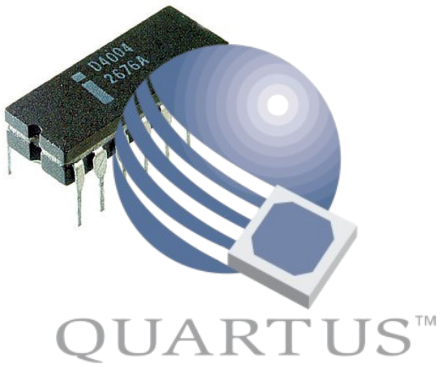
Compte rendu de TP VHDL

Mise en oeuvre d'un Microprocesseur 4 bits par synthèse logique de haut niveau sur FPGA



Daniel Arciniegas & Quentin Bernyer

UNIVERSITÉ TOULOUSE III PAUL SABATIER

TABLE DES MATIÈRES

Introduction

Le but de ce projet est de concevoir et de simuler un microprocesseur 4 bits en VHDL. Cette architecture simplifiée est inspirée du modèle SAP (Simple As Possible) pour introduire les concepts fondamentaux des microprocesseurs. L'objectif principal est de comprendre comment les composants de base (compteur de programme, mémoire, décodeur, accumulateurs, unité arithmétique et logique) interagissent pour exécuter des instructions.

Nous utilisons une approche modulaire, en développant et en testant chaque composant individuellement avant de les intégrer dans une architecture globale. Ce travail met également en avant les principes de conception numérique, notamment l'utilisation de bus internes, de signaux de contrôle et de haute impédance pour éviter les conflits. Une fois les simulations validées, les résultats sont analysés pour évaluer la faisabilité d'une implémentation sur FPGA.

1. Compteur de programme (PC)

Objectif du PC :
Le compteur de programme (PC) permet de suivre la séquence des instructions dans la mémoire du microprocesseur. C'est comme une liste de tâches où le PC pointe sur la prochaine tâche à exécuter.

Choix de la structure du code :
Nous utilisons un processus pour gérer le PC, car un processus permet d'exécuter des actions sur chaque impulsion d'horloge (le "tic-tac" du microprocesseur). Ce comportement cyclique est essentiel pour synchroniser tous les composants du microprocesseur.

Choix des variables :

- `integer range 0 to 15` : Un entier est utilisé pour représenter les valeurs possibles (de 0 à 15), car nous travaillons avec une mémoire contenant 16 adresses (4 bits suffisent pour compter jusqu'à 15). C'est un choix naturel pour gérer un compteur.

- `std_logic_vector` : Une fois le comptage effectué, le résultat est converti en un vecteur binaire (`std_logic_vector`), car les autres composants du microprocesseur communiquent en binaire.

Exemple simple pour comprendre : Imaginez un escalier de 16 marches. Le compteur de programme est comme quelqu'un qui monte une marche à chaque signal (horloge), puis recommence à la première marche après la 16e.

Résultat de simulation :

- Le compteur fonctionne correctement, incrémentant à chaque front d'horloge.

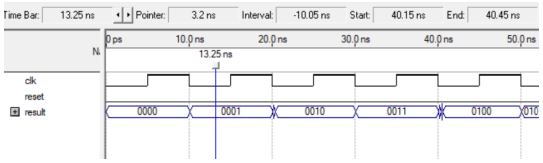


Fig 1 : Simulation du compteur

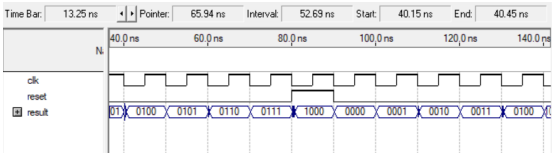


Fig 2 : Simulation du compteur avec reset

- Problème rencontré : La fréquence d'horloge était trop rapide, ce qui a nécessité un ralentissement (>20ns) pour assurer un fonctionnement stable.

2. Mémoire de programme (PM)

Objectif de la PM :
La mémoire stocke le programme que le microprocesseur doit exécuter. Chaque instruction y est codée sur 8 bits : 4 bits pour l'action (par exemple, "ajouter") et 4 bits pour les données associées (par exemple, "ajoute 3").

Choix de la structure :
Nous utilisons une RAM prédéfinie (méga-fonction) car elle est optimisée pour les FPGA. Cela simplifie la conception et assure des performances fiables sans devoir coder la mémoire à la main.

Pourquoi 8 bits ?

- 4 bits sont suffisants pour coder jusqu'à 16 instructions différentes (comme NOP, ADD, SUB, etc.).
- Les 4 bits restants permettent de transmettre une petite donnée (par exemple, un nombre à additionner). C'est comme écrire une commande et une petite note dans un carnet.

Réalisation :
Utilisation de *MegaWizard Plugin Manager* pour générer une mémoire RAM type `LPM_RAM_DQ`.

