Architecture des Ordinateurs
Partie 1 “Micro-Machine”

Partie I: Micromachine - Programmation Assembleur

1 Jeu d’instruction

1.1 Description

Dans ce TD, nous allons lire et écrire des programmes assembleur pour un processeur 8-bit, avec 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 Program Counter (PC) : le processeur démarre donc avec 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`	`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

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). 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.

1.2 Encodage

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[4 :0]	offset signé sur 5 bits

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

2 Lisons un peu d'assembleur

Dans tout le module, nous allons écrire des programmes en assembleur (que ce soit pour la micro-machine ou pour le msp430). Il est important de se familiariser avec les concepts des jeux d'instruction, c'est le but de cet exercice. Ensuite, il sera également important de se familiariser avec la traduction de ce langage assembleur en sa version binaire, appelé "langage machine". C'est notamment nécessaire pour être capable de suivre facilement le programme pendant son exécution par le processeur.

2.1 Étiquettes - labels

`prog1`, `prog2`, `init`, `bcl`, ... sont des *étiquettes :* (en anglais et en franglais aussi "labels"). Leur déclaration, par exemple `bcl:` définit une adresse. On peut les utiliser à la place de la destination d'un saut, c'est plus lisible, et on saura les remplacer (au moment de la traduction de l'assembleur en binaire) par la distance de saut (pour les sauts relatifs) ou l'adresse correspondante (pour les sauts absolus).

2.2 Échauffements

Lisez les programmes ci-dessous. Proposez un comportement pour chacun d'eux.

```
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
```

2.3 Un programme plus intéressant (à finir impérativement avant le prochain TP)

Attention : les numéros à gauche indiquent des numéros de ligne de programme source.

```
1  ; (Le point-virgule introduit un commentaire.)
2  ; En entree de ce programme, on a un tableau
3  ; d'octets positifs ou nuls, stockes aux adresses
4  ; 100, 101, 102, ...
5
6  init: A-A   -> A     ; 0->A en 1 octet
7        A     -> *98
8        A     -> *255  ; index i
9  bcl:  100   -> A     ; adresse du tableau T
10       *255  -> B     ; i
11       B+A   -> A
12       *A    -> B     ; ainsi B contient T[i]
13       *98   -> A
14       B-A?
15       JR +3 IFN
16       B     -> *98   ; instruction en 2 octets
17       *255  -> B
18       B+1   -> B
19       B     -> *255
20       *99   -> A     ; qu'y a-t-il a l'adresse 99 ?
21       B-A?
22       JR +3 IF Z     ; +3 car le JA est en 2 octets
23       JA bcl
24 fini:
```

Interprétez pas-à-pas ce programme.
Pour cela, répondez succintement aux questions suivantes :

— Que fait l'instruction de la ligne 11 ?

— Que contient le registre A après exécution de l'instruction de la ligne 13 ?

— Que compare l'instruction de la ligne 14 ?

— Où est-ce que l'instruction de la ligne 15 "saute" si sa condition est satisfaite ?

— Les 3 lignes 17, 18 et 19 réalisent une action assez typique, toutes les trois ensemble. Laquelle ?

3 Assemblons et désassemblons

La section 1.2 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 1.2 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`		

NB : Ce travail 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

On a parfois besoin de s'assurer que le programme binaire qu'on souhaite faire exécuter à un processeur est bien 'le bon'. Dans la vraie vie c'est un programme dédié qui s'occupe de transformer du binaire en son interprétation sous la forme d'instruction de notre processeur. Ce processus s'appelle **désassemblage**. En pratique, les chaînes de compilation fournissent un tel outil. Dans le cas de gcc (que vous avez utilisé en Progammation C), l'outil s'appelle objdump (pour Object Dump, car le code binaire s'appelle parfois aussi "code objet"). Cet outil fait beaucoup de choses différentes et pour désassembler il convient de l'appeler avec l'option objdump -d. Nous en reparlerons dans la séquence de TP "msp430".
Aujourd'hui, on vous demande maintenant de désassembler les codes hexadécimaux suivants.

— Commencez par traduire l'hexadécimal en binaire, puis servez-vous des tableaux en section 1.2 pour reconnaître chaque instruction.

— Attention, comme certaines instructions sont sur 2 octets, il vous faut désassembler au fur et à mesure : le premier octet traduit vous dit comme interpréter l'octet suivant, etc.

