Lab 5 - Architecture de processeur

Université de Cergy-Pontoise

Ce sujet de TP sert également de sujet de projet pour le module d'architecture.

La Figure 1 montre l'architecture d'un système numérique composé de registres 16 bits, un multiplexeur, un additionneur/soustracteur et une unité de contrôle (implémentée comme une FSM). Les données entrantes passent par le bus de 16 bits nommé DIN et peuvent être rangées dans un des registres ($R0, \ldots, R7$ ou A) en sélectionnant le chemin approprié dans le multiplexeur. Ce mutliplexeur permet aussi de déplacer les données d'un registre vers un autre via son bus de sortie.

Pour réaliser une opération dans l'unité arithmétique et logique (Addition ou soustraction dans un premier temps) le multiplexeur doit être commandé par l'unité de contrôle pour positionner la première opérande sur le bus pour la placer dans le registre *A*. Puis, au cycle suivant, une seconde opérande doit être positionnée sur le bus. L'opération est alors réalisée entre le registre *A* et le contenu du bus. Le résultat est rangé dans le registre *G* et peut ensuite être déplacé si cela est nécessaire.

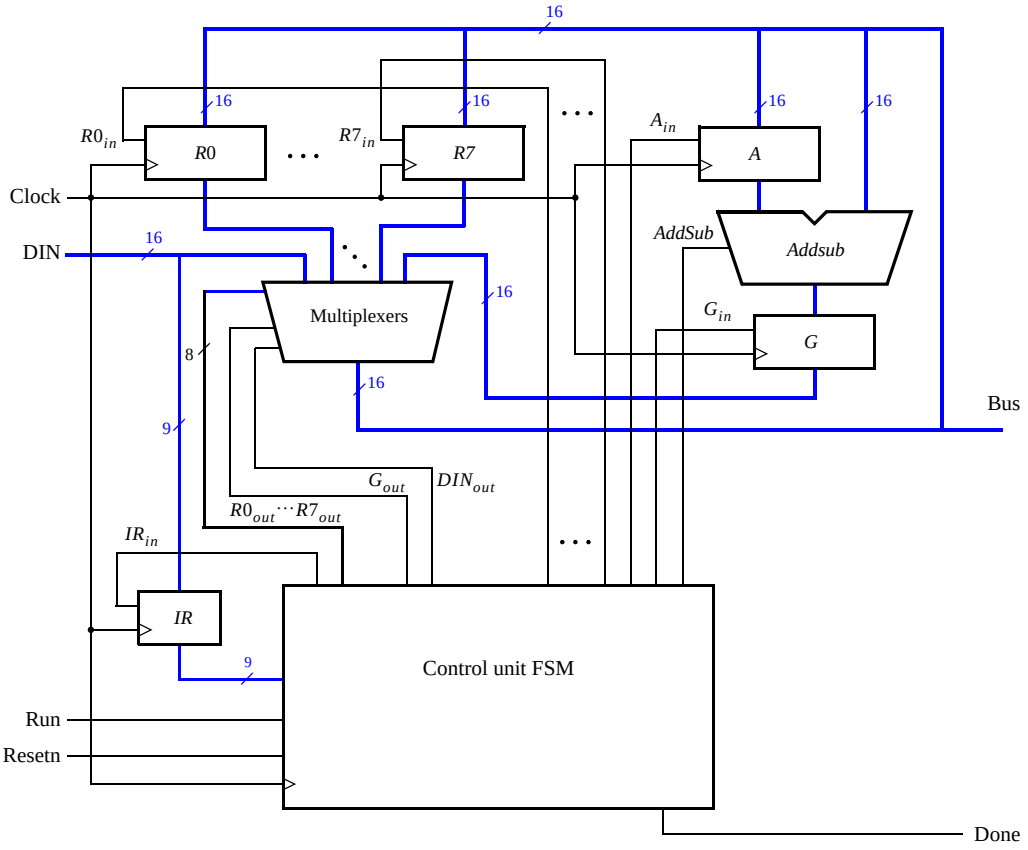

Figure 1: Architecture de processeur à accumulateur.

Les signaux de commande sortant de l'unité de contrôle permette de sélectionner les opérations réalisées par chaque composant. Par exemple, si les signaux $R0_{out}$ et A_{in} sont actifs, alors c'est le registre $R0$ qui écrira sur le bus pour ranger son contenu au cycle suivant dans le registre *A*.