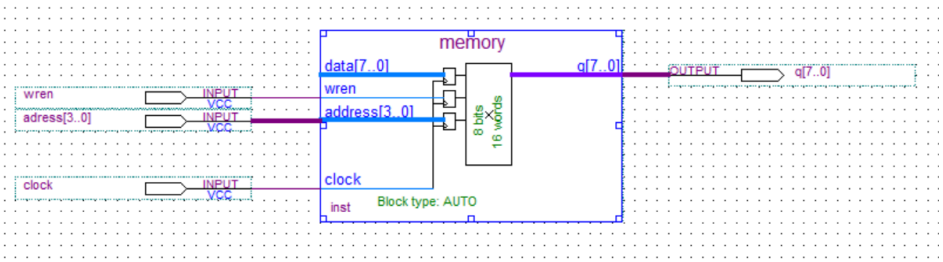


Fig 3 : Bloc mémoire.

Simulation de la mémoire seul :

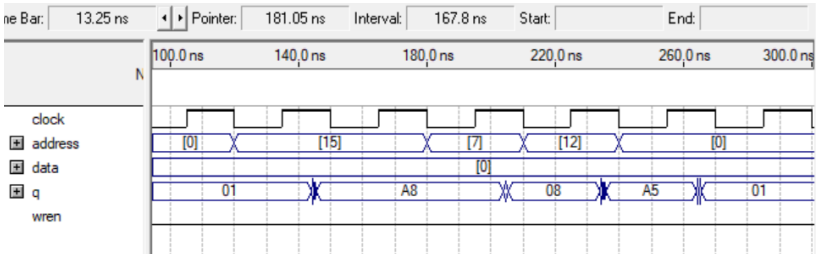


Fig 4 : Simulation de la mémoire seul.

Simulation pour afficher les bonnes données à partir du compteur.

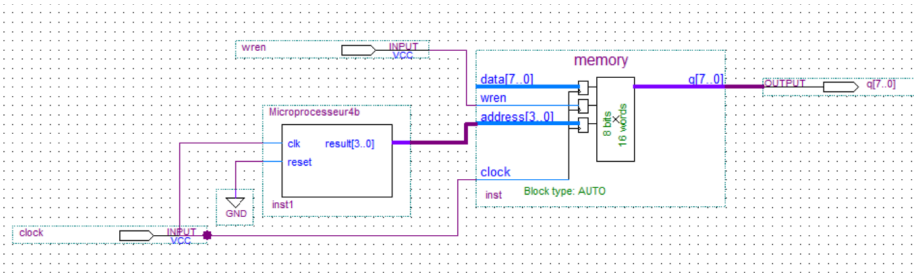


Fig 5 : Bloc compteur et mémoire.

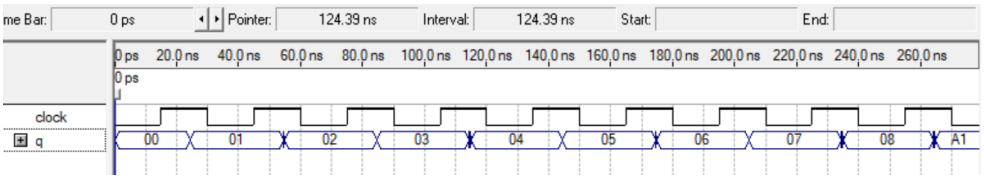


Fig 6 : Simulation de la mémoire avec le compteur.

3. Décodeur d'instructions (IR)

Objectif du décodeur :
Le décodeur sépare les 8 bits de la mémoire en deux parties :

- Les 4 bits les plus significatifs (MSB) sont l'instruction (ex. : "ajouter").
- Les 4 bits les moins significatifs (LSB) sont les données associées (ex. : "3").

Choix de la structure du code :
Un processus est utilisé pour surveiller les signaux d'horloge et d'activation (EN). Lorsque le décodeur est activé, il lit les 8 bits en entrée et divise l'information.

Choix des types de données :

- `std_logic_vector` : Utilisé pour représenter les 4 bits d'instruction et les 4 bits de données. Cela reflète directement la nature binaire des données dans la mémoire.
- `ZZZZ` : Lorsque le décodeur n'est pas activé (`EN=0`), les sorties sont mises en haute impédance (Z). Cela signifie qu'elles ne transmettent rien, pour éviter des conflits si un autre composant utilise le même bus.

Justification simplifiée : Pensez au décodeur comme un filtre dans une machine à café : il sépare le café (instruction) des grains (données).

Simulation :

- Fonctionnement validé : L'instruction et les données sont correctement décodées.

