ARC TD4

Un CPU von Neuman complet
(4h, dont 2h sur papier au debut), 4 juin 2026

Construit à partir du poly de TD-TP des cours IF-AC et IF-AO

J'ai mis a jours la micro machine IF avec celle proposée sur Digital, à voir si ce n'est pas trop compliqué. Timing : finir les partie 1/2/3 à la pause, commence la partie digital après la pause.

Dans ce TD, on va étudier un CPU simple 8 bits. Nous proposerons d'abord un assembleur possible pour cette machine simple et étudierons les notions d'encodage d'instruction, d'assemblage, de de-assemblage, format binaire etc.
Puis il vous sera fournit le code au format Digital d'une architecture de CPU qui implémente ce jeu d'instruction. Il s'agit donc d'un petit processeur 8-bits décrit en Digital. il n'implémente pas toutes les instructions mais juste le saut, conditionnel et inconditionnel. Le travail consistera a modifier l'automate du CPU pour ajouter progressivement des instructions.
La micromachine, i.e. le processeur 8-bit, possède les spécifications suivantes :

- 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. C'est vrai pour `A` et `B`, et aussi pour le PC : le processeur démarre donc avec le programme à l'adresse 0.
Les instructions assembleur de ce processeur sont regroupées en 5 catégories : Instructions de calcul, Instructions d'accès à la mémoire, sauts absolus, sauts relatifs et comparaison arithmétique.

Instructions de calcul à un ou deux opérandes par exemple

```
B -> A        21 -> B          B + A -> 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
`JA lbl` saute à l'adresse du *label* `lbl` (un label peut être mis au début de chaque ligne, avant l'instruction assembleur de cette ligne).

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

`JR offset` | `JR offset IFZ` exécutée si Z=1 | `JR offset IFC` exécutée si C=1 | `JR offset IFN` 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). Ces drapeaux sont mis à jour par les instructions arithmétiques et logiques.

— Z vaut 1 si l'instruction 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 résultat d'une opération arithmétique ou logique.

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.

1 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. Les instructions micromachine peuvent donc avoir une longueur de un octet ou deux octets 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[4 :0]	offset signé sur 5 bits

QUESTION 1 ▶ Verifiez que vous comprennez les tables 3 et 2, quelle est l'encodage de l'instruction `B+A -> B`

Bit 7 à 0, codeop = 0000, arg2S=0 (A), arg1S=1 (B), destS=1 (B), d'ou : 000000011_b=0x03

2 Lisons un peu d'assembleur

Que font les programmes suivants ?
Trois petits programmes d'échauffement à gauche, un gros programme à droite (`init`, `bcl` et `fini` sont des labels permettant de faire un saut absolu à ces instructions).

TABLE 3 – Encodage des différentes opérations possibles

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

```
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 +2 IFN
      B      -> *98
      *255   -> A
      1      -> B
      B+A    -> B
      B      -> *255
      *99    -> A
      B-A?
      JR +3 IFZ      ; +3 : le JA est en 2 octets
      JA bcl
fini:
```

☞ **Validez avec un enseignant.**

A gauche : le premier programme calcule 21+42. Le second calcule la somme des cases mémoires d'adresse 21 et 42, et stocke le résultat dans la case mémoire d'adresse 43. Le troisième calcule le MAXIMUM des valeurs contenues aux adresses 100 et 101, et stocke le résultat dans la case 102 (if B-A <0, on saute l'affectation B->A).
A droite : on calcule dans *98 le max des éléments d'un tableau commencant à l'adresse 100 et dont la taille est rangée dans *99. L'index i est rangé à la case 255.

```
max rangé à &98
size rangé à &99
```

```
T rangé à &100
i rangé à & 255

max=-128
while (i<size) {
    if (T[i]>max) max=T[i];
    i=i+1;
}
```

Désolé, on a oublié de parler de *labels*. Une mise au point générale s'imposer, et en plus c'est taquiin à espliquer.
