USTL

INFO 202 – **2009/2010**

Architecture des ordinateurs : Fiche de TD 3

Microprocesseurs

novembre 2009

Architecture mono-bus

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

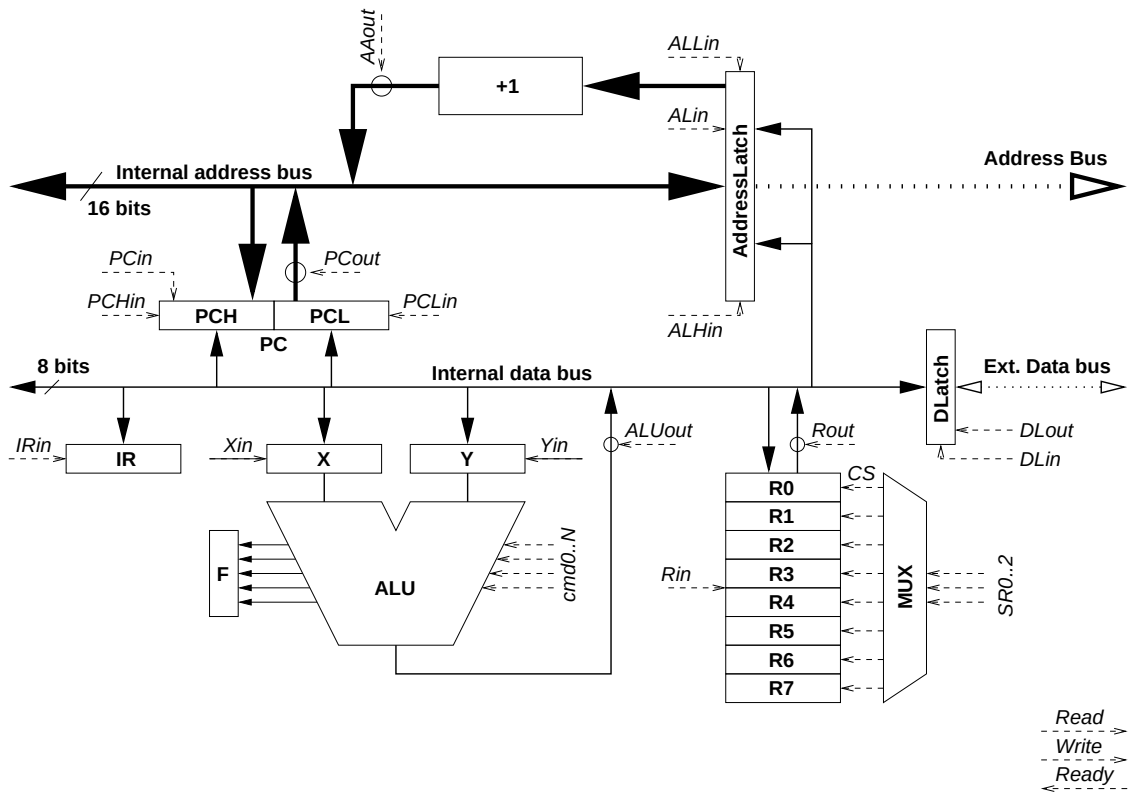

Figure 1: Architecture mono-bus

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

Table 1: Sélection dans le bloc registres

On dénomme `RX0` le registre 16 bits obtenu par la concaténation de `R1` et `R0` (`R1` octet de poids fort), `RX1` par `R2` et `R3`, `RX2` par `R4` et `R5` et `RX3` par `R6` et `R7`.

- **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 F de flags appelé registre drapeaux. 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 reliée au bus de données et est activée 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 8 bits d'instruction noté *IR* et d'une unité de contrôle. L'unité de contrôle est chargée d'engendrer les signaux nécessaires 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 8 bits de données *DLatch* (pour *Data Latch*) 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*.