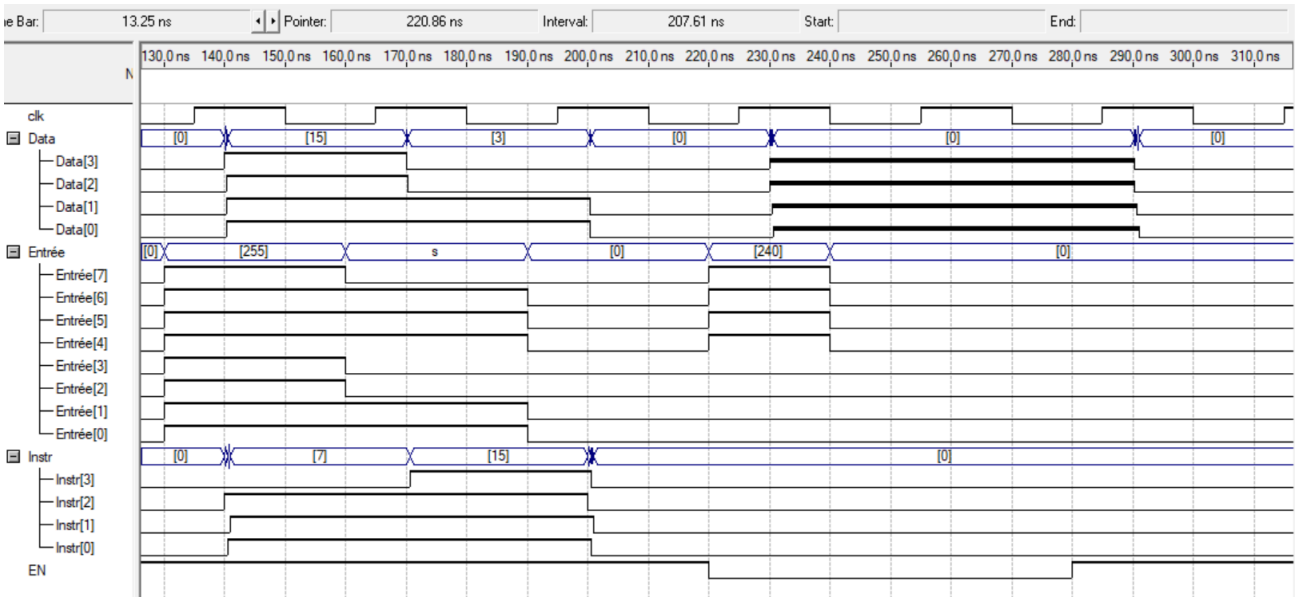


Fig 7 : Simulation du décodeur d'instructions

- Intégration avec le compteur et la mémoire testée avec succès.

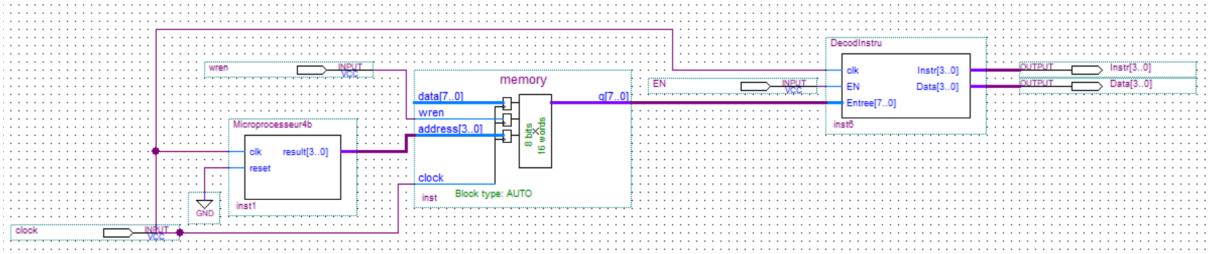


Fig 8 : Bloc compteur, mémoire et DecodInstru.

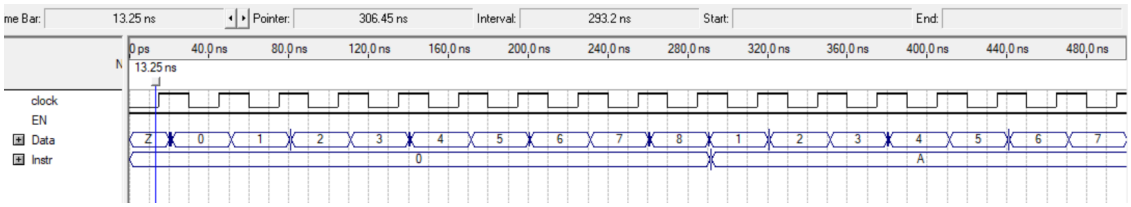


Fig 9 : Simulation d'ensemble compteur, mémoire et DecodInstru.

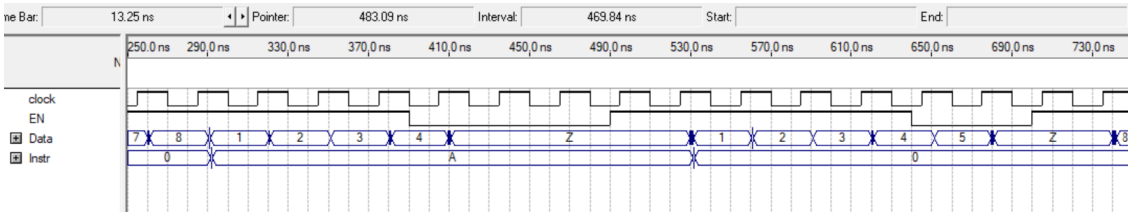


Fig 10 : Simulation d'ensemble compteur, mémoire et DecodInstru avec le signal EN (enable).