Demander pour prog3 de l'expliciter comme *102=f(*100, *101) en explicitant la fonction f.
Bien faire identifier le paquet d'instructions qui fait index++.

TANG : j'ai supprimé la question *écrivons un peu d'assembleur*, je l'ai rajouté en fin de sujet.

3 Assemblons et désassemblons

La section 1 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 1 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.**

Code assembleur	binaire	hexadécimal
`21 -> A`	`0 100 1 1 x 0 0001 0101`	4C 15
`42 -> B`	`0 100 1 1 x 1 0010 1010`	4D 2A
`B+A -> B`	`0 000 0 0 1 1`	03
`*21 -> A`	`0 110 1 1 x 0 0001 0101`	6C 15
`*42 -> B`	`0 110 1 1 x 1 0010 1010`	6D 2A
`B+A -> B`	`0 000 0 0 1 1`	03
`B -> *43`	`0 111 0 1 1 x 0010 1011`	76 2B
`*100 -> A`	`0 110 1 1 x 0 0110 0100`	6C 64
`*101 -> B`	`0 110 1 1 x 1 0010 1011`	6D 65
`B-A ?`	`0 011 0 0 1 x`	32
`JR +2 IFN`	`1 11 0 0010`	E2
`B -> A`	`0 100 0 0 1 0`	42
`A -> *102`	`0 111 0 1 0 x 0110 0110`	74 66

Remarque : attention à l'instruction `B -> A` qu'on ne peut pas implémenter avec le MOV classique 1001, car celui-ci ne permet pas de sélectionner B comme source. Pour cette instruction, il faut utiliser le not-conditionnel 1000.

(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, 6D, 4D, 80, 32, E2, 42, 80 ?

```
"01001100", --0 cst -> a
"01101101", --1   (constante 0x6D)
"01001101", --2 cst -> b
"10000000", --3   (constante: -128)
"00110010", --4 B-A?
"11100010", --5 jr 2 IFN
"01000010", --6 B -> A
"10000000", --6 jr 0 (boucle infinie)
```

Ce programme fait : A=max(A,B) ;

Quel est le programme qui a pour codes hexadécimaux 6D, 4D, 80, 32, E2, 42, 80 ?
Concluez-en qu'il ne faut pas se tromper dans ses calculs d'offset quand on fait un saut.

*0x4D -> B ; puis une boucle infinie.... Ce programme est correct même si on a pris un autre programme que l'on a décalé d'un octet.

```
"01101101" -> "0""1101" *arg2 -> dest avec arg1=1 (B), dest =1 donc cst
"0x4D" -> cst 0x4D
"1000000" -> "1" -> JR "00" allways "00000" -> JR 0 boucle infini
```

☞ **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 Datapath et Automate de notre micro-machine

Interessons nous maintenant à la machine qui va exécuter cet assembleur. Nous présentons Ici le *datapath* de la micro-machine (Fig. 1). Pour l'instant on ne peut pas comprendre complètement ce *datapath* mais on peut en comprendre certaines parties.

QUESTION 2 ▶ Repérez sur cette figure : les registre A et B utilisés dans l'assembleur. Le compteur de programme ainsi que le registre d'instruction.

> Pas dur, on voit les registres qui s'appellent Reg A et Reg B, ils sont utilisés en entrée de l'ALU.

QUESTION 3 ▶ les signaux bleus sont ceux provenant de l'automate de contrôle (i.e. le décodeur d'instruction). l'acronyme "`ce`" signifie *clock enable*, ce sont des signaux qui vont activer ou désactiver le chargement de certains registres. Repérez par exemple, le signal qui vont autoriser le chargement d'une nouvelle instruction dans le registre d'instruction, ainsi que celui qui va autoriser la modification du compteur de programme

> Il s'agit de `cpPC` et `ceIR`

QUESTION 4 ▶ Les signaux sur la droite vont faire l'interface avec la mémoire : MA (Memory Data Address), MDO (Memory Data Out), MDI (Memory Data IN), ceI (Clock Enable Memory). Dans quels registres peuvent arriver les données venant de la mémoire. Et que contiennent ces données qui viennent de la mémoire ?