La configuration de l'ensemble des signaux de commande à un instant donné correspond à l'exécution d'une instruction du processeur. La Table 1 liste les instructions qui seront supportées par cette première version du processeur. Dans la syntaxe assembleur, l'écriture RX ← [RY] signifie que le contenu du registre RY est chargé dans le registre RX (sans modifié le contenu de RY), ce qui correspond à une instruction move **mv**. Une instruction **mvi** (move immediate) permet de spécifier une constante sur 16 bits à placer dans le registre RX : RX ← D.

Opération	Fonction réalisée
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 1. Instructions réalisée par le processeur.

A chaque cycle, une nouvelle instruction est donc chargée depuis le bus DIN dans le registre *IR* de 9 bits. Ces instructions sont encodées suivant 2 formats.

Dans le format R (registre), les 9 bits sont organisées de la manière suivante : IIIXXXYYY où III représente le code opération (*codop*) de l'instruction à réaliser, XXX code le numéro du registre RX et YYY code le registre RY. Trois bits sont utilisées pour le *codop* pour convenir aux futures extensions du processeur.

Dans le format I (Immédiat), les 9 bits sont organisées de la manière suivante : IIIXXX000. En effet, le registre RY n'est pas utilisé mais un second mot de 16-bits doit être lu sur le port DIN pour connaître la valeur de la constante (immédiate) à placer dans RX.

Les instructions utilisant l'UAL sont codées selon le format R mais utilisent plusieurs cycles pour s'exécuter à cause des multiples transferts à opérer sur le bus. C'est alors la machine d'états de l'unité de contrôle qui dirige ces opérations de transfert à chaque cycle. Une opération n'est exécutée que si le signal d'entrée *Run* est actif, et le processeur indique qu'une instruction est terminée en activant le signal de sortie *Done*. La Table 2 indique quelles sont les signaux de commande à activer à chaque cycle pour réaliser les instructions de la Table 1. Une étape préalable (non représentée) à toute instruction est l'étape T_0 pendant laquelle le signal IR_{in} est activée pour stocker la nouvelle instruction à décoder.

	T_1	T_2	T_3
(**mv**): I_0	RY_{out}, RX_{in}, *Done*		
(**mvi**): I_1	DIN_{out}, RX_{in}, *Done*		
(**add**): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
(**sub**): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, *AddSub*	G_{out}, RX_{in}, *Done*

Table 2. Signaux de commande positionnés à chaque étape d'exécution des instructions.

Partie I

Réaliser l'implémentation de ce processeur en VHDL :

1. Créer un nouveau projet Quartus-II.

2. Réaliser le code VHDL du processeur en réutilisant les composants déjà réalisés dans les TPs précédents. Le squelette de code VHDL de l'entité est donné dans la figure $2a$.

3. Utilisez la simulation fonctionnelle pour vérifier le comportement du circuit. Un exemple de simulation est donné dans la Figure 3. Il montre que la valeur $(2000)_{16}$ est chargée dans le registre *IR* depuis le port *DIN* à la date 30 ns. Les 9 bits de poids fort (gauche) correspondent à une instruction **mvi** R0,#D, où $D = 5$ est chargé dans $R0$ sur front montant à 50 ns. Puis la simulation montre la lecture de l'instruction **mv** R1,R0 à 90 ns, **add** R0,R1 à 110 ns, et **sub** R0,R0 à 190 ns.

4. Créer un nouveau projet Quartus-II ainsi qu'une nouvelle entité Top dans laquelle vous instancierez votre processeur et dont vous connecterez les ports aux périphériques de la carte: les switchs SW_{15-0} pour écrire sur le bus *DIN* et le switch SW_{17} pour la commande *Run*. Le bouton KEY_0 sera utilisé pour configurer le *Resetn* et KEY_1 pour l'horloge *Clock*. Connectez le bus de sortie du processeur bus aux $LEDR_{15-0}$ et le drapeau *Done* à $LEDR_{17}$.

5. Compiler le design en incluant l'affectation des pins de la carte, puis vérifier le fonctionnement du processeur manuellement.