1 Jeu d'instructions

On souhaite que notre microprocesseur dispose des instructions suivantes :

- `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` ;
- `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 ;
- `LD Rn, HHLL` — (*pour LoaD*) qui charge dans le registre `Rn` la valeur stockée en mémoire à l'adresse (sur 16 bits) `HHLL` ;
- `LD R0, RXn` — qui stocke la valeur du registre 0 en mémoire à l'adresse définie par le registre `RXn` ;
- `MV Rn, Rm` — (*pour MoVe*) qui charge dans le registre `Rn` la valeur contenu dans le registre `Rn` ;
- `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 ;
- `ST Rn, HHLL` — (*pour STore*) qui stocke la valeur du registre `Rn` en mémoire à l'adresse (sur 16 bits) `HHLL` ;
- `ST R0, RXn` — qui stocke la valeur du registre 0 en mémoire à l'adresse définie par le registre `RXn` ;
- `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` ;
- `SWP Rn, Rm` — (*pour MoVe*) qui échange les valeurs contenues dans les registres `Rn` et `Rn` ;
- `JMP HHLL` — (*pour JuMP*) qui effectue un branchement à l'adresse (sur 16 bits) `HHLL` ;
- `JMP RXn` — (*pour JuMP*) qui effectue un branchement à l'adresse définie par le registre `RXn` ;
- `JZ HHLL` — (*pour Jump if Zero*) qui effectue un saut à l'adresse 16 bits `HHLL` si l'opération précédente a donné un résultat nul ;
- `JC HHLL` — (*pour Jump if Carry*) qui effectue un saut à l'adresse 16 bits `HHLL` si l'opération précédente a engendré une retenue ;
- `NOP` — (*pour No Operation*) qui est une instruction qui ne fait rien.

Question 1.1 : *Combien d'instructions peuvent être codées en ne considérant que* `MV`, `SWP`, `ADD`, `SUB` *et* `AND` *?*

Solution 1.1 : Brutalement, Il y a $5 \cdot 8 \cdot 7 = 280$ possibilités.

On conclut de cette question qu'il faut restreindre les opérandes possibles ; pour `ADD`, `SUB` et `AND`, on peut imposer que la première opérande soit `R0`, `R1`, `R2` ou `R3`.

Question 1.2 : *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 :*

```
70 LL HH
```

Le code opération est l'octet `70h` *(*112 *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).*

Solution 1.2 :

1. Pour faire les opérations les plus courtes possibles en nombre d'octets, on va essayer de mettre les numéros des registres concernés (éventuellement) dans l'octet codant l'opération. Ceci nous donne :

Opération	Codage binaire	Commentaire
`ADD R`*n*`, R`*m*	??nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre
`SUB R`*n*`, R`*m*	??nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre
`AND R`*n*`, R`*m*	??nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre
`MV R`*n*`, R`*m*	??nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre
`DEC R`*n*	???? ?nnn	nnn correspond au numéro de registre
`INC R`*n*	???? ?nnn	nnn correspond au numéro de registre
`NOT R`*n*	???? ?nnn	nnn correspond au numéro de registre
`ST R`*n*`, HHLL`	???? ?nnn - hhhh hhhh - llll llll	nnn correspond au registre hhhh hhhh correspond à HH llll llll correspond à LL
`LD R`*n*`, HHLL`	???? ?nnn - hhhh hhhh - llll llll	idem
`MV R`*n*`, arg#`	???? ?nnn - aaaa aaaa	nnn correspond au registre aaaa aaaa correspond à l'argument
`JMP HHLL`	???? ???? - llll llll - hhhh hhhh	hhhh hhhh correspond à HH llll llll correspond à LL
`JZ HHLL`	???? ???? - llll llll - hhhh hhhh	hhhh hhhh correspond à HH llll llll correspond à LL
`JC HHLL`	???? ???? - llll llll - hhhh hhhh	idem
`NOP`	???? ????	

2. Premier constat : 'y a pas assez de place sur un octet ! rien qu'avec `MV R`*n*`, R`*m*, `ADD`, `SUB` et `AND` on a 256 possibilités.