4. Accumulateurs (AccuA et AccuB)

Objectif des accumulateurs :
Les accumulateurs sont des petits blocs mémoire qui stockent temporairement les données nécessaires aux calculs. Ils servent à manipuler et transmettre les valeurs à l'ALU pour les opérations (addition, soustraction, etc.).

Choix de la structure du code :
L'utilisation d'un processus permet de surveiller les signaux d'horloge et de contrôle (`EnableA`, `load`), garantissant que les données sont correctement stockées ou transférées.

Choix des variables :

- **`data`** : Une variable interne représente les données en mémoire de l'accumulateur.
- **`std_logic_vector`** : Utilisé pour communiquer avec d'autres composants via le bus, car les systèmes numériques fonctionnent en binaire.
- **Haute impédance (`ZZZZ`)** : Lorsque l'accumulateur n'interagit pas avec le bus, il ne doit pas transmettre de données pour éviter des conflits.

Analogie simple : Les accumulateurs fonctionnent comme des notes autocollantes temporaires : vous écrivez un chiffre dessus, l'utilisez pour un calcul, puis vous pouvez le réutiliser ou l'effacer.

Simulation :

- Attention : La gestion du bus interne en haute impédance est assurée pour éviter les conflits.

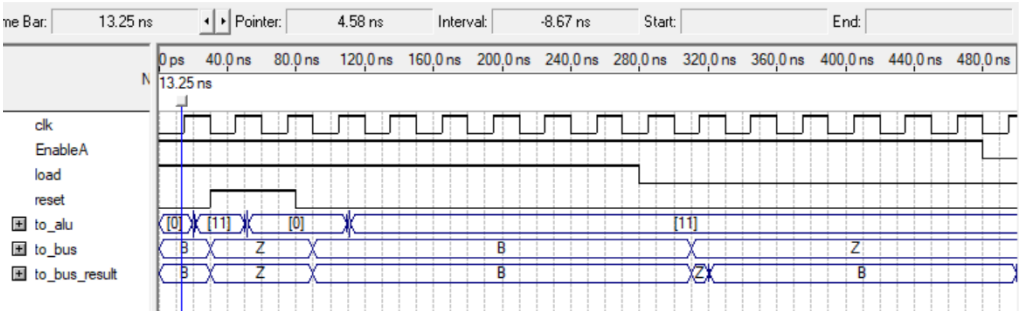


Fig 11 : Simulation de l'accumulateur.

5. Unité arithmétique et logique (ALU)

Objectif de l'ALU :
L'ALU est le "cerveau mathématique" du microprocesseur. Elle additionne ou soustrait des nombres et signale des informations comme un dépassement de capacité (carry) ou un résultat négatif.

Choix de la structure du code :
L'ALU surveille ses entrées (AccuA et AccuB) et utilise un signal de contrôle (`selct`) pour choisir entre addition ou soustraction.

Choix des variables :

- `std_logic_vector` : Utilisé pour recevoir et transmettre les données, car les accumulateurs travaillent en binaire.
- **Drapeaux** :
    - `C` : Signale un dépassement si le résultat est supérieur à 15 (plus grand que 4 bits).
    - `N` : Indique si le résultat est négatif après une soustraction.

Analogie simple : L'ALU est comme une calculatrice avec deux boutons principaux (addition et soustraction). Elle affiche aussi des informations supplémentaires comme "Erreur" (carry) ou "Négatif" (N).

Simulation :

- Addition, soustraction, et gestion des drapeaux validées.
- Overflow (C) et résultats négatifs (N) correctement détectés.

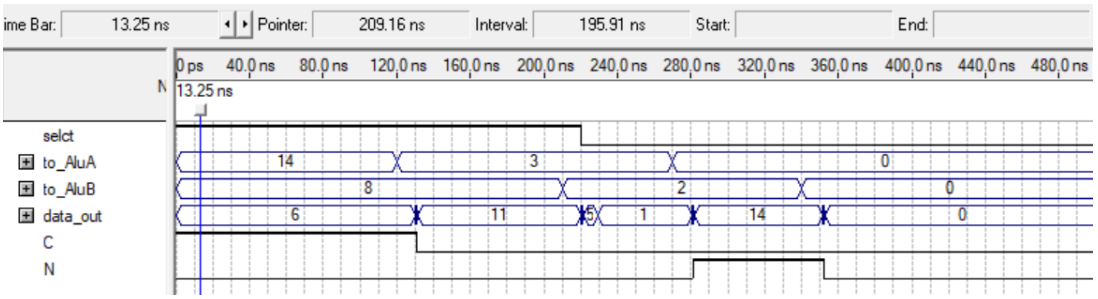


Fig 12 : Simulation de l'unité arithmétique.

- Avec le signale Enable Alu :

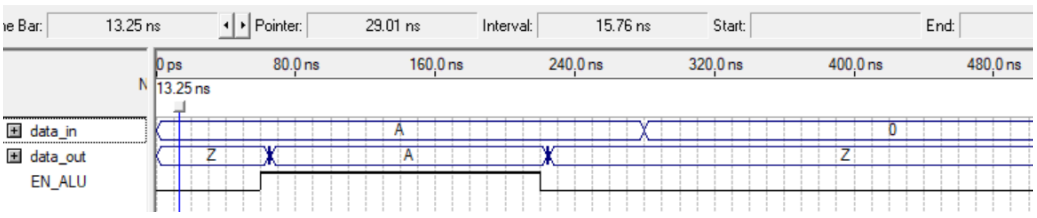


Fig 13 : Simulation de l'unité arithmétique avec le signal enable.