Programme 1 – 4C, 6D, 4D, 80, 32, E2, 42, 80

Programme 2 – 6D, 4D, 80, 32, E2, 42, 80

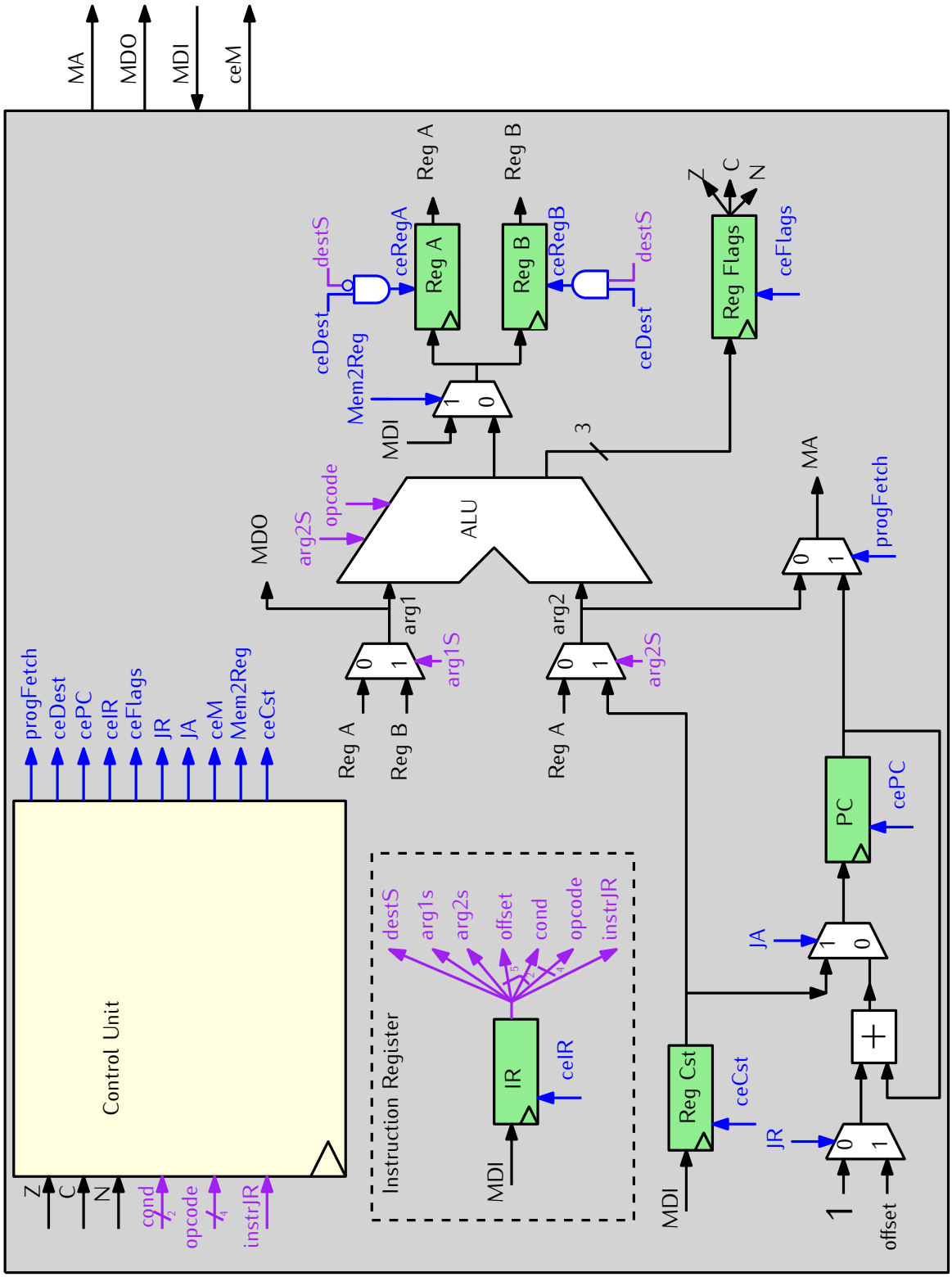

FIGURE 1 – Le datapath de notre micro-machine.

Partie II: Micromachine - Construction

Dans cette partie, nous allons explorer et compléter la micro-machine. En complément des supports de cours qui vous donnent les versions “complètes” de l’automate (Control Unit) et du chemin de donnée (Datapath), on vous donne des fichiers Digital qui implémente une version de la micro-machine capable d’interpréter les instructions suivantes :

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

— Sauts absolus.

Ces fichiers sont disponibles sur moodle, dans la section "Labs" de la page Architecture des Ordinateurs.

La figure 2 donne l’automate correspondant à cette version initiale. La figure 1 (voir pages précédentes) donne une version complète du Datapath qu’on cherche à construire.

