TD+TP : micromachine 2015

Ce processeur est un pur 8-bit :

- ses bus d'adresse et données sont sur 8 bits ;
- le seul type de donnée supporté est l'entier 8 bits signé ;
- il possède deux registres de travail de 8 bits, notés `A` et `B`.

Au démarrage du processeur, tous les registres sont initialisés à 0, y compris le PC : le processeur démarre le programme à l'adresse 0.

Les instructions offertes par ce processeur sont

Instructions de calcul à un ou deux opérandes par exemple

```
B -> A          21 -> B          A + B -> A       B xor -42 -> A
not B -> A      LSR A -> A       A xor 12 -> A    B - A -> A ;
```

Explications :

- la destination (à droite de la flèche) peut être `A` ou `B`.
- Pour les instructions à un opérande, celui ci peut être `A`, `B`, `not A`, `not B`, ou une constante signée de 8 bits. L'instruction peut être NOT (bit à bit), ou LSR *(logical shift right)*. Remarque : le *shift left* se fait par `A+A->A`.
- Pour les instructions à deux opérandes, le premier opérande peut être `A` ou `B`, le second opérande peut être `A` ou une constante signée de 8 bits. L'opération peut être `+`, `-`, `and`, `or`, `xor`.

Instructions de lecture ou écriture mémoire parmi les 8 suivantes :

```
*A -> A         *A -> B          A -> *A          B -> *A
*cst -> A       *cst -> B        A -> *cst        B -> *cst
```

La notation `*X` désigne le contenu de la case mémoire d'adresse X (comme en C).

Comprenez bien la différence : `A` désigne le contenu du registre `A`, alors que `*A` désigne le contenu de la case mémoire dont l'adresse est contenue dans le registre `A`.

Sauts absolus inconditionnels par exemple `JA 42` qui met le PC à la valeur 42

Sauts relatifs conditionnels par exemple `JR -12` qui enlève 12 au PC

`JR offset`	`JR offset IFZ`	`JR offset IFC`	`JR offset IFN`
	exécutée si Z=1	exécutée si C=1	exécutée si N=1

Cette instruction ajoute au PC un offset qui est une constante signée sur 5 bits (entre -16 et +15). Précisément, l'offset est relatif à l'adresse de l'instruction JR elle-même. Par exemple, JR 0 est une boucle infinie, et JR 1 est un NOP (*no operation* : on passe à l'instruction suivante sans avoir rien fait).

La condition porte sur trois drapeaux (Z,C,N) qui sont mis à jour par les instructions arithmétiques.

- Z vaut 1 si la dernière instruction arithmétique a retourné un résultat nul, et zéro sinon.
- C recoit la retenue sortant de la dernière addition/soustraction, ou le bit perdu lors d'un décalage.
- N retient le bit de signe du dernier résultat d'une opération arithmétique.

Comparaison arithmétique par exemple `B-A?` ou `A-42?`

Cette instruction est en fait identique à la soustraction, mais ne stocke pas son résultat : elle se contente de positionner les drapeaux.

La spécification complète, qui fait référence, se trouve en section 7. On y trouvera aussi l'encodage de chacune de ces instructions dans un mot de 8 bits, qui est inutile au début.

1 Lisons un peu d'assembleur

Que font les programmes suivants ?
Trois petits programmes d'échauffement à gauche, un gros programme à droite.

```
prog1: 21    -> A
       42    -> B
       B+A   -> B

prog2: *21  -> A
       *42  -> B
       B+A   -> B
       B     -> *43

prog3: *100 -> A
       *101 -> B
       B-A ?
       JR +2 IFN
       B     -> A
       A     -> *102
```

```
init: -128   -> A
      A      -> *98
      A-A    -> A      ; 0->A en 1 octet
      A      -> *255   ; index
bcl:  100    -> A      ; adresse du tableau T
      *255   -> B      ; i
      B+A    -> A
      *A     -> B      ;  ainsi B contient T[i]
      *98    -> A
      B-A?
      JR +3 IFN
      B      -> *98
      *255   -> A
      1      -> B
      B+A    -> B
      B      -> *255
      *99    -> A
      B-A?
      JR +3 IF Z     ; +3 : le JA est en 2 octets
      JA bcl
fini:
```

☞ **Validez avec un enseignant.**

2 Écrivons un peu d'assembleur

Devinez quoi, on vous demande encore d'écrire une routine qui divise le contenu de la case mémoire 255 par le contenu de la case 254, et renvoie le reste dans la case mémoire 253 et le quotient dans la case 252. Elle pourra utiliser d'autres cases mémoires comme mémoire de travail (251, 250,...)