6. Le générateur de phase

Objectif du générateur de phase :
Le générateur de phase est un composant clé qui divise le fonctionnement du microprocesseur en quatre étapes distinctes (ou phases). Ces phases sont utilisées pour séquencer l'exécution des instructions et coordonner les différents blocs du microprocesseur.

Justifications des choix techniques :

1. **Structure en processus** :
   Le processus surveille l'horloge (`clk`) pour générer successivement les phases. Cette approche garantit que chaque phase est synchronisée avec l'horloge principale.
2. **Variable `cnt` pour compter les phases** :
   - Un entier (`integer range 0 to 4`) est utilisé pour représenter le cycle des phases.
   - La limite de 4 permet de revenir automatiquement à la phase 1 après la phase 4.
3. **`std_logic_vector` pour la sortie `phase`** :
   - Les phases sont codées sur 3 bits (`std_logic_vector (2 downto 0)`), car 4 phases nécessitent au moins 2 bits (mais 3 sont utilisés pour respecter la simplicité d'extension future).

Description du fonctionnement :

1. Lors d'un front descendant de l'horloge (`clk='0'`), le générateur :
   - Passe à la phase suivante si le signal `reset` est actif (`reset='1'`).
   - Revient à l'état initial (phase "000") si le signal `reset` est inactif.
2. Le compteur `cnt` incrémente les phases successives :
   - **Phase 1** : Initialisation des données.
   - **Phase 2** : Chargement de l'instruction dans les registres internes.
   - **Phase 3** : Exécution partielle de l'instruction (première étape).
   - **Phase 4** : Finalisation de l'instruction (seconde étape).
3. Les sorties du générateur (codes binaires pour les phases) :
   - 001 : Phase 1, 010 : Phase 2, 011 : Phase 3, 100 : Phase 4.
   - Une valeur par défaut "000" est activée en dehors des phases valides.

Exemple concret :
Le générateur de phase est comme un chef d'orchestre qui indique quand chaque instrument (bloc) doit jouer, s'assurant que tout est bien synchronisé.

Résultats de simulation :

- Les quatre phases sont générées dans l'ordre prévu.

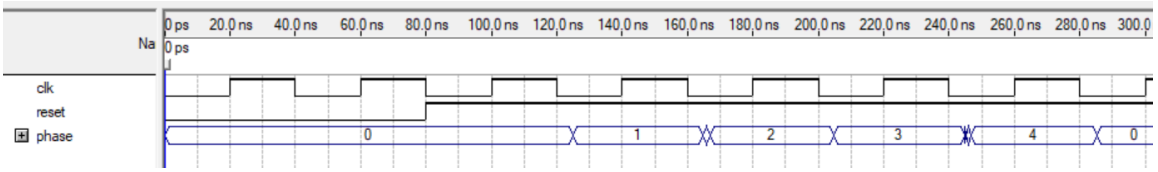


Fig 14 : Simulation du générateur de phase.

- Le générateur répond correctement au signal de remise à zéro (`reset`).

7. Le bloc Micro instruction

Objectif du bloc MI :
Le bloc de micro-instructions (MI) contrôle les signaux nécessaires pour exécuter une instruction en séquence. En fonction de la phase et de l'instruction en cours, il active ou désactive les composants du microprocesseur.

Justifications des choix techniques :

1. **Structure en processus avec `case` :**
    - Le processus est surveillé par les phases pour s'assurer que les signaux appropriés sont générés au bon moment.
    - L'utilisation de `case` pour les instructions et les phases simplifie la logique en organisant les actions de manière structurée.
2. **Choix des signaux (`std_logic`)** :
    - Chaque sortie (`EnableInstr`, `LoadA`, etc.) est un signal binaire (`std_logic`) qui active ou désactive un composant précis. Cela correspond au fonctionnement matériel des FPGA.
3. **Hiérarchie des phases** :
    - Les deux premières phases (fetch) sont communes à toutes les instructions, elles servent à charger l'instruction depuis la mémoire et à la transmettre au décodeur.
    - Les phases suivantes dépendent de l'instruction, comme l'addition (ADD) ou la soustraction (SUB).

Description du fonctionnement :

1. **Phase 1 ("001")** :
    - Le MI génère les signaux pour lire une instruction depuis la mémoire (`ReadMem='1'`) et la charger dans le registre d'instruction (`LoadInstr='1'`).
2. **Phase 2 ("010")** :
    - Le compteur de programme est incrémenté (`clk_PC='1'`).
    - Le décodeur d'instructions est activé pour extraire les bits correspondant à l'instruction et aux données (`EnableInstr='1'`).
3. **Phases spécifiques aux instructions (3 et 4)** :
    - Par exemple, pour l'instruction ADD (`instruction="0001"`) :

        - **Phase 3 ("011")** : La donnée est chargée dans l'AccuB (`LoadB='1'`), prête pour l'opération.
        - **Phase 4 ("100")** : L'opération d'addition est effectuée par l'ALU (`EnableAlu='1'`), et le résultat est stocké dans l'AccuA (`LoadA='1'`).
    - Pour les instructions NOP, SUB, ou GET INPUT, des signaux similaires sont activés selon leurs besoins spécifiques.
4. **Signal par défaut :**
    - En dehors des phases valides, tous les signaux sont désactivés (`'0'`) pour éviter des comportements imprévus.