   On va donc restreindre les opérandes possibles pour ces opérations. Pour le `MOV`, ce n'est pas raisonnable de restreindre, on garde donc toutes les possibilités. Par contre pour `ADD`,

`SUB` et `AND`, on peut imposer que la première opérande soit `R0`, `R1`, `R2` ou `R3`. Ceci donne (en regroupant par type) :

Opération	Codage binaire	Commentaire
`NOP`	???? ????	
`JMP HHLL`	???? ???? - llll llll - hhhh hhhh	hhhh hhhh correspond à HH llll llll correspond à LL
`JZ HHLL`	???? ???? - llll llll - hhhh hhhh	hhhh hhhh correspond à HH llll llll correspond à LL
`JC HHLL`	???? ???? - llll llll - hhhh hhhh	idem
`ST R`*n*`, HHLL`	???? ?nnn - llll llll - hhhh hhhh	nnn correspond au registre hhhh hhhh correspond à HH llll llll correspond à LL
`LD R`*n*`, HHLL`	???? ?nnn - llll llll - hhhh hhhh	idem
`MV R`*n*`, arg#`	???? ?nnn - aaaa aaaa	nnn correspond au registre aaaa aaaa correspond à l'argument
`DEC R`*n*	???? ?nnn	nnn correspond au numéro de registre
`INC R`*n*	???? ?nnn	nnn correspond au numéro de registre
`NOT R`*n*	???? ?nnn	nnn correspond au numéro de registre
`ADD R`*n*`, R`*m*	???n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SUB R`*n*`, R`*m*	???n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`AND R`*n*`, R`*m*	???n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`MV R`*n*`, R`*m*	??nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre

3. Après, pour affecter les valeurs finales, c'est un peu à l'intuition. Par exemple, il est judicieux que le `MV R`*n*`, R`*m* soit codé par 00nn nmmm car du coup, on peut faire d'une pierre deux coups : le NOP peut être codé par 0000 0000 qui est équivalent à un `MV R0, R0` (usuellement, le `NOP` est codé par 00h :-).

   Reste tous les codes opérations ne commençant pas par 00.

4. Pour les `ADD`, `SUB` et `AND`, on peut utiliser les valeurs 100, 101 et 110 par exemple.

   Reste tous les codes opérations de type 01** **** et 111* ****.

5. Pour les instructions de `ST` à `NOT`, on peut utiliser les codes 01000, 01001, 01010, 01011, 01100 et 01101.

   Reste tous les codes opérations de type 0111 **** et 111* ****.

6. Pour les instructions `JMP`, `JZ` et `JC`, on peut utiliser les codes 0111 0000, 0111 0001 et 0111 0010.

   Reste tous les codes opérations de type 0111 0011, 0111 1*** et 111* ****.

7. Fini, on a tout mis ! On va profiter du fait que l'on a encore de la place pour ajouter une instruction pour compenser les restrictions que l'on a mis sur les `ADD`, `SUB` et `AND`.

   On ajoute l'instruction `SWP R`*n*`, R`*m* qui permute deux registres et dont n est compris entre 0 et 3. Code opération : 111n nmmm.

   Reste : 0111 0011 et 0111 1***.

8. Il manque 3 types d'instructions pour que le microprocesseur soit complet : `ST` en mémoire à l'adresse donnée par un registre, `LD` en mémoire à l'adresse donnée par un registre et un `JMP` à une adresse contenue dans un registre.

   On profite des places restantes pour mettre ces instructions. On dénomme `RX0` le registre 16 bits obtenu par la concaténation de `R1` et `R0` (`R1` octet de poids fort, `RX1` par `R2` et `R3`, `RX2` par `R4` et `R5` et `RX3` par `R6` et `R7`.

   - 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, RX`*n*	0111 10nn	nn correspond au registre 16 bits concerné
`LD R0, RX`*n*	0111 11nn	idem
`ST R`*n*`, HHLL`	0100 0nnn - llll llll - hhhh hhhh	nnn correspond au registre hhhh hhhh correspond à HH llll llll correspond à LL
`LD R`*n*`, HHLL`	0100 1nnn - llll llll - hhhh hhhh	idem
`MV R`*n*`, `*arg#*	0101 0nnn - aaaa aaaa	nnn correspond au registre aaaa aaaa correspond à l'argument
`DEC R`*n*	0101 1nnn	nnn correspond au numéro de registre
`INC R`*n*	0110 0nnn	nnn correspond au numéro de registre
`NOT R`*n*	0110 1nnn	nnn correspond au numéro de registre
`ADD R`*n*`, R`*m*	100n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SUB R`*n*`, R`*m*	101n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`AND R`*n*`, R`*m*	110n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SWP R`*n*`, R`*m*	111n nmmm	0nn correspond au premier registre mmm correspond au deuxième registre
`MV R`*n*`, R`*m*	00nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre

Question 1.3 : *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 héxadécimal en supposant que le programme commence à l'adresse 0200h.