☞ **Validez avec un enseignant.**

3 Assemblons et désassemblons

La section 7 précise le codage de chaque instruction sous forme binaire.

3.1 Assemblage

Pour chacun de nos programmes d'échauffement, écrivez à droite de chaque instruction son code hexadécimal. Par exemple, on lit dans les tableaux de la section 7 que le code de `21->A` est composé des deux octets : 01001100 (0x4C) suivi de la constante 21 (0x15). N'hésitez pas à commencer par l'écrire en binaire.

Code assembleur	binaire	hexadécimal

`prog1: 21 -> A`
`42 -> B`
`B+A -> B`		

`prog2: *21 -> A`
`*42 -> B`
`B+A -> B`
`B -> *43`		

`prog3: *100 -> A`
`*101 -> B`
`B-A ?`
`JR +2 IFN`
`B -> A`
`A -> *102`		

☞ **Validez avec un enseignant.**
(Ce travail stupide et dégradant s'appelle l'assemblage *(assembly)*, et vous avez déjà envie d'écrire un programme qui le fait pour vous. Un tel programme s'appelle un assembleur.)

3.2 Désassemblage

Quel est le programme qui a pour codes hexadécimaux 4C, 11, 4D, 47, 32, E2, 42, 80 ?

Quel est le programme qui a pour codes hexadécimaux 11, 4D, 47, 32, E2, 42, 80 ?
Concluez-en qu'il ne faut pas se tromper dans ses calculs d'offset quand on fait un saut.

☞ **Validez avec un enseignant.**
(Ce travail rébarbatif et vexatoire s'appelle le désassemblage *(disassembly)*, et le programme correspondant s'appelle un désassembleur – c'est le `-d` de `objdump -d`.)

4 Architecture du processeur

La micro-architecture de ce processeur est donnée par la figure 1. Nous allons commencer par la reconstruire petit-à-petit. Vous trouverez sur Moodle un squelette de ce processeur.
Quelques remarques :

- Comme d'hab, deux fils qui ont le même nom sur le dessin sont implicitement connectés. N'hésitez pas à ajouter des fils explicites sur le dessin si cela vous aide. Mais je conseille quand même le crayon à papier...
- Les signaux dont le nom commence par "ce" sont les *clock enable* des registres correspondant.
- Quand la taille d'un vecteur de bits n'est pas explicitée, c'est qu'il s'agit d'un vecteur de 8 bits.
- L'automate est essentiellement chargé de positionner tous les *clock enable* (dont celui de la mémoire). On le construira en section 5.

Pour chaque question, on demande 2 choses : 1/ repassez la partie correspondante du dessin, et 2/indiquer la ou les ligne(s) de `processor.vhdl` correspondant.
A la fin de cette partie, vous aurez normalement repassé toute la figure 1 avec de jolies couleurs, et lu tout le vhdl...

4.1 Le PC et le cycle de von Neumann

Prenez un crayon rouge.
Ou est le registre PC ?
Comment peut-il être envoyé sur le bus d'adresse de la mémoire ?
Par quel chemin peut-on ajouter 1 au PC ?
Où est le registre d'instruction ?
Où est le registre de constante ?

4.2 Les branchements

Prenez un crayon vert.
Repassez d'abord ce qui permet de réaliser le `JA`.
Puis ce qui permet de réaliser le `JR`.
Les deux signaux ja et jr ne sont définis nulle part. Donnez en les équations logiques, à partir des bits du mot d'instruction.
`offset` est un entier signé de 5 bits extrait du registre d'instruction. Comment peut-on l'étendre à un mot de 8 bits ayant la même valeur ?

4.3 Les instructions arithmétiques

Prenez un crayon bleu.
Repassez les registres A et B.
Repassez l'ALU.
Comment sont choisis l'operande 1 et l'opérande 2 ?
Comment est choisie la destination ?