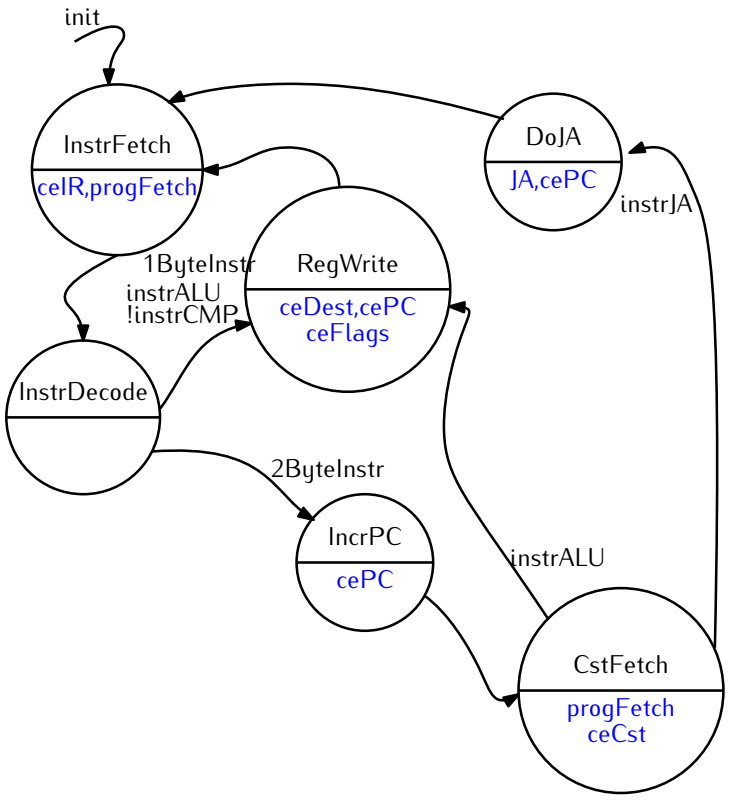

FIGURE 2 – Automate de la version initiale de notre micro-machine Digital.

Dans ce TP, vous allez progressivement augmenter l’implémentation Digitale de l’automate de contrôle afin d’ajouter, dans cet ordre, la prise en charge :

— de l’instruction de comparaison arithmétique ainsi que (dans la même version) les drapeaux (en anglais flags). Voir la section 2.1 ;

— de l’instruction de saut relatif conditionnel. Voir section 2.2 ;

— des instructions de lecture et d’écriture mémoire. Voir section 2.3.

Remarque : Pour chaque version, prenez soin de travailler dans un fichier différent : à chaque instant, vous aurez toujours moyen de retourner vérifier ce que vous avez fait dans l’étape précédente en ayant une micro-machine fonctionnelle pour faire des tests.

```
prog1: 21    -> A
       42    -> B
       B+A   -> B
       JA 5     ;; loop forever here
```

FIGURE 3 – Programme de test `prog1.s` interprétable par la micro-machine initiale.

1 Exploration de la machine existante

1.1 Découverte de l'implémentation existante

Récupérez les fichiers `mm-minimale.dig` `ALU.dig` et `Reg.dig` sur la page moodle du module. Ouvrez ces fichiers dans Digital. 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 `CST`.

Note - encodage des états : 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*" [voir chapitre "Sequential Circuits" du module AC].

Note - activation des registres en écriture : chacun des registres du circuit est un registre à commande de chargement synchrone. Pour un registre `X`, on notera `ceX` le signal *clock enable*. Ainsi pour écrire une valeur *v* dans le registre `X`, il faut faire circuler la valeur *v* sur le bus d'entrée de `X` *puis* activer le signal `ceX` et le maintenir actif jusqu'au prochain front montant d'horloge.

1.2 Exécution d'un programme simple

Récupérez maintenant le fichier `prog1.hex`, qui est l'implémentation (hexadécimale) du prog1 rappelé à la figure 3. Puis :

- Chargez ce programme dans la RAM associée au processeur.
- Pour l'exécuter sur la micro-machine, lancez la simulation du circuit sous Digital.
- Dans la partie "Console", cliquez deux fois sur "Init" (une fois pour le passer à vrai, la seconde pour le passer à faux) pour initialiser les registres. 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, c'est à dire **en même temps** dans digital **ET** sur le dessin de l'automate de la figure 2. Attention, chaque `TOP` de l'horloge, déclenche une transition dans l'automate de contrôle.

2 Construction par étapes de l'automate

2.1 **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 donc ajouter la mise à jour de ces drapeaux, ainsi que le nécessaire à l'exécution de l'instruction de comparaison arithmétique. Les modifications à apporter à l'automate de contrôle pour ces deux éléments sont visibles sur la version de la Figure 4.

2.1.1 Sauvegarde des flags

Vous procéderez comme suit :

- Ouvrez le composant ALU pour identifier dans quel ordre les flags sont rangés. Changez le nom du port Flags pour qu'il fasse apparaître les 3 flags dans le "bon ordre".
- Ajoutez un registre de taille 3 bits, nommé `Flags`, permettant de stocker les flags en sortie de l'ALU.

— Comme dans la Figure 2, ajoutez dans l'état `regWrite` la mise à jour du signal d'écriture dans le registre `Flags` (ceFlags).

Test Modifiez le programme manipulé par votre processeur pour qu'il manipule des valeurs permettant de vérifiez que la valeur des flags est bien mise à jour correctement. Vous choisirez de charger deux valeurs opposées dans A et B pour constater que leur somme fait bien passer le flag Z à 1. Ou vous choisirez des valeurs de façon à provoquer une retenue sortante (flag C à 1).

2.1.2 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 (cf figure 4), l'ajout de cette instruction nécessite l'ajout d'un seul état nommé "NoRegWrite".
Ajoutez maintenant cet état à la partie "Control Unit" de votre micro-machine, en augmentant les parties "Transition Function" et "Output Function".