> Les données qui rentrent sont soit des intructions, soit des données (avec l'instruction *Cst → A). Les instructions peuvent être de 1 octet ou 2 octets, lorsqu'elles sont de deux octets, le deuxième octet est une constante. Lorsque la données est une constante, elle sera chargée dans le registre `Reg Cst`.

Etudions maintenant un peu plus en détail l'architecture de l'unité de calcul de la micro-machine de la Fig. 1.
L'automate de contrôle (correspondant à la boite *Control Unit* sur la Fig. 1) est représenté en Fig. 2. On voit que cet automate comprend 6 états nommés `InstrFetch`, `InstrDecode` (correspondant aux notion de *Fetch* et *Decode* vues en cours), `regWrite` (correspondant au *Write Back*), `incrPc` (qui permet d'incrémenter PC pour lire une constante), `cstFetch` (qui charge une constante) et `DoJA` (qui executera l'instruction de saut absolu JA).

QUESTION 5 ▶ On comprend, en regardant cet automate, qu'il y a deux cycles possibles : l'un des cycle comporte trois états (donc trois cycles d'horloge), l'autre en comporte 5 (donc 5 cycles d'horloge). Comprenez vous dans quels cas on va utiliser trois états ou 5 états dans l'automate ?

> soit on décode une instruction de 1 octet :
> `InstrFetch→InstrDecode→regWrite→InstrFetch`
> soit on décode une instruction de 2 octets :
> `InstrFetch→InstrDecode→incrPC→cstFetch→(RegWrite` ou `DoJA)→InstrFetch`,

5 Lancement de la micromachine avec `digital`

Téléchargez sur Moodle le processeur VonNeuman (fichier nommé `MM-Minimal.tar`). Extrayez l'archive, cela crée un répertoire `MM-Minimal`. Descendez dans ce repertoire est ouvrez avec `digital` le fichier `mm-minimale.dig`
Le circuit comporte 3 parties :

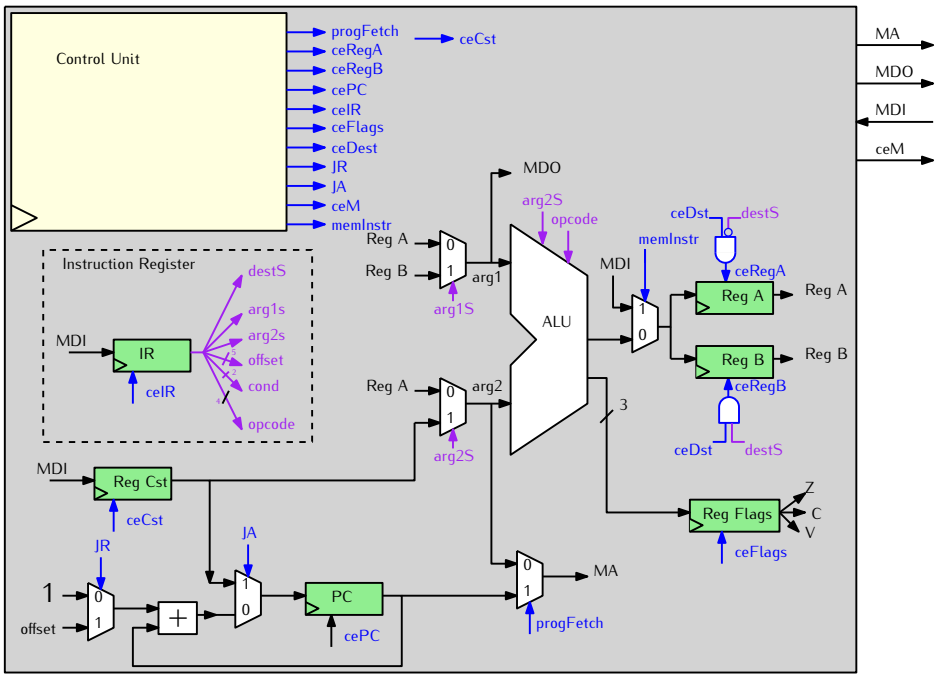

FIGURE 1 – Le datapath de la micro-machine (on notera les similarité avec les premier le processeur (Intel 8080 en figure 7, page 13).

