Archi+: TP1

1 Architecture mono-bus

On considère le microprocesseur dont l'architecture est présentée Figure 1. Ce microprocesseur est composé des éléments suivants :

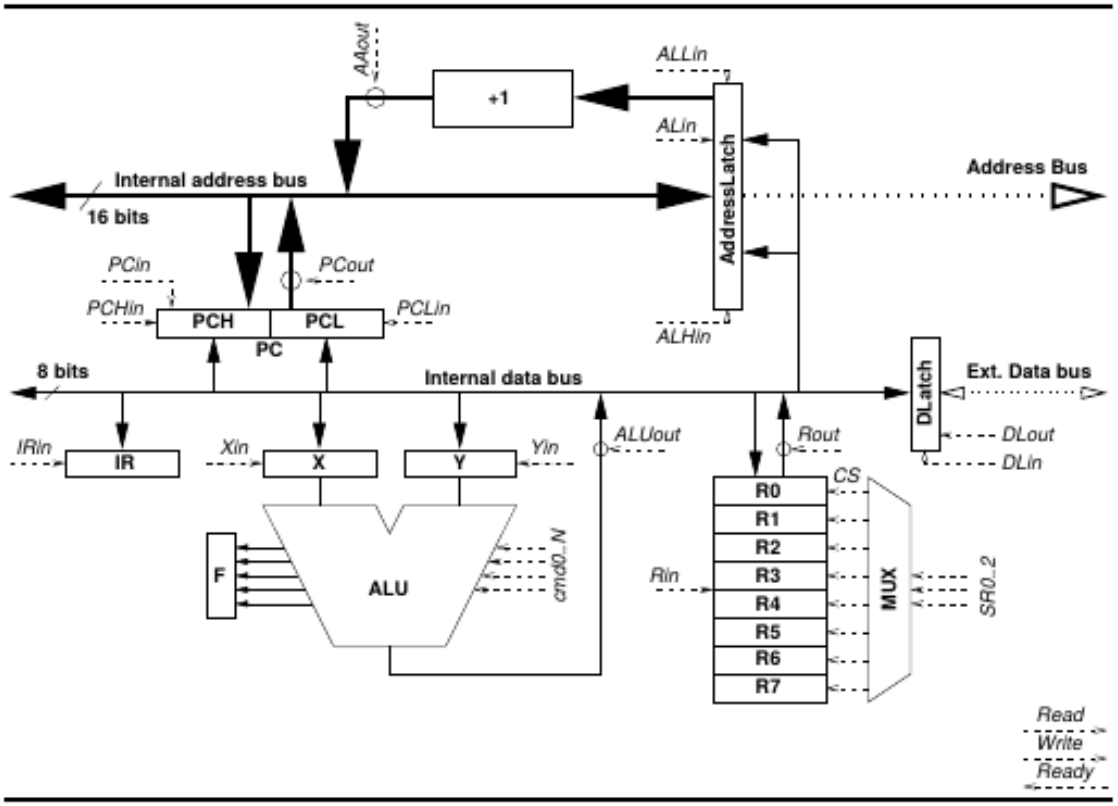

Figure 1: Architecture mono-bus

- **un bloc registres 8 bits** : nommés de $R0$ à $R7$, il s'agit de registres généraux 8 bits. Pour manipuler ce bloc, nous disposons d'une commande de sélection $SR0..2$ (*Select Register*) permettant de sélectionner l'un des huit registres comme illustré Tableau 1. Ceci se traduit par l'activation d'un unique CS (Chip Selector) de l'un des huit registres. Le signal *Rin* provoque le chargement dans le registre sélectionné de l'information disponible sur le bus. Le signal *Rout* provoque la mise sur le bus de l'information contenue dans le registre sélectionné.

SR2..0	Registre sélectionné
000	R0
001	R1
010	R2
011	R3
100	R4
101	R5
110	R6
111	R7

Table 1: Sélection dans le bloc registres

- **Un bloc ALU** : il est constitué d'une unité arithmétique et logique possédant un certain nombre de commandes $cmd0..N$ dont la taille est à déterminer. Un certain nombre de Flags (à déterminer) sont stockés dans un registre de flags appelé F. Les entrées de l'ALU sont les sorties de deux registres 8 bits X et Y. Ces registres peuvent charger l'information présente sur le bus de données par les signaux Xin et Yin. La sortie de l'ALU est relié au bus de données et est activées à l'aide du signal $ALUout$.