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.

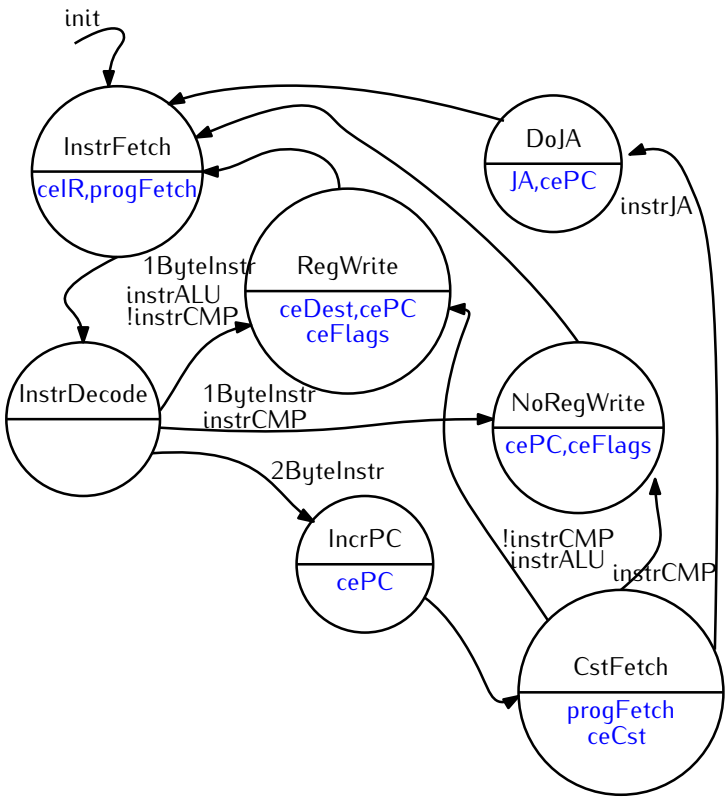

FIGURE 4 – Le cycle de Von Neumann avec **mise à jour des drapeaux** et **instruction de comparaison**

Test Dans votre programme `prog1.hex` ajoutez une instruction de comparaison et testez le programme à nouveau.

2.2 **Version 2 —** Les sauts relatifs (in-)conditionnels

On vous demande maintenant d'ajouter la prise en charge du saut relatif. La Figure 5 décrit l'automate attendu. Cette étape est un peu plus longue car il s'agît de prendre en compte les sauts inconditionnels aussi bien que les saut conditionnels. Pour ces derniers, on doit regarder la valeur des drapeaux (flags) et les comparer avec la condition exprimée dans l'instruction de saut (`IFN`, `IFZ` ou `IFC`).
Cette partie est à réaliser par un petit sous-circuit qui doit calculer le signal `JumpCondTrue`(discuté dans le cours également). Ce signal est vu dans la Figure 5 comme une entrée de l'automate. Il fait pourtant bien partie de la "Control Unit" et pour plus de lisibilité, vous le rangerez donc bien dans le sous-circuit du côté "Control Unit" (sic !).

