1 par R2 stocké dans R7

Ce chronogramme est en tous points similaire au chronogramme précédent, à l'exception qu'il s'agit d'une soustraction et que son code d'exécution devient **x"03"**, comme visible sur les lignes **OP** entourées en *rouge*.

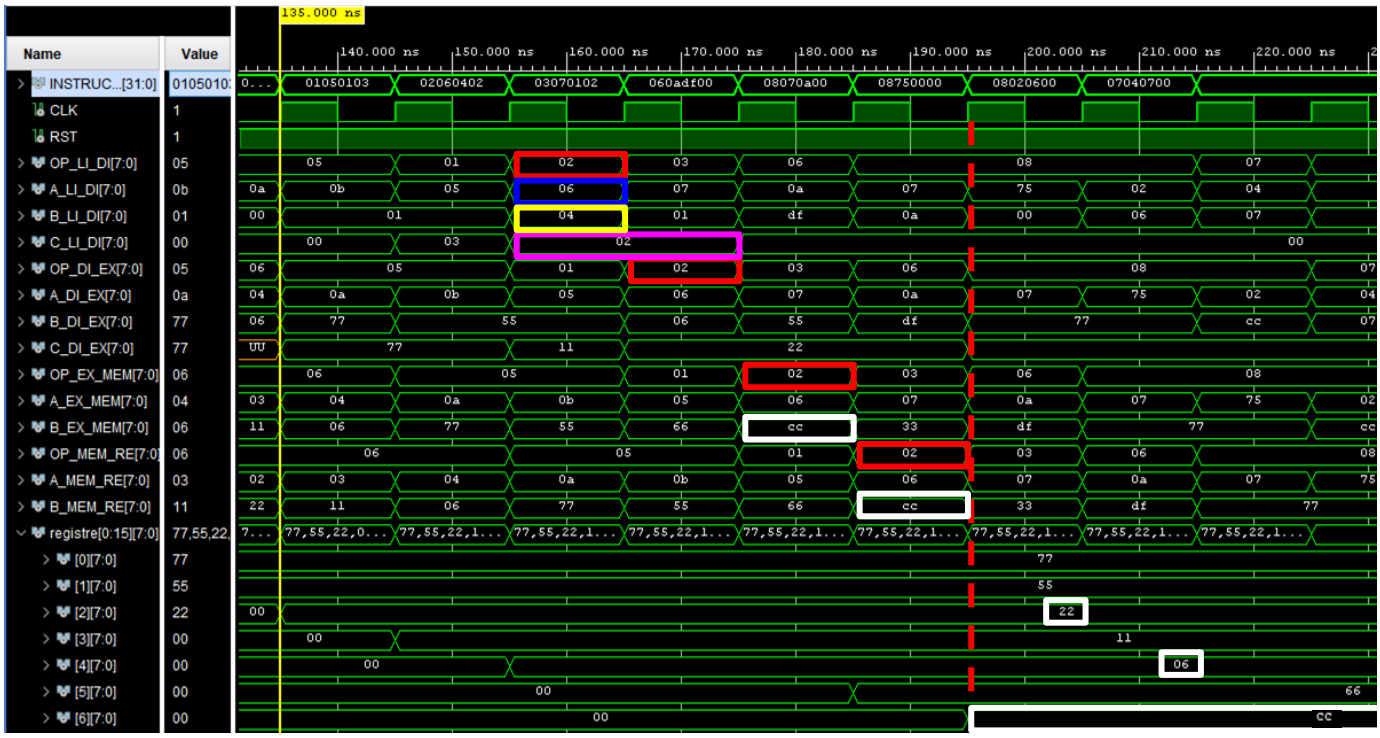


Figure 4 : Multiplication de R4 et de R2 stocké dans R6

Ce chronogramme est en tous points similaire au chronogramme précédent, à l'exception qu'il s'agit d'une multiplication et que son code d'exécution devient **x"02"**, comme visible sur les lignes **OP** entourées en *rouge*.