Exemple concret :
Le bloc MI est comme un panneau de contrôle automatique qui active les machines (composants) au bon moment selon le type de tâche (instruction) à exécuter.

Résultats de simulation :

- Les signaux générés par le MI correspondent correctement aux besoins de chaque phase et instruction.

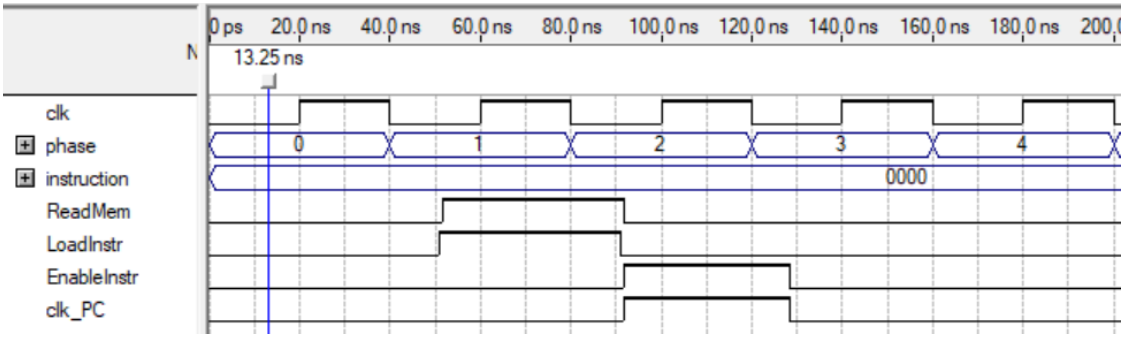


Fig 15 : Simulation du MI instruction "NOP".

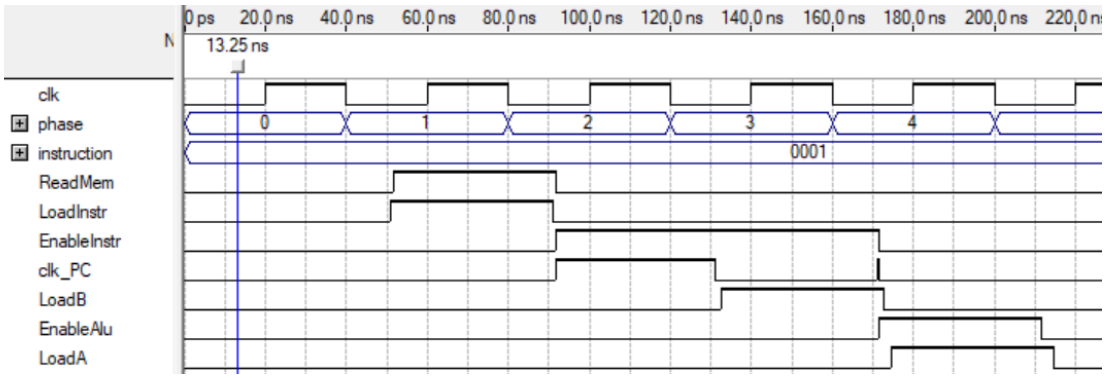


Fig 16 : Simulation du MI instruction "ADD" et "SUBB".

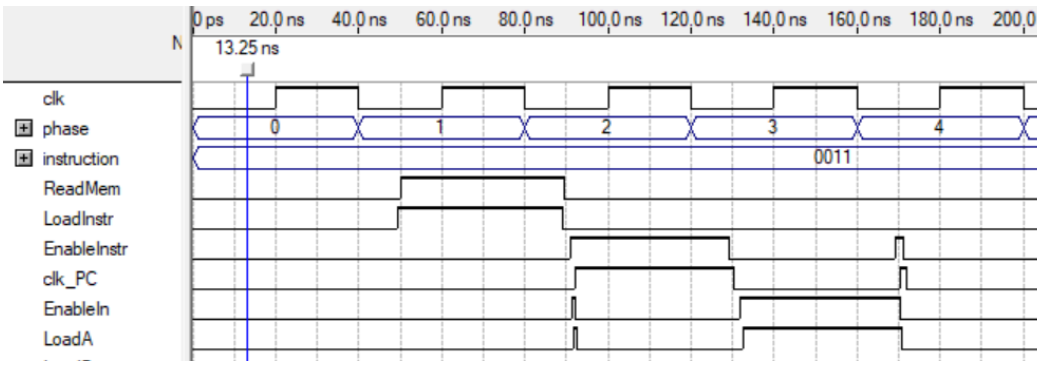


Fig 17 : Simulation du MI instruction “IN”.