  Notons que les registres X et Y sont des registres internes et que le programmeur n'a pas accès à ces registres.

- **Un bloc gestion des adresses** : comprenant un bus interne d'adresse 16 bits, un registre *AddressLatch* 16 bits connecté au bus d'adresse externe, un registre PC (Programm Counter) 16 bits et un incrémenteur.

  Les signaux associés au registre *AddressLatch* sont au nombre de trois : $ALin$ qui permet de charger dans le registre l'information sur 16 bits présente sur le bus d'adresse interne, $ALLin$ qui permet de charger dans la partie basse du registre l'information sur 8 bits présente sur le bus de données interne et $ALHin$ qui permet de charger dans la partie haute du registre l'information sur 8 bits présente sur le bus de données interne.

  Les signaux associés au registre PC sont au nombre de quatre et sont similaires à ceux pour *AddressLatch* : $PCin$ qui permet de charger dans le registre l'information sur 16 bits présente sur le bus d'adresse interne, $PCLin$ qui permet de charger dans la partie basse du registre l'information sur 8 bits présente sur le bus de données interne, $PCHin$ qui permet de charger dans la partie haute du registre l'information sur 8 bits présente sur le bus de données interne et enfin le signal $PCout$ qui met sur le bus d'adresse interne l'information contenue dans PC.

  L'incrémenteur prend en entrée le contenu du registre *AddressLatch* et présente en sortie l'adresse suivante (incrémentation de 1). La sortie de l'incrémenteur est mise sur le bus d'adresse interne par le biais du signal $AAout$.

- **un bloc de contrôle** : constitué d'un registre d'instruction IR 8 bits et d'une unité de contrôle. L'unité de contrôle est chargée de générer les signaux nécessaire au fonctionnement du microprocesseur en exécutant le cycle :

  1. lire l'instruction ;
  2. décoder l'instruction ;
  3. exécuter l'instruction ;
  4. préparer l'instruction suivante.

  Dans le reste de l'exercice, on ignorera la phase 4. Le registre IR copie l'information positionnée sur le bus de données interne suivant le signal $IRin$.

- **un bloc gestion de la mémoire** : il comprend un registre de données *DLatch* (pour *Data Latch*) 8 bits qui copie la valeur du bus de données interne sur le signal *DLin*. Sur un signal *Read*, la mémoire met dans le registre *DLatch* la valeur de la case mémoire d'adresse *AddressLatch*. Sur un signal *Write*, la mémoire copie dans la case mémoire d'adresse *AddressLatch* l'information qui est dans *DLatch*. Lorsque la mémoire a terminé l'opération demandée (*Read* ou *Write*), elle positionne en retour le signal *Ready*.

2 Jeu d'instructions

On souhaite que notre microprocesseur dispose des instructions suivantes :

- `ST Rn, HHLL` — (*pour STore*) qui stocke la valeur du registre `Rn` en mémoire à l'adresse (sur 16 bits) `HHLL` ;
- `LD Rn, HHLL` — (*pour LoaD*) qui charge dans le registre `Rn` la valeur stockée en mémoire à l'adresse (sur 16 bits) `HHLL` ;
- `MV Rn, arg` — (*pour MoVe*) qui charge dans le registre `Rn` la valeur de l'argument "`arg`". L'argument est soit une valeur immédiate, soit un registre ;
- `JMP HHLL` — (*pour JuMP*) qui effectue un branchement à l'adresse (sur 16 bits) `HHLL` ;
- `ADD Rn, Rm` — (*pour ADD :-)*) qui additionne la valeur du registre `Rn` avec la valeur du registre `Rm` et stocke le résultat dans le registre `Rn` ;
- `SUB Rn, Rm` — (*pour SUBstract*) qui soustrait la valeur du registre `Rm` à la valeur du registre `Rn` et stocke le résultat dans le registre `Rn` ;
- `AND Rn, Rm` — qui fait la conjonction bit à bit des deux registres et range le résultat dans `Rn` ;
- `DEC Rn` — (*pour DECrement*) qui soustrait 1 à la valeur de `Rn` et stocke le résultat dans `Rn` ;
- `INC Rn` — (*pour INCrement*) qui ajoute 1 à la valeur de `Rn` et stocke le résultat dans `Rn` ;
- `NOT Rn` — qui inverse bit à bit le registre `Rn` et qui y stocke le résultat ;
- `JZ HHLL` — (*pour Jump if Zero*) qui effectue un saut à l'adresse 16 bits `HHLL` si l'opération précédente à donné un résultat nul ;
- `JC HHLL` — (*pour Jump if Carry*) qui effectue un saut à l'adresse 16 bits `HHLL` si l'opération précédente à engendré une retenue ;
- `NOP` — (*pour No Operation*) qui est une instruction qui ne fait rien...