Solution 1.3 :

1. en assembleur

```
; r{\'e}cup{\'e}rer l'adresse de la premi{\`e}re zone
LD R2, 0100h
```

```
        LD R3, 0101h
        ; r{\'e}cup{\'e}rer l'adresse de la deuxi{\`e}me zone
        LD R4, 0102h
        LD R5, 0103h
        ; commencer la boucle
boucle:
        LD R0, RX1
        ST R0, RX2
        ; est-ce que l'on a copi{\'e} 00h
        MOV R1, 00h ; peut {\^e}tre remplac{\'e} par SUB R1, R1
        SUB R0, R1
        JZ fin
        INC R2
        JC incRX1 ; dans ce cas, il faut incr{\'e}menter R3
suite1:
        INC R4
        JC incRX2 ; dans ce cas, il faut incr{\'e}menter R5
        JMP boucle
incRX1:
        INC R3
        jmp suite1
incRX2:
        INC R5
        jmp boucle
fin:
```

2. en héxa

```
          0200:   4A 00 01   LD R2, 0100h
          0203:   4B 01 01   LD R3, 0101h
          0206:   4C 02 01   LD R4, 0102h
          0209:   4D 03 01   LD R5, 0103h
 boucle:  020C:   7D         LD R0, RX1
          020D:   7A         ST R0, RX2
          020E:   51 00      MOV R1, 00h
          0210:   A1         SUB R0, R1
          0211:   71 27 02   JZ fin
          0214:   62         INC R2
          0215:   72 1F 02   JC incRX1
 suite1:  0218:   64         INC R4
          0219:   72 23 02   JC incRX2
          021C:   70 0C 02   JMP boucle
 incRX1:  021F:   63         INC R3
          0220:   70 0C 02   JMP boucle
 incRX2:  0223:   65         INC R5
          0224:   70 0C 02   JMP boucle
 fin:     0227:
```

2 Micro-programmation

Question 2.1 : *Écrire La séquence de signaux (micro-instruction) réalisant la phase* 1. *On écrira sur une même ligne tous les signaux pouvant être envoyés simultanément. On utilisera un "signal" particulier :* WaitMemory *qui signale à l'unité de contrôle d'attendre le signal* Ready *de la mémoire.*

Solution 2.1 :

```
PCout, ALin, Read
AAout, PCin, WaitMemory
DLout, IRin
```

On peut également profiter du dernier cycle pour mettre la nouvelle valeur de *PC* dans *AddressLatch* (au choix *PCout*, *ALin* ou *AAout*, *ALin* qui est meilleur car moins de changements sur le bus d'adresses interne), ce qui donne :

```
PCout, ALin, Read
AAout, PCin, WaitMemory
AAout, ALin, DLout, IRin
```

Ainsi, lors de la phase 3, on a déjà *AddressLatch* qui pointe sur le premier argument éventuel.

Question 2.2 : *Écrire la phase 3 pour les instructions dont le code d'opération est : 42h, 49h, 53h, 78h, 7Fh, 81h, F5h, 70h, 73h*

Solution 2.2 :

1. 42h : `ST R2, HHLL` il y a deux octets d'opérandes.

```
phase 3.1 (r{\'e}cup{\'e}ration des arguments)
  Read, WaitMemory
  DLout, Xin, AAout, ALin, Read ; l'octet bas de l'@ est dans X
  AAout, PCin, WaitMemory
phase 3.2 (ex{\'e}cution)
  DLout, ALHin                  ; l'octet haut de l'@ est dans ALH
  RepX, ALUout, ALLin           ; la commande RepX commande {\`a} l'ALU
                                ; de reporter sur sa sortie la valeur de X
  SR2, Rout, Write, WaitMemory  ; SR2 place 010 dans SR2..0
```