- Cela valide que les composants s’activent et interagissent de manière fluide dans les simulations (par exemple, ADD charge les bonnes données dans l’ALU au moment précis).

Conclusion

Le développement du microprocesseur 4 bits a permis de couvrir plusieurs aspects de la conception numérique. Voici les principaux points à retenir :

1. **Structure modulaire** :
   Chaque composant (PC, PM, IR, ALU, Accumulateurs) a été conçu séparément, testé et intégré dans l'architecture globale. Cette approche modulaire facilite la conception, le débogage et la validation.
2. **Simulations réussies** :
   Les simulations montrent que toutes les instructions de base (ADD, SUB, IN, OUT) fonctionnent comme prévu. L'ALU gère correctement les opérations et les drapeaux (carry, négatif, zéro), tandis que les accumulateurs assurent un stockage et une transmission fiables des données.
3. **Choix des types de variables** :
   Les variables et types ont été sélectionnés pour refléter fidèlement les contraintes matérielles. Par exemple, `std_logic_vector` a été utilisé pour la communication binaire, et les signaux `ZZZZ` pour éviter les conflits sur le bus interne.
4. **Analyse temporelle et matériel** :
   Une analyse temporelle a permis de garantir la synchronisation des composants avec une horloge ajustée à une période de 20 ns. Le microprocesseur peut être implémenté sur des FPGA comme le MAX7000S ou le Stratix, bien que la taille de la mémoire et les besoins en calcul soient limités par l'architecture simple.
5. **Perspectives d'amélioration** :
   - Ajout d'instructions plus complexes (JUMP, BRNE, BRZ).
   - Optimisation de l'ALU pour prendre en charge des opérations supplémentaires.
   - Étendre la mémoire pour exécuter des programmes plus longs.

Impact pédagogique :
Ce projet a permis d'appréhender les concepts de base des microprocesseurs, de comprendre les interactions entre les composants et de développer des compétences en simulation numérique avec VHDL. La validation fonctionnelle du microprocesseur ouvre la voie à des implémentations physiques sur FPGA ou d'autres plateformes.

En conclusion, la conception d'un microprocesseur 4 bits, bien que simple, représente un excellent cas d'étude pour comprendre les principes fondamentaux des systèmes embarqués. Ce travail a non seulement permis de maîtriser les concepts théoriques, mais également de les appliquer dans un cadre pratique.

ANNEXE : Code VHDL

1. Compteur de programme (PC)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Microprocesseur4b is
   port (clk, reset : in bit;
         result : out std_logic_vector (3 downto 0));
end entity Microprocesseur4b;

architecture beh of Microprocesseur4b is
begin
   process(clk)
      variable decpt : integer range 0 to 15 := 0;
   begin
      if (clk'event and clk = '1') then
         if reset = '0' then
            if decpt = 15 then
               decpt := 0;
            else
               decpt := decpt + 1;
            end if;
         else
            decpt := 0;
         end if;
         result <= std_logic_vector(to_unsigned(decpt, 4));
      end if;
   end process;
end beh;
```

2. Mémoire de programme (PM)

3. Décodeur d’instructions (IR)

```
library ieee;
use ieee.std_logic_1164.all;

entity DecodInstru is
   port (clk, EN : in bit;
         Entrée : in std_logic_vector (7 downto 0);
         Instr, Data : out std_logic_vector (3 downto 0));
end entity DecodInstru;

architecture beh of DecodInstru is
begin
```

```
    process(clk)
    begin
        if (clk'event and clk = '1') then
            if EN = '1' then
                Instr <= Entrée(7 downto 4);
                Data <= Entrée(3 downto 0);
            else
                Instr <= "ZZZZ";
            end if;
        end if;
    end process;
end beh;
```

4. Accumulateurs (AccuA et AccuB)

```
library ieee;
use ieee.std_logic_1164.all;

entity Accu is
    port (clk, reset, EnableA, load : in bit;
          to_bus : inout std_logic_vector (3 downto 0);
          to_alu : out std_logic_vector (3 downto 0));
end entity Accu;

architecture beh of Accu is
    signal data : std_logic_vector (3 downto 0) := "0000";
begin
    process(clk)
    begin
        if (clk'event and clk = '1') then
            if reset = '0' then
                if EnableA = '1' then
                    if load = '1' then
                        data <= to_bus;
                        to_alu <= to_bus;
                    else
                        to_bus <= data;
                    end if;
                else
                    to_bus <= "ZZZZ";
                end if;
            else
                to_bus <= "ZZZZ";
                to_alu <= "0000";
            end if;
        end if;
    end process;
end beh;
```

5. Unité arithmétique et logique (ALU)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Alu is
   port (selct : in bit;
         to_AluA, to_AluB : in std_logic_vector (3 downto 0);
         N, C : out std_logic;
         data_out : out std_logic_vector (3 downto 0));
end entity Alu;

architecture beh of Alu is
begin
   process (to_AluA, to_AluB, selct)
   begin
      if selct = '1' then -- Addition
         data_out <= std_logic_vector(unsigned(to_AluA) + unsigned(to_AluB));
         if (unsigned(to_AluA) + unsigned(to_AluB) > 15) then
            C <= '1';
         else
            C <= '0';
         end if;
      else -- Soustraction
         data_out <= std_logic_vector(unsigned(to_AluA) - unsigned(to_AluB));
         if (unsigned(to_AluA) < unsigned(to_AluB)) then
            N <= '1';
         else
            N <= '0';
         end if;
      end if;
   end process;
end beh;
```