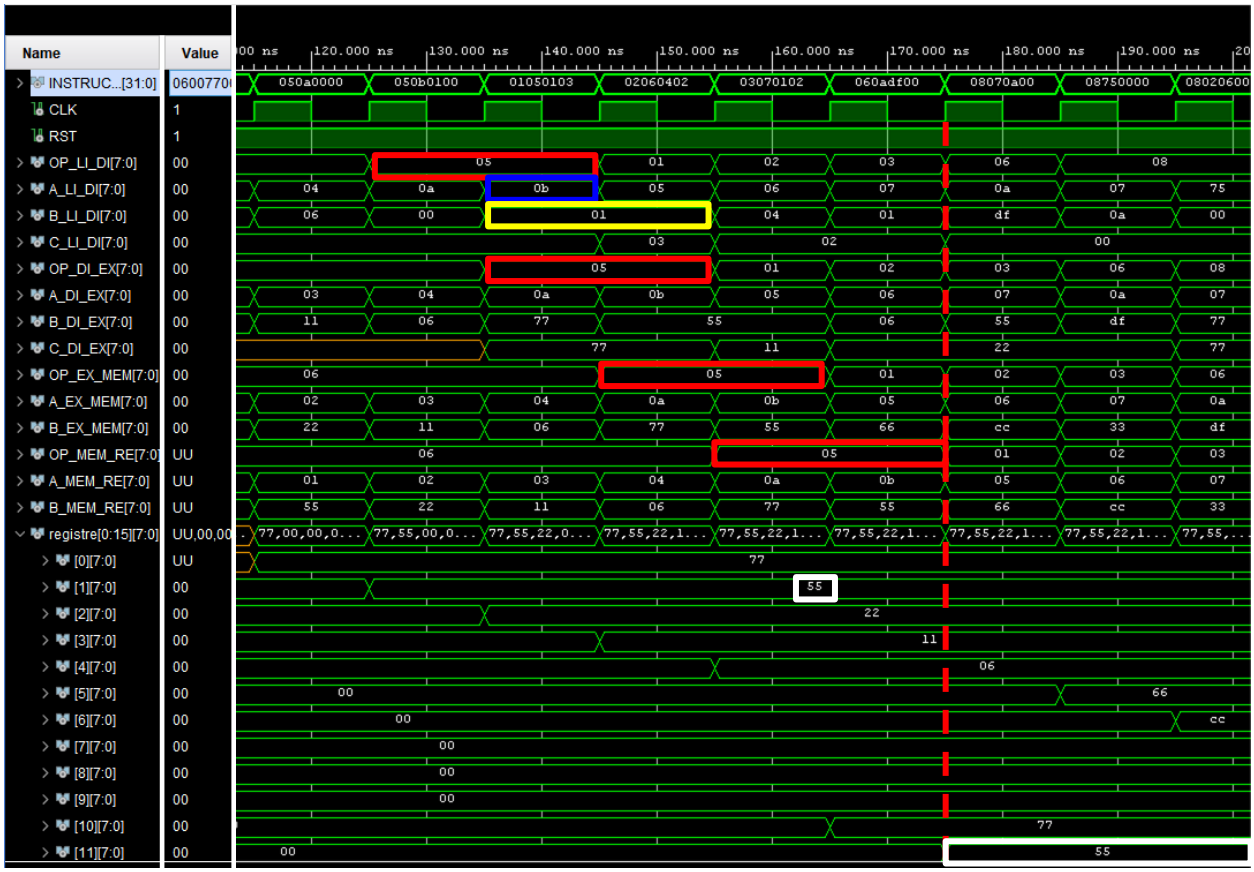


Figure 5 : Copie de R1 dans R11

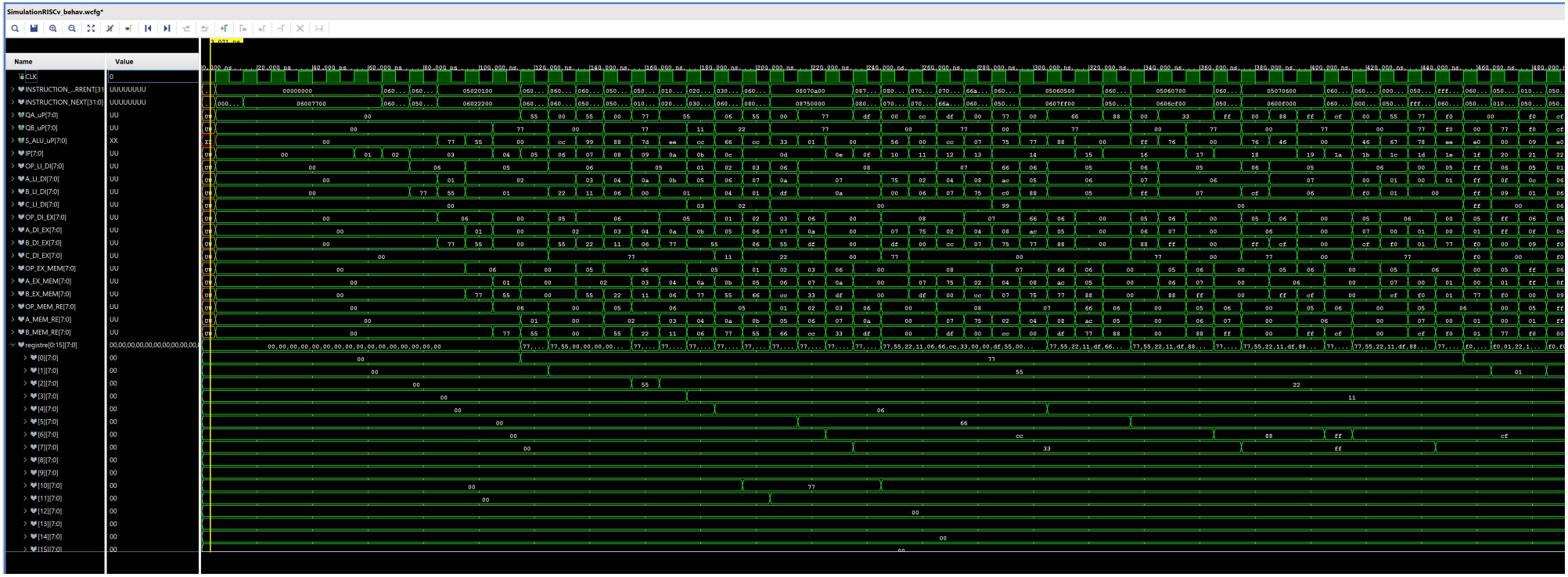