Question 2.1 : *Proposez un format hexadécimal pour ces instructions en minimisant la taille des instructions (le minimum d'octets possibles). Toutes les opérations devront être de la forme* `CodeOpération Opérandes`.

Par exemple, l'instruction `JMP HHLL` *peut être codé sur trois octets :*

```
FF LL HH
```

Le code opération est l'octet `FF` *(255 en décimal) suivi de deux octets qui indiquent l'adresse où l'on désire faire le branchement.*
NB. Ici on choisit de mettre d'abord l'octet de poids faible. Ceci correspond à la notation "big endian" *(grand-boutiste en français). La notation qui consiste à mettre d'abord l'octet de poids fort est la notation* "little endian" *(petit-boutiste).*

- 0111 0011 codera `JMP RX0` ;
- 0111 10nn codera `ST R0, RXn` ;
- 0111 11nn codera `LD R0, RXn`.

Opération	Codage binaire	Commentaire
`NOP`	0000 0000	
`JMP HHLL`	0111 0000 - llll llll - hhhh hhhh	hhhh hhhh correspond à HH llll llll correspond à LL
`JZ HHLL`	0111 0001 - llll llll - hhhh hhhh	hhhh hhhh correspond à HH llll llll correspond à LL
`JC HHLL`	0111 0010 - llll llll - hhhh hhhh	idem
`JMP RX0`	0111 0011	
`ST R0, RXn`	0111 10nn	nn correspond au registre 16 bits concerné
`LD R0, RXn`	0111 11nn	idem
`ST Rn, HHLL`	0100 0nnn - llll llll - hhhh hhhh	nnn correspond au registre hhhh hhhh correspond à HH llll llll correspond à LL
`LD Rn, HHLL`	0100 1nnn - llll llll - hhhh hhhh	idem
`MV Rn, arg#`	0101 0nnn - aaaa aaaa	nnn correspond au registre aaaa aaaa correspond à l'argument
`DEC Rn`	0101 1nnn	nnn correspond au numéro de registre
`INC Rn`	0110 0nnn	nnn correspond au numéro de registre
`NOT Rn`	0110 1nnn	nnn correspond au numéro de registre
`ADD Rn, Rm`	100n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SUB Rn, Rm`	101n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`AND Rn, Rm`	110n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SWP Rn, Rm`	111n nmmm	0nn correspond au premier registre mmm correspond au deuxième registre
`MV Rn, Rm`	00nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre

Question 2.2 : *Écrire le programme assembleur effectuant une copie de la zone mémoire commençant à l'adresse stockée en mémoire à l'adresse 0100h dans la zone mémoire dont l'adresse est stockée en 0102h. La copie s'arrête dès que l'on a copié la valeur 00h. On considérera que les zones mémoires sont distinctes.*

L'écrire en hexadécimal en supposant que le programme commence à l'adresse 0200h.

Solution 2.2 :

1. en assembleur

```
        ; récupérer l'adresse de la première zone
        LD R2, 0100h
        LD R3, 0101h
        ; récupérer l'adresse de la deuxième zone
        LD R4, 0102h
        LD R5, 0103h
        ; commencer la boucle
boucle:
        LD R0, RX1
```

1. Objectif du TP

Le but est de réaliser un simulateur basique, en langage C, pour le processeur vu en TD. Ce simulateur devra être capable de reproduire toutes les étapes réelles d'exécution sur la micro-architecture du processeur.

2. Entrées/sorties du simulateur

2.1. Entrées

Le simulateur prendra en entrée un programme hexa comme sur la figure ci-dessous :

```
0200:   4A 00 01
0203:   4B 01 01
0206:   4C 02 01
0209:   4D 03 01
020C:   7D
020D:   7A
020E:   51 00
0210:   A1
0211:   71 27 02
0214:   62
0215:   72 1F 02
0218:   64
0219:   72 23 02
021C:   70 0C 02
021F:   63
0220:   70 0C 02
0223:   65
0224:   70 0C 02
```

Sur chaque ligne du programme figure l'adresse mémoire du début de l'instruction suivie de l'instruction proprement dite.
Remarque : afin de détecter « facilement » la fin du programme par votre simulateur, on rajoutera systématiquement l'adresse FFFF à la fin du programme (sans aucune instruction sur la même ligne).