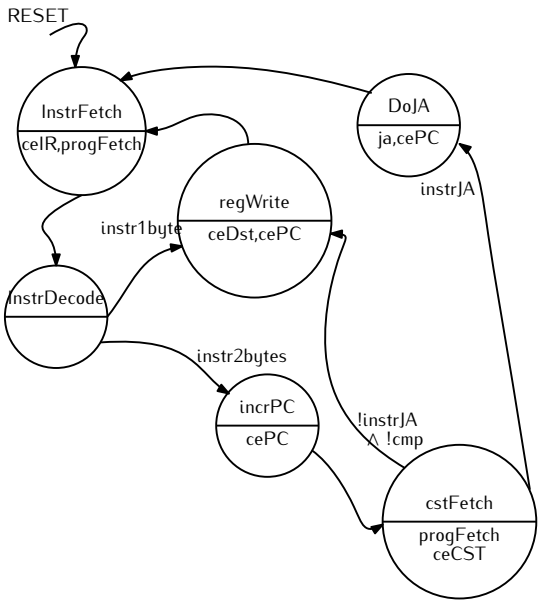

FIGURE 2 – Automate de la version initiale de notre micro-machine Digital.

— **MM Control Unit** : L'unité de contrôle ou décodeur d'instruction, qui implémente l'automate représenté en Fig 2 (et qui est aussi représenté par le bloc `Control Unit` sur la Fig. 1).

— **MM DataPath** : le chemin de donné, représenté en Fig. 1.

— **Console** : un module qui nous permet de surveiller l'évolution de la machine, grâce aux afficheurs sept segments branchés sur plusieure registres : `MA` (addresse mémoire), `MDO` (Donnée envoyée à la mémoire), `MDI` (donnée reçue de la mémoire), `PC`, `RegIR` et `RegCst` déjà vu en fin de dernier TD. On y voit aussi l'état courant de l'automate ainsi qu'un compteur du nombre de cycles effectués.

— **RAM** : pour faire les accès mémoire.

QUESTION 6 ► Faites le lien entre le fichier `mm-minimale.dig` et les figures 1 et 2. En particulier, vous prendrez soin de repérer l'ALU, l'automate de contrôle, les registres `A`, `B`, `PC` et `Reg Cst`.

QUESTION 7 ► quels sont les registres utilisés pour stocker l'état de l'automate, combien de registre pour combien d'état ?

chaque état de l'automate est encodé par **exactement 1 registre à 1 bit qui contient la valeur '1' lorsque l'automate est dans l'état correspondant**. C'est ce qu'on appelle l'encodage "one-hot coding".

6 Execution d'un programme simple

La version de la micro-machine présent ici est capable d'interpréter les instructions suivantes :

— Instructions logiques et arithmétiques, y compris en 2 octets ;

— Sauts absolus.

On rappelle le codage de certaines de ces instructions :

Code assembleur	binaire	hexadécimal
`21 -> A`	`0 100 1 1 x 0 0001 0101`	4C 15
`42 -> B`	`0 100 1 1 x 1 0010 1010`	4D 2A
`B+A -> B`	`0 000 0 0 1 1`	03
`JA 0x05`	`0 111 1 0 0 0 0000 0101`	78 05

QUESTION 8 ► Visualisez d'abord le programme stocké dans la mémoire, pouvez vous le décoder ?

c'est exactement le programme montré juste avant : il calcule 21+42=63 puis fait une boucle infinie. 4C 15 4D 2A 03 78 05 AA
soit : 21 → A, 42 → B, B+A → A, JA 0x5
Le AA ne sert à rien puisqu'on arrive jamais là.