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY proc IS
   PORT ( DIN                 : IN     STD_LOGIC_VECTOR(15 DOWNTO 0);
          Resetn, Clock, Run  : IN     STD_LOGIC;
          Done                : BUFFER STD_LOGIC;
          BusWires            : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0));
END proc;

ARCHITECTURE Behavior OF proc IS
   . . . declare components
   . . . declare signals
   TYPE State_type IS (T0, T1, T2, T3);
   SIGNAL Tstep_Q, Tstep_D: State_type;
   . . .
BEGIN
   I <= IR(1 TO 3);
   decX: dec3to8 PORT MAP (IR(4 TO 6), ...);
   decY: dec3to8 PORT MAP (IR(7 TO 9), ...);
```

Figure $2a$. Squeleton VHDL du processeur.

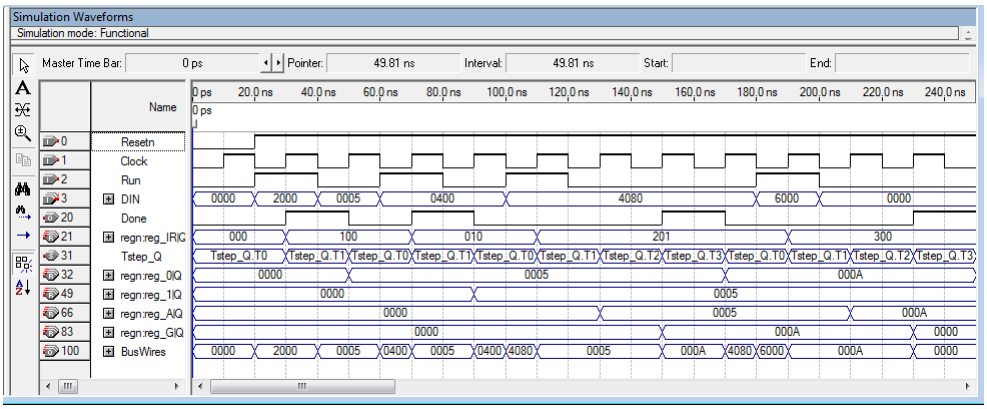

Figure 3. Simulation du processeur.

Partie II - Ajout de la mémoire

Dans cette partie, vous allez ajouter une mémoire au processeur pour éviter de positionner manuelle les instructions. Le circuit correspondant est représenté dans la Figure 4, Le compteur est utilisé pour incrémenter l'adresse. Dans les processeurs il est appelé Compteur Ordinal (ou Program Counter). Pour simplifier le test du circuit, deux horloges sont utilisées : *PClock* et *MClock*, le premier pour le Processeur, le second pour la Mémoire.

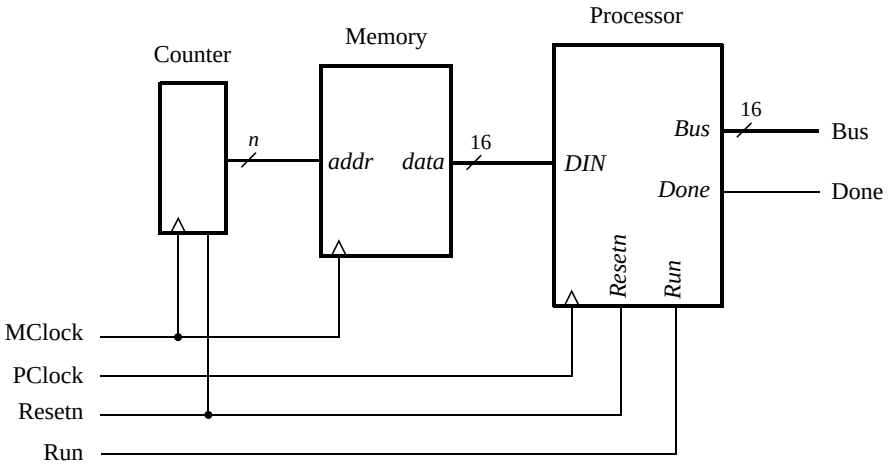

Figure 4. Mémoire et processeur.

1. Créer un nouveau projet.

2. Pour générer la mémoire, nous allons utiliser un bloc pré-existant. Utilisez l'outil Quartus II MegaWizard Plug-In Manager qui permet de générer le code d'un composant de la librairie Altera (library of parameterized modules, LPMs). Le composant à choisir se trouve dans la catégorie *Memory Compiler* et est appelée *ROM: 1-PORT*. Suivez les écrans de configuration pour générer une mémoire simple port de lecture dont les données sont sur 16-bits et pouvant mémoriser 32 mots. La Page 4 du wizard est indiquée dans la Figure 5. Puisque cette mémoire n'a pas de port d'écriture elle est appelée *synchronous read-only memory (synchronous ROM)*.