2. 49h : `LD R1, HHLL` il y a deux octets d'opérandes

```
phase 3.1 (r{\'e}cup{\'e}ration des arguments)
  Read, WaitMemory
  DLout, Xin, AAout, ALin, Read ; l'octet bas de l'@ est dans X
  AAout, PCin, WaitMemory
phase 3.2 (ex{\'e}cution)
  DLout, ALHin
  RepX, ALUout, ALLin, Read, WaitMemory
  DLout, SR1, Rin
```

3. 53h : `MV R3, arg#` il y a un octet d'opérande

```
phase 3.1 (r{\'e}cup{\'e}ration des arguments)
  Read, AAout, PCin, WaitMemory
phase 3.2 (ex{\'e}cution)
  DLout, SR3, Rin
```

4. 78h : `ST R0, RX0` (pas d'opérandes)

```
  SR0, Rout, ALLin
  SR1, Rout, ALHin
  SR0, Rout, DLin, Write, WaitMemory
```

5. 7Fh : `LD R0, RX3` (pas d'opérandes)

```
SR6, Rout, ALLin
SR7, Rout, ALHin, Read, WaitMemory
SR0, DLout, Rin
```

6. 81h : `ADD R0, R1` (pas d'opérandes)

```
SR0, Rout, Xin
SR1, Rout, Yin
ADD, ALUout, SR0, Rin
```

7. F5h : `SWP R2, R5` (pas d'opérandes)

```
SR2, Rout, Xin
SR5, Rout, Yin
RepX, ALUout, SR5, Rin
RepY, ALUout, SR2, Rin
```

8. 70h : `JMP HHLL` deux opérandes

```
phase 3.1 (r{\'e}cup{\'e}ration des arguments)
  Read, WaitMemory
  DLout, Xin, AAout, ALin, Read, WaitMemory
phase 3.2 (ex{\'e}cution)
  DLout, PCHin
  RepX, ALUout, PCLin
```

   Il y a plus simple si l'on entrelace récupération des arguments et exécution. En effet, on n'a plus besoin de PC et on utilise uniquement *AddressLatch* et l'incrémenteur pour récupérer l'octet de poids fort :

```
Read, WaitMemory
DLout, PCLin, AAout, ALin, Read, WaitMemory
DLout, PCHin
```

9. 73h : `JMP RX0` (pas d'arguments)

```
SR0, Rout, PCLin
SR1, Rout, PCHin
```

Annexe : codage binaire des instructions assembleur

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
`ST R`*n*`, HHLL`	0100 0nnn - llll llll - hhhh hhhh	nnn correspond au registre hhhh hhhh correspond à HH llll llll correspond à LL
`LD R`*n*`, HHLL`	0100 1nnn - llll llll - hhhh hhhh	idem
`MV R`*n*`, `*arg#*	0101 0nnn - aaaa aaaa	nnn correspond au registre aaaa aaaa correspond à l'argument
`DEC R`*n*	0101 1nnn	nnn correspond au numéro de registre
`INC R`*n*	0110 0nnn	nnn correspond au numéro de registre
`NOT R`*n*	0110 1nnn	nnn correspond au numéro de registre
`ADD R`*n*`, R`*m*	100n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SUB R`*n*`, R`*m*	101n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`AND R`*n*`, R`*m*	110n nmmm	0nn correpond au premier registre mmm correspond au deuxième registre
`SWP R`*n*`, R`*m*	111n nmmm	0nn correspond au premier registre mmm correspond au deuxième registre
`MV R`*n*`, R`*m*	00nn nmmm	nnn correpond au premier registre mmm correspond au deuxième registre

Reste : 0111 0011 et 0111 1***.

Il manque 3 types d'instructions pour que le microprocesseur soit complet : `ST` en mémoire à l'adresse donnée par un registre, `LD` en mémoire à l'adresse donnée par un registre et un `JMP` à une adresse contenue dans un registre.

On profite des places restantes pour mettre ces instructions. On dénomme `RX0` le registre 16 obtenu par la concaténation de `R1` et `R0` (`R1` octet de poids fort), `RX1` par `R2` et `R3`, `RX2` par `R4` et `R5` et `RX3` par `R6` et `R7`.

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