4.4 Les accès mémoire

Prenez un crayon mauve ou turquoise ou jaune vif.
Pour faire une lecture mémoire, il faut d'abord positionner l'adresse qu'on veut accéder sur le bus d'adresse. Montrez par quel chemin cela se fait.
La donnée arrive ensuite sur le bus MDI.
Montrez comment elle va pouvoir rejoindre les registres `A` ou `B`.
Pour faire une écriture, c'est pareil mais c'est le contraire.
N'oubliez pas de lever ceM.

☞ **Validez avec un enseignant.**

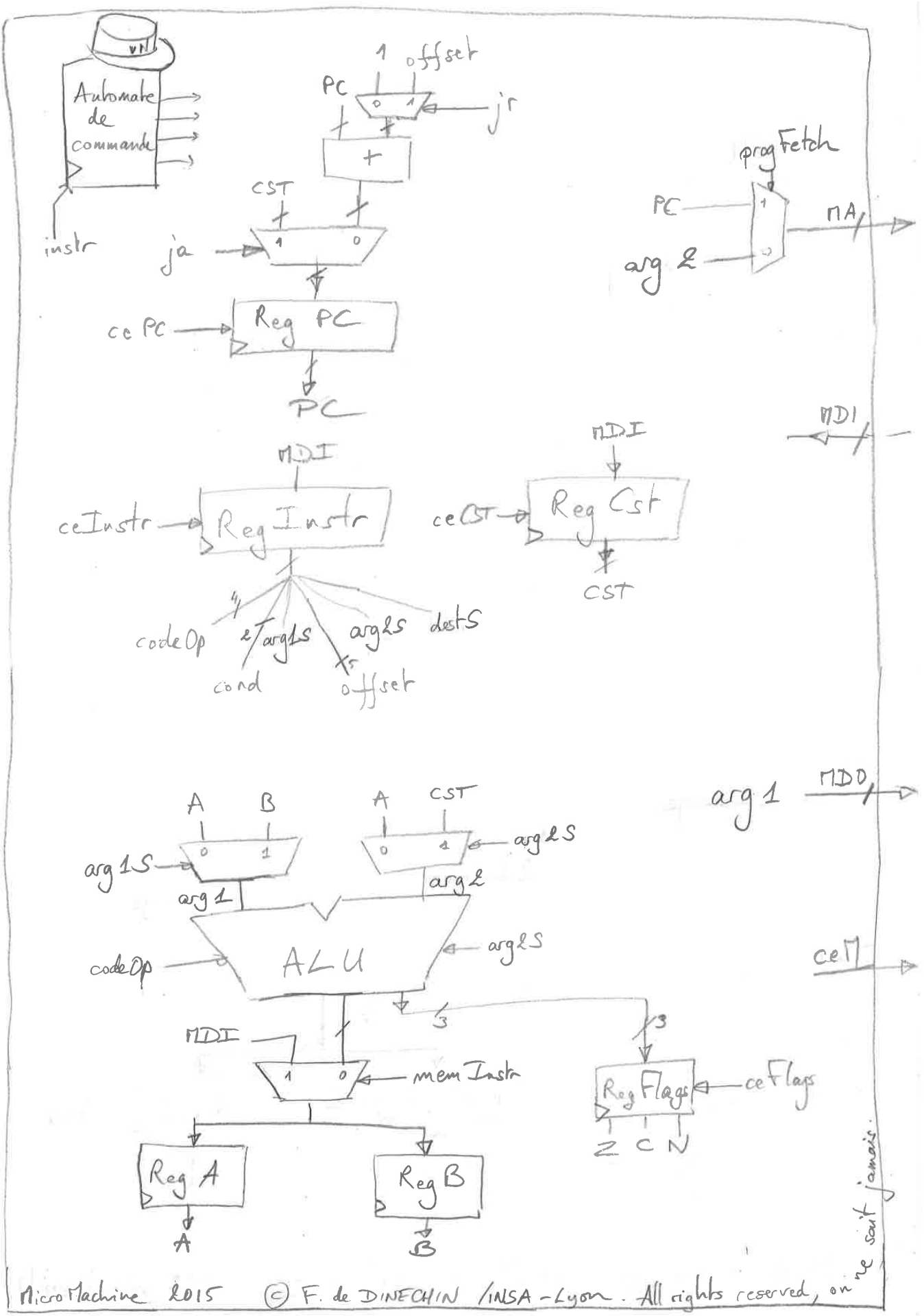

FIGURE 1 – Les différents blocs fonctionnels de notre processeur

5 Construction de l'automate

Une version de l'automate qui est facile à implémenter est décrite sur la figure 2.
Vous pouvez le reprendre en mentionnant dans les patates quels signaux il faut lever, mais vous pouvez aussi directement attaquer le code VHDL.
WriteBack est l'état qui fait le stockage du résultat dans les registres A et B.

Quelques trucs pour le debuggage de l'automate :
- Laissez gtkwave ouvert tout le temps, et contentez-vous de faire shift+ctrl+R (Reload Waveform) quand vous modifiez et resimulez votre VHDL.
- Affichez tout en haut ck, currentState, instr, A et B. Orientez-vous par rapport à ces signaux.
- Dès que vous constatez qu'un signal n'a pas la valeur qu'il devrait, commencez par remonter dans le temps au premier cycle où cela arrive, puis remontez dans le code quels signaux influent le signal fautif. Observez-les dans gtkwave, jusqu'à trouver la source du problème.