   Pour placer les instructions dans la mémoire, vous devez spécifier un contenu initial de la mémoire. Cela se précise dans le wizard par un fichier appelé *memory initialization file (MIF)*. L'écran de configuration est représenté dans la Figure 6. Utilisez l'aide de Quartus-II pour créer un fichier *inst_mem.mif* qui dispose de suffisamment d'instructions pour tester le circuit.

3. Utilisez la simulation fonctionnelle pour vérifier que le processeur lit bien les instructions de la mémoire.

4. Utilisez ensuite le switch SW_{17} pour commande le port *Run*, le bouton KEY_0 pour le *Resetn*, KEY_1 pour *MClock*, et KEY_2 pour *PClock*. Comme précédemment le bus de sortie sera relié aux $LEDR_{15-0}$ et le signal *Done* à $LEDR_{17}$.

5. Compilez et testez le fonctionnement sur la carte. Les 2 horloges sont séparées pour éviter de lire plusieurs instructions lors de l'exécution d'instructions multi-cycles (ALU).

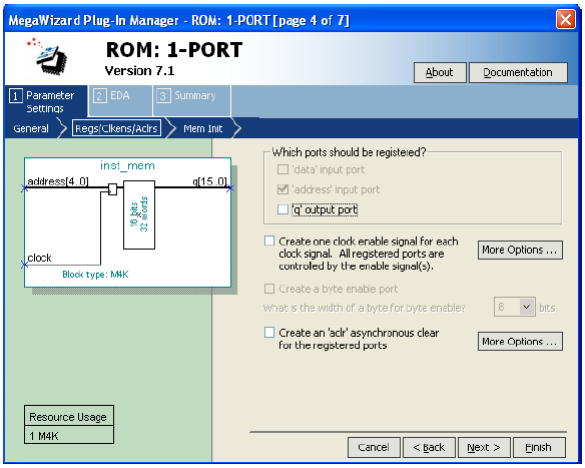

Figure 5. Configuration 1-PORT.

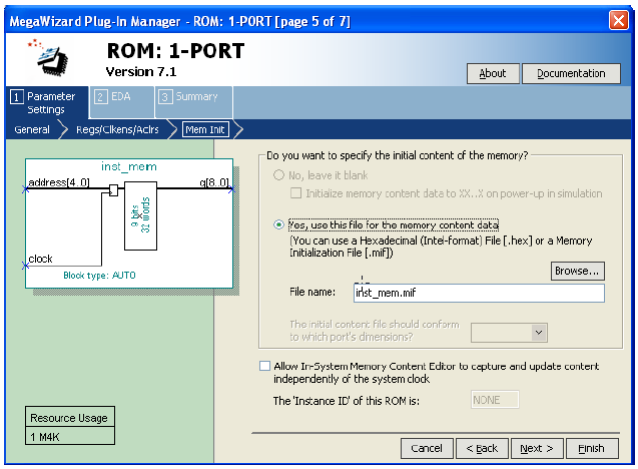

Figure 6. Spécification d'un fichier d'initialisation (MIF).

Partie III - Extensions
De manière à apporter vos propres modifications au processeur, il vous sera demander de réfléchir et de réaliser l'une des extensions suivantes :

- Ajout d'instruction de type I pour le chargement (**load**) et le rangement (**store**) en mémoire qui permettent respectivement de charger et de ranger une donnée depuis ou vers la mémoire. Les opérandes de ces instructions sont un regstre et une adresse mémoire codée comme un immédiat de 16 bits.
- Ajout d'instruction de type R pour les opérations logiques (AND, OR, NOT).
- Ajout d'instruction de type R pour la multiplication de deux nombres stockés dans des registres. Vous porterez attention à expliquer comment est traité le résultat dans les registres.
- Ajout d'un nouveau type d'instruction de type J pour réaliser des sauts dans l'exécution des instructions. Ce type d'instructions n'a qu'une seule opérande : une adresse de saut sur 16 bits.
- Écriture d'un assembleur en Java, C ou C++ qui permet de générer directement le fichier MIF à partir d'une fenêtre de saisie du code assembleur.