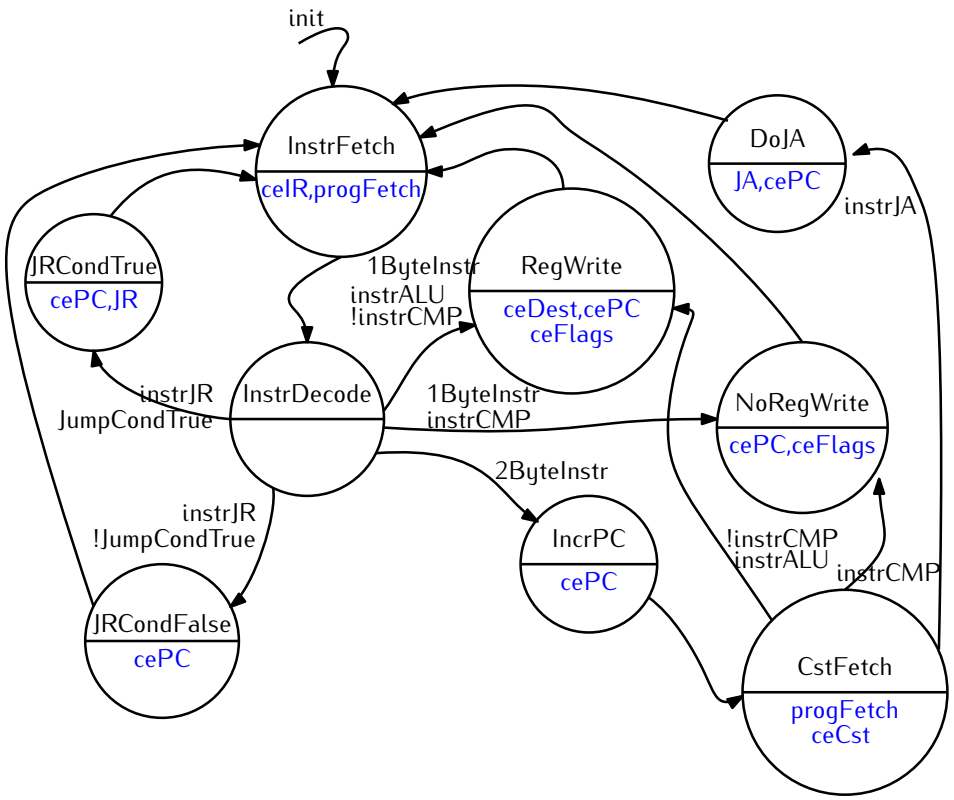

FIGURE 5 – Le cycle de Von Neumann avec **sauts relatifs conditionnels**

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

```
   42 -> B
   7 -> A
f: B-A?
   JR 3 IFN
   B-A -> B
   JR -3       ;; ie, aller au label 'f'
   JR 0        ;; boucle infinie a la suite du programme
```

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

2.3 **Version 3 —** Les accès mémoire (version finale)

Afin de réaliser les accès mémoire, nous vous proposons l'automate de la Figure 7.

— Dans l'état `MemWrite` on se contente de positionner les bus adresse et donnée comme il faut.

— Dans l'état `MemRead`, on positionne l'adresse, puis on lève `ceDest` comme dans l'état `RegWrite`.

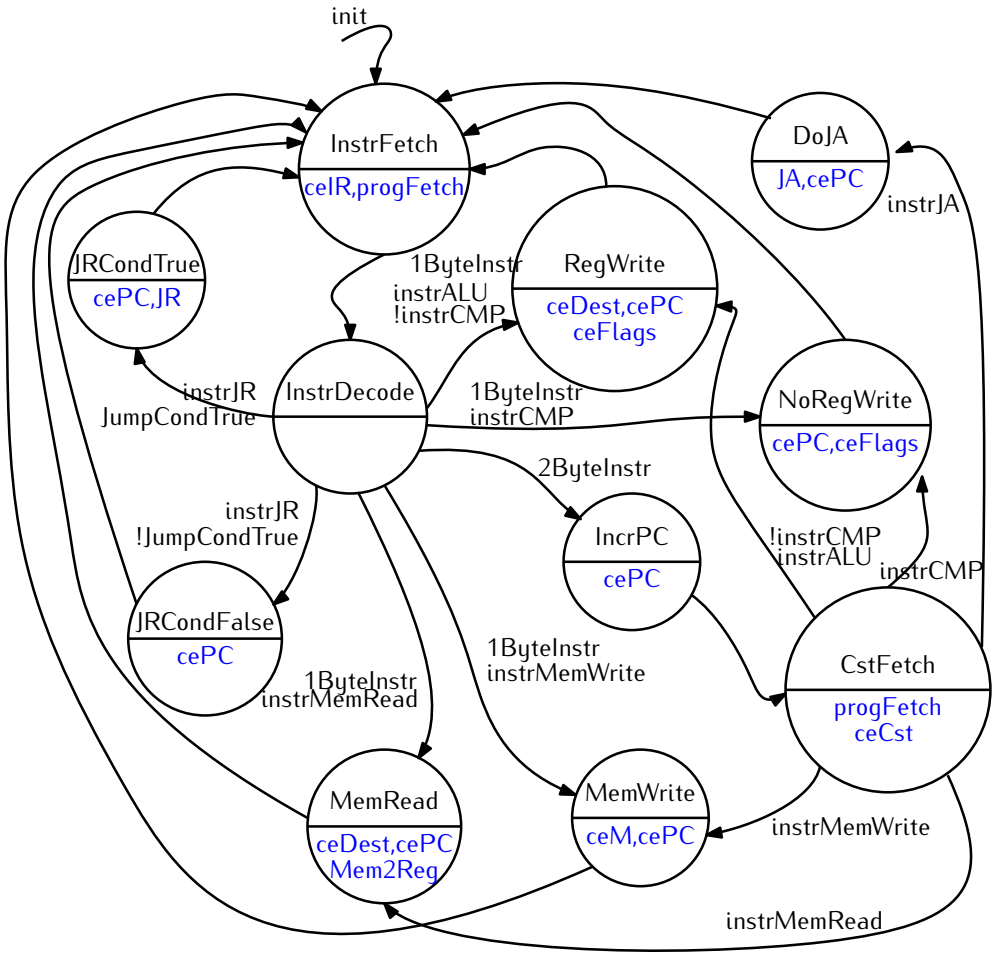

FIGURE 7 – Le cycle de Von Neumann avec **les accès mémoire**

Modifiez le Control Unit et le Datapath de façon à prendre en compte ces accès mémoire.

2.4 Test complet de votre micro-machine

Sur moodle, on vous fournit un script qui assemble un programme dans le langage assembleur de la micro-machine en du binaire (hexadécimal) au format de fichier supporté par Digital. Attention cet assembleur ne traite pas les labels. Vous allez donc d'abord devoir traduire toutes vos instructions `JR label IFN` en remplaçant label par la bonne valeur d'offset. Ceci fait, servez vous de cet assembleur pour tester différents programmes sur votre micro-machine. En particulier, utilisez les exemples discutés dans la première partie de la séquence de TD/TPs.