2.2. Sorties

Le simulateur produira, en plus de l'exécution du programme, le fichier texte (.s) comportant le code assembleur équivalent au programme d'entrée.

3. Structure du processeur (micro-architecture)

Le simulateur doit comporter une représentation (sous formes de structures de données) dans tous les éléments composant le processeur (une variable par registre, une variable pour l'ALU, une variable par bus, un tableau pour représenter la RAM). Pour simplifier les choses, je vous propose de limiter la taille de la RAM à 1024 ou 2048 cases.
Remarque : il sera bien plus facile et plus naturel de représenter les registres R0 à R7 sous forme d'un tableau de registres et non pas sous la forme de plusieurs variables séparées….

4. Exécution des programmes

L'exécution doit se faire en respectant les étapes suivantes :

4.1. Initialisation

Copier le programme donné en entrée dans le tableau représentant la RAM (en mettant les instructions aux bonnes adresses). Puis mettre l'adresse de la première instruction du programme dans le registre PC.

4.2. Boucle à répéter pour chaque instruction

- Charger l'instruction (la mettre dans le registre RI)
- Décoder l'instruction
- Charger les éventuelles opérandes puis exécuter l'instruction

5. Fonctions à coder dans le simulateur

5.1. Décodeur

La fonction la plus importante est le décodeur d'instructions. Il s'agit d'une fonction qui prend le contenu du registre RI, puis qui le décode bit à bit en commençant par la gauche. A la fin du décodage, vous devez donc être capables de produire le code assembleur de l'instruction (exemple ADD, ..etc).

5.2. Micro-instructions

Chaque micro-instruction du processeur doit être codée sous la forme d'une fonction
Exemple :

```
function void Xin()
{
X = InternalDataBus ;
}
```

Ces fonctions représentant les micro-instructions sont, en plus du décodeur et des fonction de controle, les seuls endroits où vous avez le droit d'utiliser les opérateurs du langage C (y compris l'affectation).

5.3. Instructions assembleur

Chaque instruction assembleur doit être représentée par une fonction qui ne peut comporter que des appels aux micro-instructions. Par exemple :

```
function void ADD(1, 0) // addition de R0 et R1
{
SR(0) ;
Rout ();
Xin() ;
SR(1) ;
Rout ();
Yin() ;
addition() ; //c'est une micro instruction ! À ne pas confondre avec l'instruction assembleur.
ALUout() ;
SR(0) ;
Rin ();
}
```

6. Fonctionnement du simulateur

Le simulateur devra fonctionner en deux modes.

Premier mode : il s'agit d'une exécution simple de toutes les instructions du programme, puis la production du fichier assembleur et l'affichage des valeurs de tous les registres à la fin.

Second mode : c'est le mode debeug. Il permet de mettre des points d'arrêt (breaks), exécuter puis faire pause après chaque instruction en affichant le contenu de tous les registres.

7. Pour aller plus loin …

Vous pourrez rajouter ces fonctionnalités et/ou d'autres encore (laissez jouer votre imagination).

- Détecter et signaler des programmes comportant des erreurs (instruction non reconnue par exemple)
- Rajouter un mécanisme permettant d'afficher le nombre de cycles que prend l'exécution du programme.
- ...

8. Rendu

Le rendu se fera en 2 versions. Une première version permettant uniquement d'exécuter le programme et de générer le fichier assembleur (premier mode) et une seconde version complète et intégrant le mode debeug (second mode).

A chaque rendu, l'archive doit inclure :

- les fichiers sources
- un Readme expliquant comment compiler/exécuter votre simulateur et indiquant les éventuelles fonctionnalités que vous avez rajouté et enfin lister ce qui ne fonctionne pas.
- au moins 2 programmes hexa de 10 lignes minimum chacun pour tester votre simulateur.

Manipulation des bits en langage C

https://zestedesavoir.com/tutoriels/755/le-langage-c-1/notions-avancees/manipulation-des-bits/