QUESTION 9 ► Pour l'exécuter sur la micro-machine, lancez la simulation du circuit sous Digital. Dans la partie "Console", pour initialiser les registres, cliquez sur "Init", puis deux fois sur "CLK" puis à nouveau sur "Init" (Init doit être dans l'état 0, rouge, pour que le processeur avance.
Puis exécuter votre programme en faisant avancer l'horloge principale (à la souris en cliquant sur la "Clock Input" de la partie "Console", ou au clavier en tapant 'c'). Suivez le comportement de votre programme dans le processeur.

— Surveillez le registre `RegPC` pour voir l'avancement du compteur de programme

— Surveillez le registre `RegIR` pour voir le instructions décodées.

— Surveillez de registre `RegCst` pour voir les constantes utilisées.

— Suivez les transitions d'état sur le dessin de la figure 2.

— Que fait le programme au bout d'un moment ?

Ca se fait bien, au bout d'un moment on a une boucle infinie (JA 0X5 à l'adresse 5).

QUESTION 10 ▶ On rappelle ci-dessous le codage des instructions, repérez ou est faite la décomposition de l'instruction en ces différents champs. Ou est utilisé le champ `codeop` par exemple ?

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

La décomposition est faite en haut à gauche du datapath, dans le splitter.
L'utilisation de `codeop` est dans le grand démultiplexeur à gauche dans la control Unit.

QUESTION 11 ▶ Repérez la partie du datapath qui incrémente PC. Dans quels états incrémentent-on le PC ? Quand est ce qu'on incrémente le PC de autre chose que 1 ?

Dans les état `RegWrite`, `DoJA` et `IncrPC` : le signal qui contrôle le chargement de ce PC est `cePC`, il est mis à jours dans l'automate par un ou entre ces trois signaux. On l'incrémente d'autre chose que 1 lorsque le signal `ja` est à 1 c'est à dire lorsque l'instruction est un saut absolu (i.e. lorsque l'on passe dans l'état DoJA).

7 Construction par étapes de l'automate

L'automate actuel est capable de reconaître le `codeop` de toutes les instruction, mais les actions à effectuer ne sont pas toutes implémentées.
Nous allons progressivement augmenter l'automate pour ajouter l'implémentation des instructions manquantes. Pour cela, sauvegardez la micro-machine sous un autre nom (par exemple `MM-Minimal-login.dig`) et modifiez incrémentalement la machine en suivant les instructions qui vous sont données dans le sujet.
Nous allons d'abord ajouter l'opération de comparaison. Pour cela nous avons besoin d'activer les drapeaux (rappel, il y a trois drapeaux de 1 bit : `N` : Negative, `C` : Carry et `Z` : Zero).

7.1 Version 1 : Mise à jour des drapeaux et instruction de comparaison arithmétique

Pour l'instant, à chaque opération de l'ALU, votre processeur écrit le résultat dans un registre. Il produit les drapeaux Z, C et N mais ne les stocke pas. (Ces drapeaux sont nécessaires à l'exécution des branchements conditionnels que nous ajouterons dans la suite).
Dans cette partie, vous allez ajouter la mise à jour de ces drapeaux, ainsi que le nécessaire à l'exécution de l'instruction de comparaison arithmétique.

QUESTION 12 ▶ Allez voir l'unité arithmétique et logique, essayer de comprendre rapidement le design. Identifiez par exemple dans quelles opération le drapeau `C` (Carry) peut être mis à 1. Combien de bit le port Flags comporte-t'il

Dans le cas addition, soustraction et LSR. si les opérations correspondantes génère une retenue.
Le fil Flags comporte 3 bits, il faudra donc un splitter 3-5.

Sauvegarde des flags

Vous procéderez en deux étapes :

— Ajoutez, dans le datapath, un registre , nommé `Flags`, permettant de stocker les flags Z, C et N en sortie de l'ALU. (on utilisera un registre de 8 bits avec un splitter). Vous utiliserez le registre Reg8 présent dans le menu `Components` → `custom`. Pour que le registre apparaisse comme les autres, il faut lui mettre l'apparence "layout".