6. Le générateur de phase

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gene_phase is
port (clk, reset : in std_logic;
                  phase : OUT std_logic_vector (2 downto 0));
end entity gene_phase;

architecture beh of gene_phase is
```

```
Begin
	process(clk)
	variable cnt : integer range 0 to 4;
	Begin
		if (clk'event and clk = '0') then
			if reset = '1' then
				if (cnt>3) then
					cnt := 0;
				else
					cnt := cnt +1;
				end if;
			else
				cnt := 4;
				phase <= "000";
			End if;

			case cnt is
				when 0 => phase <= "001"; --Phase 1

				when 1 => phase <= "010"; --Phase 2

				when 2 => phase <= "011"; --Phase 3

				when 3 => phase <= "100"; --Phase 4

				when others => phase <= "000";
			End case;
		End if;
	End process;
End beh;
```

7. Le bloc Micro instruction

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mi is
port (clk : in std_logic;
		phase : in std_logic_vector (2 downto 0);
		instruction : in std_logic_vector (3 downto 0);
		clk_PC, ReadMem, LoadInstr, EnableInstr, LoadB, LoadA, EnableAlu,
EnableIn : out std_logic);
end entity mi;

architecture beh of mi is
Begin
```

```
process(phase)
Begin
        if phase = "000" or phase = "101" or phase = "110" or phase = "111"
then
                ReadMem <= '0';
                LoadInstr <= '0';
                clk_PC <= '0';
                EnableInstr <= '0';
                LoadB <= '0';
                EnableAlu <= '0';
                LoadA <= '0';
                EnableIn <= '0';
        End if;

        if phase = "001" then
                ReadMem <= '1';
                LoadInstr <= '1';
                clk_PC <= '0';
                EnableInstr <= '0';
                LoadB <= '0';
                EnableAlu <= '0';
                LoadA <= '0';
                EnableIn <= '0';
        End if;

        if phase = "010" then
                clk_PC <= '1';
                EnableInstr <= '1';
                ReadMem <= '0';
                LoadInstr <= '0';
                LoadB <= '0';
                EnableAlu <= '0';
                LoadA <= '0';
                EnableIn <= '0';
        End if;

        case instruction is
                when "0000" => if phase = "011" or phase = "100" then -- NOP
                                        ReadMem <= '0';
                                        LoadInstr <= '0';
                                        clk_PC <= '0';
                                        EnableInstr <= '0';
                                        EnableIn <= '0';
                                End if;

                when "0001" => if phase = "011" then -- ADD
                                        ReadMem <= '0';
                                        LoadInstr <= '0';
```

```
                clk_PC <= '0';
                EnableInstr <= '1';
                LoadB <= '1';
                EnableAlu <= '0';
                LoadA <= '0';
                EnableIn <= '0';
            End if;
            if phase = "100" then
                ReadMem <= '0';
                LoadInstr <= '0';
                clk_PC <= '0';
                EnableInstr <= '0';
                LoadB <= '0';
                EnableAlu <= '1';
                LoadA <= '1';
                EnableIn <= '0';
            End if;

        when "0010" => if phase = "011" then -- SUB
                ReadMem <= '0';
                LoadInstr <= '0';
                clk_PC <= '0';
                EnableInstr <= '1';
                LoadB <= '1';
                EnableAlu <= '0';
                LoadA <= '0';
                EnableIn <= '0';
            End if;
            if phase = "100" then
                ReadMem <= '0';
                LoadInstr <= '0';
                clk_PC <= '0';
                EnableInstr <= '0';
                LoadB <= '0';
                EnableAlu <= '1';
                LoadA <= '1';
                EnableIn <= '0';
            End if;

        when "0011" => if phase = "011" then -- GET INPUT
                ReadMem <= '0';
                LoadInstr <= '0';
                clk_PC <= '0';
                EnableInstr <= '0';
                LoadB <= '0';
                EnableAlu <= '0';
                LoadA <= '1';
                EnableIn <= '1';
```

```
                                        End if;
                                        if phase = "100" then
                                                ReadMem <= '0';
                                                LoadInstr <= '0';
                                                clk_PC <= '0';
                                                EnableInstr <= '0';
                                                LoadB <= '0';
                                                EnableAlu <= '0';
                                                LoadA <= '0';
                                                EnableIn <= '0';
                                        End if;

                                when others => ReadMem <= '0';
                                                LoadInstr <= '0';
                                                clk_PC <= '0';
                                                EnableInstr <= '0';
                                                LoadB <= '0';
                                                EnableAlu <= '0';
                                                LoadA <= '0';
                                                EnableIn <= '0';
                        End case;
        End process;
End beh;
```