Quelques pièges à éviter en VHDL :
- Dans la syntaxe
  ```
  toto <= '1' when (currentState="tata" and condition='1')  else '0';
  ```
  n'oubliez pas le `else` !

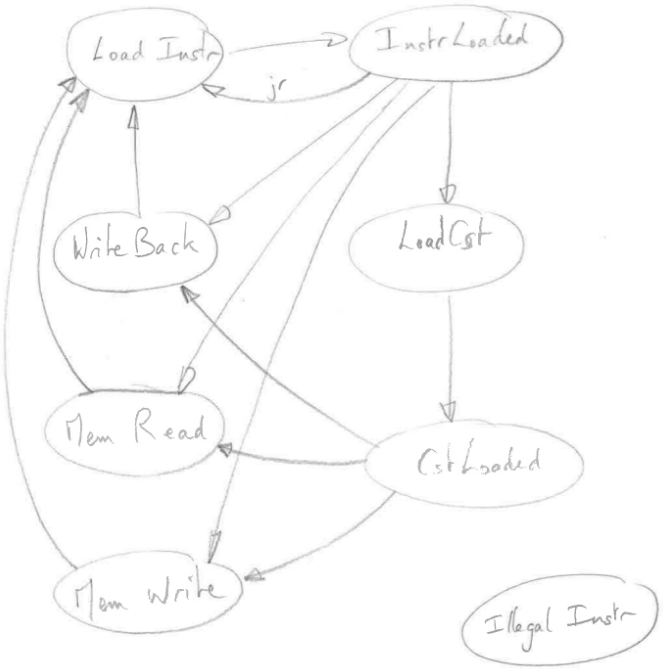

FIGURE 2 – Le cycle de von Neumann ressemble plutôt à un plat de nouilles

5.1 Les instructions arithmétiques en 1 octet

Surlignez en couleur les flèches du cycle de von Neumann correspondant, qui passe par LoadInstr, InstrLoaded, et WriteBack.
Dans le VHDL fourni sur Moodle, ces instructions fonctionnent déjà. Lisez-le attentivement, et posez des questions sur ce que vous ne comprenez pas.
Testez-les sur le programme suivant (que fait-il ?), que vous assemblerez et placerez au début de la mémoire :

```
not a -> a
b-a -> b
b-a -> b
b-a -> b
b-a -> b
b-a -> b
```

5.2 Le saut relatif (inconditionnel pour commencer)

Comme il tient lui aussi en 1 octet, on peut le greffer à notre automate.
On remarque que cePC est mis à 1 dans l'état InstrLoaded. Donc dans l'état InstrLoaded, si l'instruction est un JR, elle peut positionner les multiplexeurs devant RegPC comme il faut : le saut relatif vient tout seul au moment ou regPC est mis à jour.
Ne vous préoccupez pas des sauts conditionnels à ce stade. Testez en remplaçant par une boucle la série de `b-a -> b` dans le programme ci-dessus.