— ajoutez dans l'état `regWrite` la mise à jour du signal d'écriture (i.e. *enable*) du le registre `Flags` (ceFlags).

QUESTION 13 ▶ Re-exécuter votre programme et vérifiez que le registre Flags est bien mis à jour. Modifiez le programme pour qu'il manipule des valeurs permettant de vérifiez que la valeur des flags est bien mise à jour correctement. Par exemple, Vous pouvez utiliser le programme suivant dans la mémoire : `4C 15 4D EB 03` qui va calculer 15 - 15 et va donc positionner Flags à 3 (Carry et Zero).

Comparaison arithmétique

Pour rappel, l'opcode, le mnémonique et le comportement de cette instruction sont rappelés dans la table 5. En particulier, vous constaterez (ou vous rappellerez) que **le seul effet de cette instruction est de mettre à jour les drapeaux Z, C et N**. Du point de vue de l'automate, l'ajout de cette instruction nécessite l'ajout d'un seul état nommé "noRegWrite". Les modifications à apporter à l'automate de contrôle sont visibles sur la version de la figure 3.

codeop	mnémonique	remarques
0110	`arg1 - arg2?`	Produit le résultat de `arg1-arg2` sur la sortie mais ce ne sera pas enregistré dans un registre

TABLE 5 – Rappel : descriptif de l'instruction de comparaison arithmétique

le codage de B - A ? est 0x32 moi j'ai mis :
4C 16 4D 15 03 32 78 06
Donc il compare 0x15 (B) et 0x16 (A) $rightarrow$ B-A est négatif
On voit le flag `instrCMP` qui se met à 1 à gauche dans l'état decode et on passe dans l'état noRegWrite
On peut mettre un nouveau programme en mémoire avec memory -> edit -> load

Ajoutez maintenant cet état à la partie "Control Unit" de votre micro-machine, en augmentant les parties "Transition Function" et "Output Function".
Il faut penser à :

— à bien regarder la figure 3 pour comprendre ce qui va déclencher le passage dans l'état `noRegWrite`
— à ce que la machine à état passe dans l'état `noRegWrite` quand on a une comparaison
— que cet état mette à 1 le signal `ceFlag`
— que l'automate retourne dans l'état `InstrFetch` après l'état `noRegWrite`
— et à un autre truc encore que l'on vous laisse deviner.

Attention, là ca devient un peu plus compliqué. Il faut rajouter un registre `noRegWrite`, ce registre est triggé, comme sur la figure 3 par `1ByteInstr` et `instrCMP`, mais il faut **aussi** `InstrDecode`, sinon il est triggé trop souvent.
Ensuite il faut ajouter l'entrée `noRegWrite` pour activer `cePC` puisqu'il faut passer à l'instruction suivante (c'est ca qu'on leur dit pas) et enfin il faut ajouter l'entrée `noRegWrite` pour activer `InstrFetch` pour repasser dans l'état `instrFettch`

Test Dans votre programme ajoutez une instruction de comparaison (vous trouverez le code hexadécimal de l'instruction dans le TD précédent), et testez le programme à nouveau. vérifiez bien que le contrôle de flôt revient à un mode normal après l'exécution de votre instruction et que le résultat de la comparaison est visible sur le registre des drapeaux.