Figure 6 : Partie 1/2 du chronogramme complet

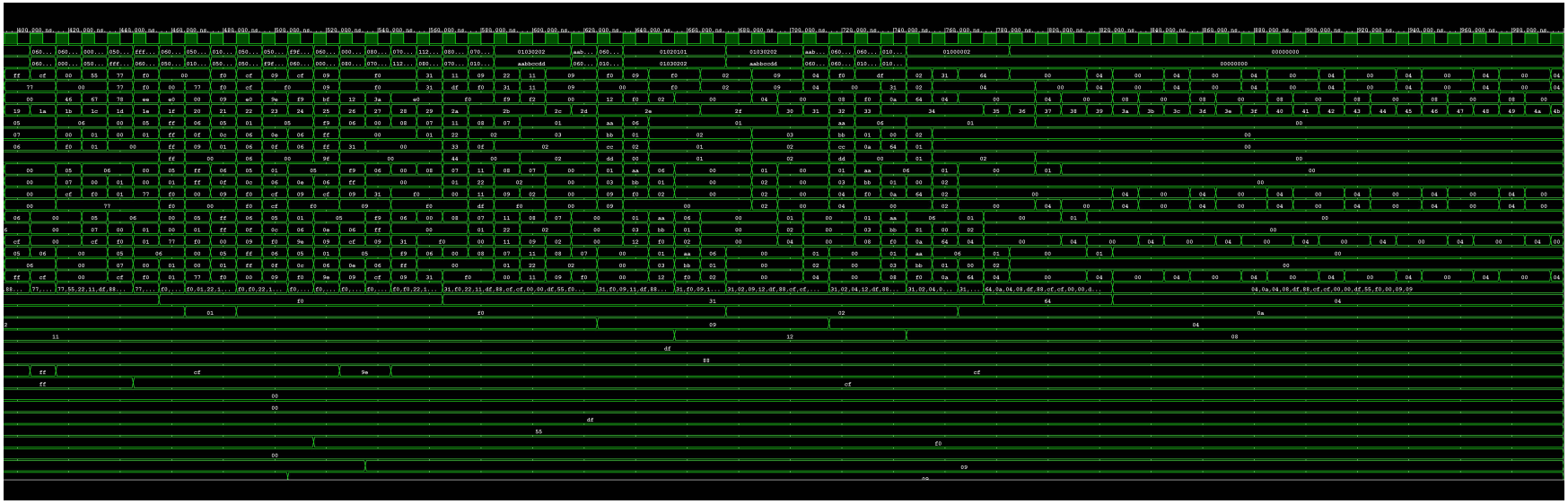


Figure 7 : Partie 2/2 du chronogramme complet