5.3 Les instructions arithmétiques en 2 octets

Il s'agit d'ajouter les états LoadCst et CstLoaded. Surlignez les flèches correspondantes d'une autre couleur.
Précisez les signaux que doit positionner l'état LoadCst.
N'oubliez pas que PC doit être incrémenté.
Testez sur prog1.

5.4 Le saut absolu

Il y a une flèche qui manque au dessin de l'automate. Ajoutez-la.

5.5 Les sauts relatifs conditionnels

Il faut créer un signal condTrue qui compare l'état des drapeaux avec les bits de condition. Le signal de commande de multiplexeur jr sera le et logique de ce signal et d'un signal vrai quand l'instruction est un jr.

5.6 Les accès mémoire

Dans l'état WriteMem on se contente de positionner les bus adresse et donnée comme il faut.
Dans l'état ReadMem, on positionne l'adresse, puis on lève ceDest comme dans l'état WriteBack.

5.7 Optimisations possibles

De toute évidence on peut compresser WriteBack et LoadInstr en un seul état. Essayez.

6 Extensions possibles de ce processeur

- Trouver une utilité aux bits inutilisés des opérations arithmétiques :
  - le shift logique peut devenir un shift arithmétique, c'est-à-dire qui recopie le bit de signe, en fonction de arg2S
- Ajouter ADC (add with carry) pour permettre de construire facilement l'arithmétique 16 et 32 bits.
- Une pile (avec push et pop).
- Un mécanisme de CALL/RET avec un link register comme sur les ARM (peut utiliser les bits inutilisés de JA)

7 Annexe : Encodage du jeu d'instruction

Les instructions sont toutes encodées en un octet comme indiqué par la table 1. Pour celles qui impliquent une `constante` (de 8 bits), cette constante occupe la case mémoire suivant celle de l'instruction.
La table 2 décrit la signification des différentes notations utilisées.

TABLE 1 – Encodage du mot d'instruction

bit	7	6	5	4	3	2	1	0
instruction autres que JR	0	codeop, voir table 3				arg2S	arg1S	destS
saut relatif conditionnel	1	cond, voir table 4		offset signé sur 5 bits				

TABLE 2 – Signification des différents raccourcis utilisés

Notation	encodé par	valeurs possibles
`dest`	destS=instr[0]	`A` si destS=0, `B` si destS=1
`arg1`	arg1S=instr[1]	`A` si arg1S=0, `B` si arg1S=1
`arg2`	arg2S=instr[2]	`A` si arg2S=0, constante 8-bit si arg2S=1
`offset`	instr[5 :0]	offset signé sur 5 bits

TABLE 3 – Encodage des différentes opérations possibles

codeop	mnémonique	MàJ drapeaux	remarques
0000	`arg1 + arg2 -> dest`	oui	addition ; shift left par `A+A->A`
0001	`arg1 - arg2 -> dest`	oui	soustraction ; `0 -> A` par `A-A->A`
0010	`arg1 and arg2 -> dest`	oui	
0011	`arg1 or arg2 -> dest`	oui	
0100	`arg1 xor arg2 -> dest`	oui	
0101	`LSR arg1 -> dest`	oui	logical shift right ; bit sorti dans C ; arg2 inutilisé
0110	`arg1 - arg2 ?`	oui	comparaison arithmétique ; destS inutilisé
1000	`(not) arg1 -> dest`	non	`not` si arg2S=1, sinon simple copie
1001	`arg2 -> dest`	non	arg1 inutilisé
1101	`*arg2 -> dest`	non	lecture mémoire ; arg1S inutilisé
1110	`arg1 -> *arg2`	non	écriture mémoire ; destS inutilisé
1111	`JA cst`	non	saut absolu ; destS, arg1S et arg2S inutilisés

Remarque : les codeop 0111, 1010, 1011, et 1100 sont inutilisés (réservés pour une extension future...).

TABLE 4 – Encodage des conditions du saut relatif conditionnel

cond	00	01	10	11
mnémonique	(toujours)	`IFZ` si zéro	`IFC` si carry	`IFN` si négatif