Partie III: Micromachine - Ajout des Interruptions

1 Spécification / cahier de charges

L'objectif de cette partie est de modifier votre micro-machine pour lui permettre de gérer les interruptions.
Une **interruption** est un événement qui peut être provoqué soit par une cause externe (il y a la plupart du temps des broches pour cela sur la puce du processeur), soit par une cause interne (division par zéro, accès mémoire interdit, accès d'un adresse mémoire délocalisée sur disque (mémoire virtuelle) etc.). Lorsque survient un tel événement, le processeur interrompt (temporairement) ce qu'il était en train de faire (i.e. exécuter son programme) pour sauter à la routine de traitement des interruptions (ISR = *Interrupt Service Routine*). Typiquement, c'est un mécanisme en deux parties :

1. le saut à l'adresse de la routine de traitement,
2. à la fin de la routine, le retour au programme interrompu. Ceci sera déclenché par une **instruction dédiée**.

Le **saut** va devoir faire deux choses d'un seul coup

(i) sauvegarder l'adresse de retour, c'est à dire l'adresse à laquelle il faudra reprendre l'exécution du programme une fois l'interruption traitée ;

(ii) charger l'adresse de la routine d'interruption dans le registre Program Counter (PC).

Il est important que ces deux actions soient faites de façon *atomique* pour ne pas permettre qu'une autre interruption intervienne entre les deux, ce qui écraserait irrémédiablement l'adresse de retour.

Dans ce TD, nous utiliserons une approche simple pour garantir cette atomicité :

> **Fonctionnement du déroutement vers la routine d'interruption —** Lors qu'une interruption est détectée par le processeur, l'instruction en cours est terminée. Puis, l'adresse de retour (i.e. la valeur de PC) est enregistrée dans un nouveau registre dédié. Les drapeaux sont également enregistrés dans un registre dédié. L'adresse de la routine de traitement est enfin chargée dans PC, puis on revient à l'étape Fetch.

2 Description papier de la solution

2.1 Observation

Datapath et automate — À la figure 8, on vous donne le chemin de donnée permettant de prendre en charge les interruptions. Retrouvez-y et entourez les éléments suivants. A chaque fois, expliquez à votre binôme le rôle de l'élément en question. Demandez à un enseignant si vous avez des questions ou des incompréhensions :

- le registre `SavedPC`
- le registre `SavedFlags`
- le signal de contrôle `ceInterrupt`
- le signal de contrôle `ceSave`

À la figure 9, on vous rappelle l'automate (vu en cours) modifié qui permet de prendre en charge les interruptions. Faites maintenant le lien avec le schéma 8. En particulier faites le lien entre les nouveaux signaux d'entrées (IRQ) et de sorties (ceSave et restore) de la Control Unit, visible dans les deux schémas.

Notez que dans cette machine, on met en place un traitant d'interruption unique, toujours stocké à l'adresse 0xA0.

Jeu d'instruction — Dans notre processeur, nous ajoutons une instruction permettant de **revenir du traitant d'interruptions vers la procédure en cours d'exécution au moment de l'interruption**. Cette instruction se nomme `RETI`. Elle sera encodée comme décrit à la figure 6. Allez vérifier dans la table des codes opératoirs (section 1.2) que le code opératoire `1011` n'a effectivement pas encore été utilisé par une instruction existante.