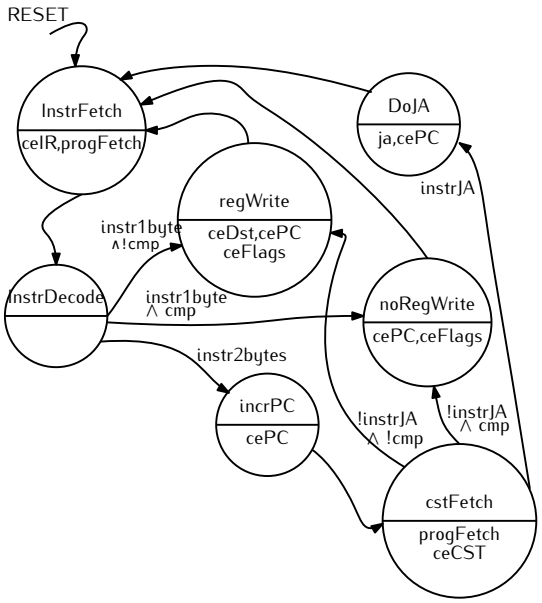

FIGURE 3 – Le cycle de Von Neumann avec **mise à jour des drapeaux** et **instruction de comparaison**

8 Pour ceux qui ont finis : autres instructions

8.1 Version 2 : Le saut relatif inconditionnel

On vous demande d'ajouter la prise en charge du saut relatif. Dans un premier temps, on vous demande de ne pas implémenter le test qui permet de savoir si oui ou non le branchement doit être effectué. On parle dans ce cas de saut relatif inconditionnel. L'automate correspondant est donné à la figure 4.

8.2 Version 3 : Les sauts relatifs conditionnels

Afin de pouvoir exécuter des sauts conditionnels, on doit regarder la valeur des drapeaux (flags) qui permettent de décider si il faut faire le saut ou non. La figure 5 décrit l'automate attendu.

Test Implémentez maintenant dans un fichier `prog2.hex` le programme de la figure 6. Chargez leur en mémoire. Testez votre micro-machine.

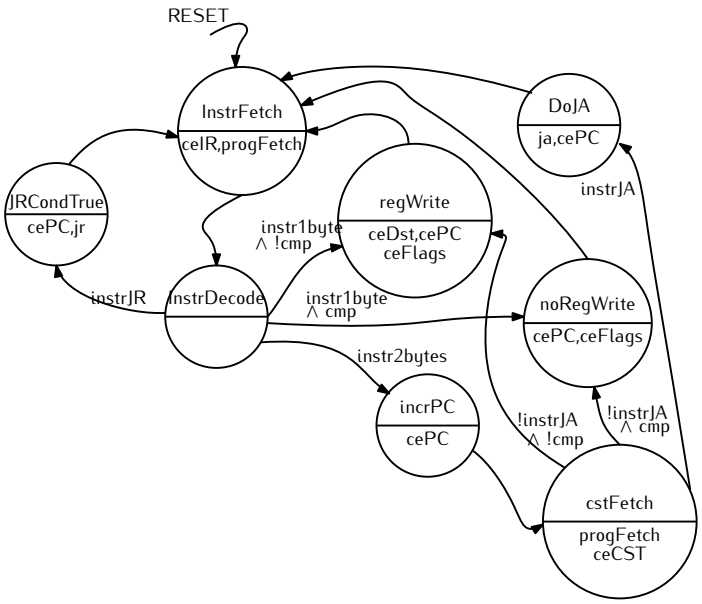

FIGURE 4 – Le cycle de Von Neumann avec **sauts relatifs inconditionnels**

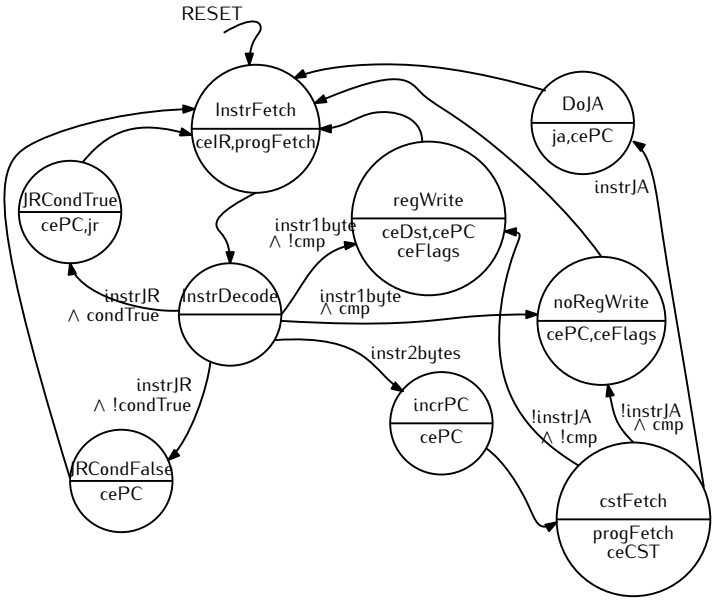

FIGURE 5 – Le cycle de Von Neumann avec **sauts relatifs conditionnels**

```
   42 -> b
   7 -> a
f: b-a?