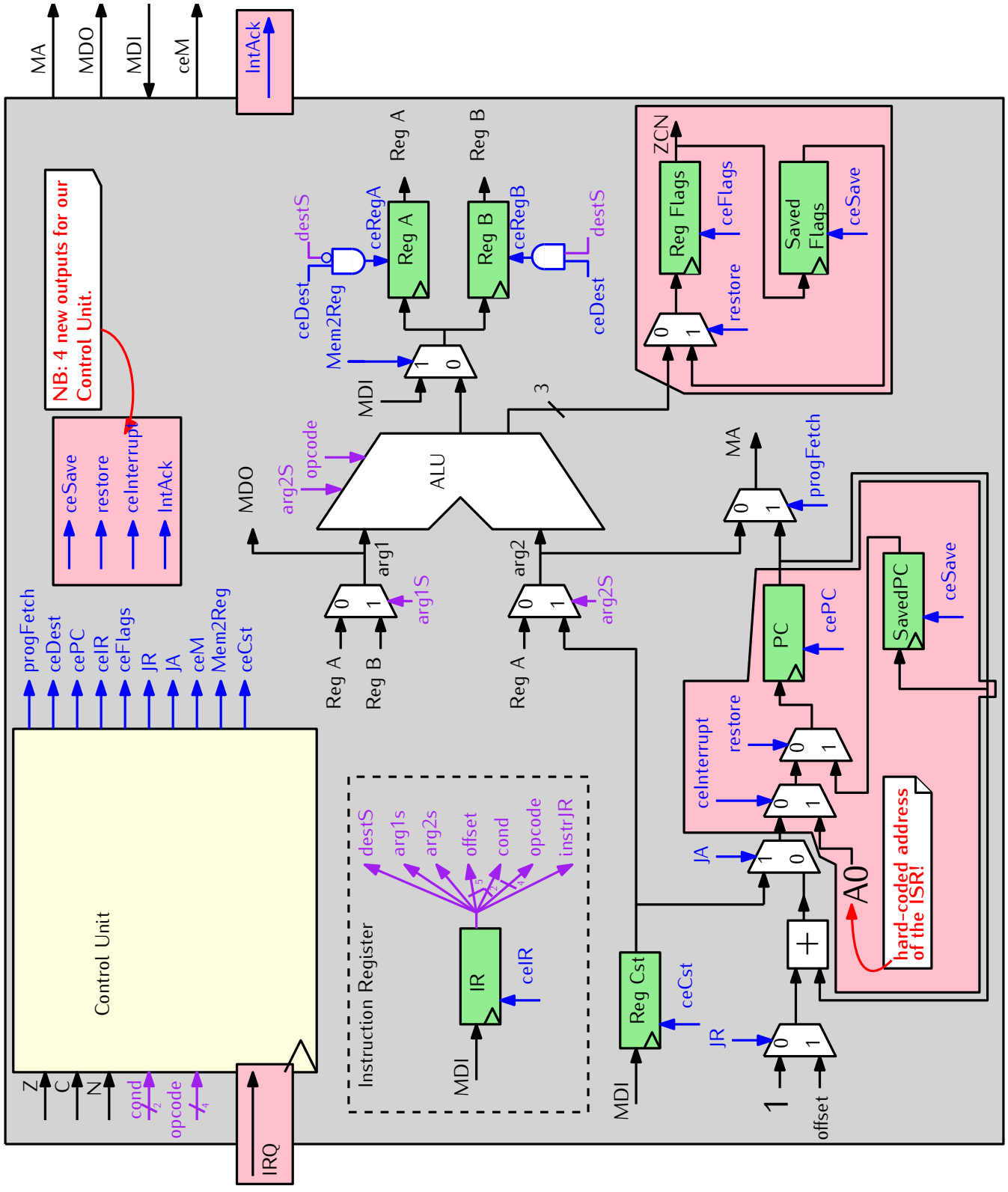

FIGURE 8 – Le chemin de donnée de la micro-machine, augmenté pour pouvoir traiter les interruptions.

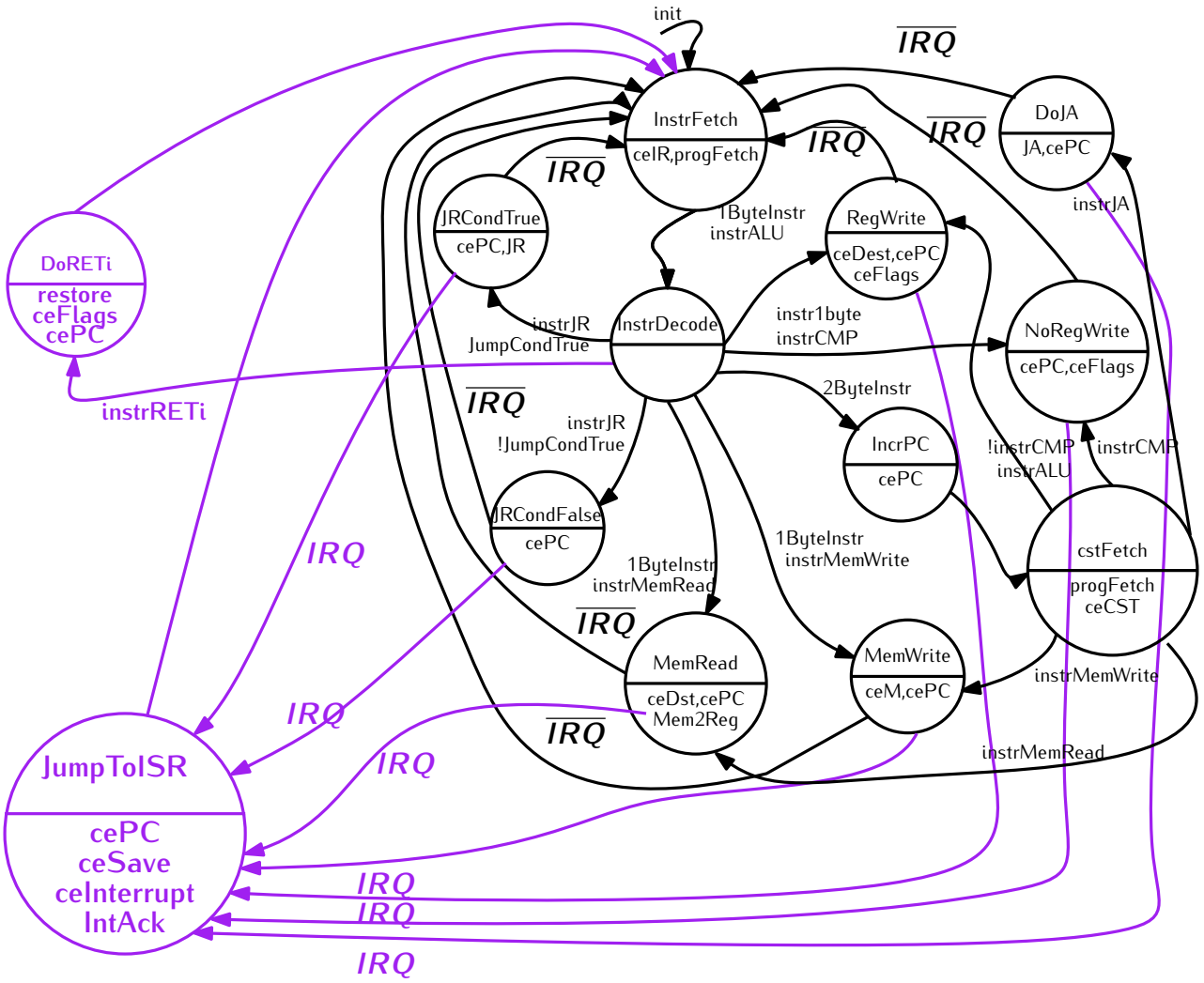

FIGURE 9 – L'automate de contrôle de la micro-machine, augmenté pour pouvoir traiter les interruptions.

codeop	mnémonique	remarques
1011	`reti`	retour de traitant d'interruption

TABLE 6 – `RETI` : descriptif de l'instruction de retour d'interruption.

```
prog1: 21     -> A
       42     -> B
       B+A    -> B
       JA 0x05      ;; loop forever
```

FIGURE 10 – Programme de test `prog1.s` interprétable par la micro-machine initiale.

2.2 "Simulation" à la main

Considérez maintenant le programme de la figure 10 (le même que dans les sections précédentes), stocké à l'adresse 0 de notre mémoire.
Les instructions de la figure 11 stockées en mémoire à l'addresse 0xA0. Elles constituent notre traitant d'interruption.

```
isr: 10     -> A
     reti
```

FIGURE 11 – Traitant d'interruption simple.

On se place maintenant précisément dans la situation suivant :

— La première instruction a été entièrement traitée.

— Le premier octet de la seconde instruction a été lue en mémoire, et on se trouve maintenant dans l'état `InstrDecode`.

— Un signal d'interruption est émis par notre périphérique **à cet instant précis**.

Question : Suivez le comportement de la micro-machine à la suite de la réception de ce signal d'interruption. Indiquez la séquence d'état par lesquels l'automate passe. Vous pousserez l'exercice jusqu'au traitement complet de l'instruction qui est exécutée **tout de suite après l'instruction reti**.

3 Réalisation sur Digital

Sur moodle vous trouvez une solution de la micromachine sans gestion des interruptions. Cette solution comprend également un bouton poussoir permettant à l'utilisateur de demander le déclenchement d'une interruption. Pour le moment, ce signal n'est pas traité par le processeur.

3.1 Modification du chemin de données

1. Ajoutez les deux registres *SavedPC* et *SavedFlags*.
2. Connectez ces registres pour permettre (i) la sauvegarde du PC et des drapeaux, (ii) la restauration des valeurs sauvegardées. Prévoyez des signaux permettant de déclencher ces 4 actions.
3. Ajoutez de quoi charger la valeur 0xA0 dans le PC (saut vers la routine de traitement / ISR). Prévoyez un signal pour déclencher cette action.

3.2 Modification de l'automate

1. Ajoutez l'état *JumpToISR* et les transitions le concernant.
2. Ajoutez l'état *doRETI* et les transitions le concernant. Pour cela il faut une comparaison du code opératoire de l'instruction (issue du registre IR) avec la constante 1011.
3. Ajoutez les signaux sortants pour ces deux états : sauvegarde PC et drapeaux ; reprise PC et drapeaux ; saut vers l'ISR ; acquittement de l'ISR qui dans notre cas est une "simple" émission du signal *IntAck* (pour "Interrupt Acknowledge").

3.3 Testez

Reprenez le programme et l'ISR des figures 10 et 11. Codez l'ISR en hexadécimal. Placez ce code à l'adresse `0xA0`. Testez le fonctionnement de votre micro-machine en suivant le même scénario que sur le papier, plus tôt dans la séance.

Sur moodle, vous trouverez un scénario de test plus complet. Il vous est donné en assembleur, mais aussi en hexadécimal. Testez votre micro-machine avec ce scénario.

4 Bonus : Extensions possibles

Si il vous reste du temps (et l'envie de comprendre plus), voici quelques suggestions d'extensions à apporter à la micro-machine. N'hésitez pas à demander à vos enseignants des précisions.

— 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).