   JR +3 IFN
   b-a -> b
   JR -3
```

FIGURE 6 – Un programme utilisant le test et les branchements relatifs conditionnels (et inconditionnels).

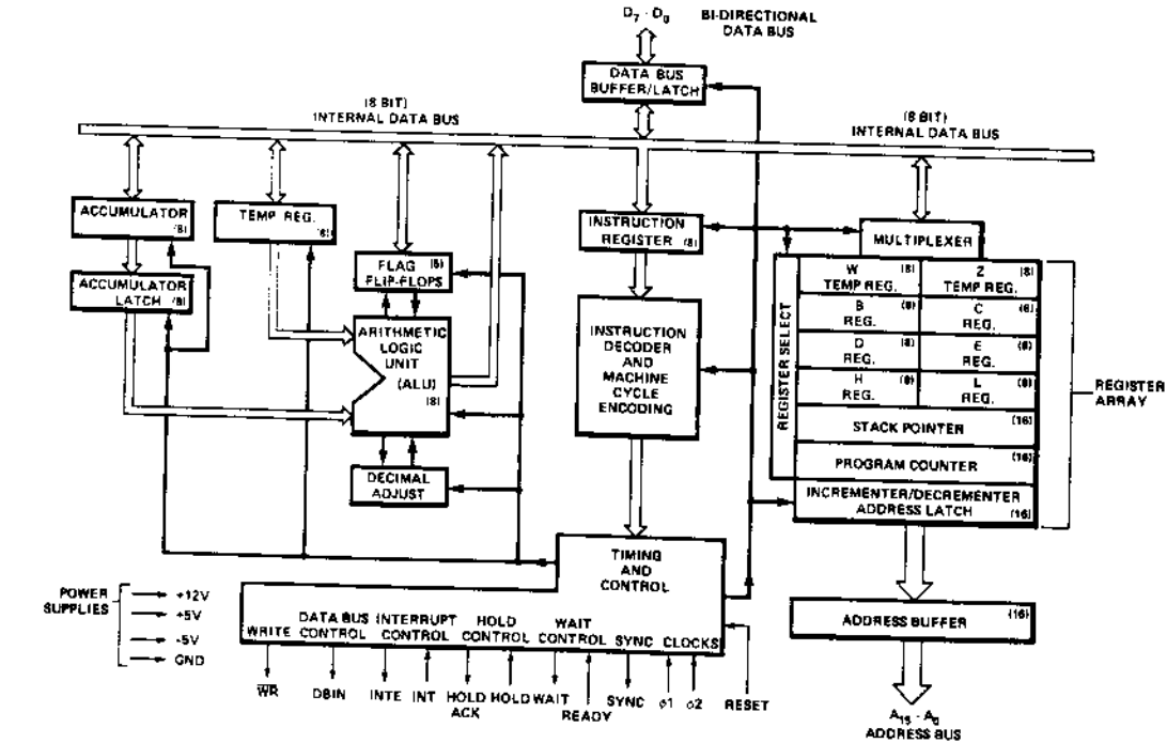

Figure 2-2. 8080 CPU Functional Block Diagram

FIGURE 7 – Le schéma original du l'Intel 8080 extrait de la documentation de 1975 (à rapprocher du schéma de la figure 1). Origine : https://bitsavers.trailing-edge.com/components/intel/MCS80/