

NOM, Prénom :

DS Architecture des Ordinateurs

25/01/2022

Durée 1h30.

Répondez sur le sujet.

REMPLISSEZ VOTRE NOM TOUT DE SUITE.

Tous documents autorisés, mais en papier seulement.

Crayon à papier accepté, de préférences aux ratures et surcharges.
 Les questions de cours sont de difficulté variable et dans le désordre.
 Les cadres donnent une idée de la taille des réponses attendues.

1 msp430

On rappelle que le msp430 est un micro-contrôleur contenant un processeur 16 bits, que la mémoire est adressée par octets, que la pile est descendante avec pointeur sur case pleine, etc.

Q1. Un processeur 16 bits a les 4 drapeaux classiques : Z, C, N, et V (on rappelle que le drapeau V signale le dépassement de capacité du complément à 2).

Il exécute le calcul suivant : $0xFFFF + 0x0001$ A l'issue de cette opération, lesquels de ces drapeaux valent 1 ?

Q2. Le plan mémoire du msp430 manipulé dans les TPs est rappelé ci-dessous.

Address		Access
FFFFh	Interrupt Vector Table	Word/Byte
FFC0h		
FFBFh	Flash/ROM	Word/Byte
3100h		
30FFh		
1100h	RAM	Word/Byte
	Reserved	No access
01FFh	16-Bit Peripheral Modules	Word
0100h		
00FFh	8-Bit Peripheral Modules	Byte
0010h		
000Fh		
0000h	Special Function Registers	Byte

Quelle est la taille de la RAM disponible (en octets) ?

Quelle est la place maximale (théorique) disponible pour le stockage de la pile (en ko) ?

Quelle est la place maximale (théorique) disponible pour le code (en ko) ?

Q3. On considère un certain processeur qui adresse sa mémoire par octet (comme le Pentium, ARM et le MSP430). Ce processeur dispose d'une instruction POP codée sur 2 octets. Les cases de la pile font 2 octets. La pile descend en mémoire, c'est à dire qu'empiler diminue le pointeur de pile. L'encodage des entiers est *little-endian*, c.à.d que les octets de poids faibles sont encodés aux adresses plus petites. Le pointeur de pile pointe vers le dernier élément de la pile.

Dans le tableau ci-dessous, remplissez la ligne spécifiant les valeurs **après** l'exécution de l'instruction POP R7. Laissez vides les cases inchangées.

	Registres (16 bits)					Cases mémoires (octets)									
	PC	SP	R5	R6	R7	65	66	67	68	69	6a	6b	6c	6d	6e
avant POP R7	0xa0	0x6a	6	d	a	0	d	e	a	d	b	e	e	f	0
après POP R7															

Q4. Un MSP430 exécute le code ci-dessous à gauche, qui prétend réaliser quelque chose de bien utile.

Remarque : le MSP 430 accède à la mémoire en mode **little-endian**, c'est-à-dire que les valeurs 16bit sont stockées tel que l'octet de poids faible se trouve à l'adresse faible.

A droite, vous trouverez le contenu d'un extrait de la mémoire avant l'exécution du programme.

```

mov #0x0a04, R3
mov #0x0a0a, R4
mov #0x0000, R8
mov #0x0003, R9      ; vector space dimension
loop:
  mov @R3, R6
  mov @R4, R7
  sub R6, R7
  jn negate
  jmp accumulate

negate:      inv R7          ; r7 = -1 * r7
             add #1, R7     ; (bitwise invert + increment)

accumulate:
  add R7, R8
  add #2, R3
  add #2, R4
  sub #1, R9
  cmp #0, R9
  jnz loop

fin:

```

0x0a03	00
0x0a04	01
0x0a05	03
0x0a06	02
0x0a07	0a
0x0a08	02
0x0a09	00
0x0a0a	03
0x0a0b	03
0x0a0c	01
0x0a0d	0a
0x0a0e	02
0x0a0f	01

Le jeu d'instruction du msp430 est rappelé dans la page suivante. Donnez dans le tableau suivant les différentes valeurs de R8 obtenues à chaque itération de la boucle, c'est à dire au moment où PC = loop. Remarques : certaines cases à la fin du tableau peuvent être vides.

--	--	--	--	--	--

Quelles sont les valeurs de R3, R4, R6, R7, R8 et R9 à la fin du programme ?

R3 : R4 : R6 : R7 : R8 : R9 :

Expliquez en une seule phrase l'objectif du programme :

Liste compacte des instructions MSP430

Mnemonic	Description	Operation	V	N	Z	C	
ADC (.B)	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD (.B)	src, dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC (.B)	src, dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND (.B)	src, dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC (.B)	src, dst	Clear bits in destination	.not.src .and. dst → dst	-	-	-	-
BIS (.B)	src, dst	Set bits in destination	src .or. dst → dst	-	-	-	-
BIT (.B)	src, dst	Test bits in destination	src .and. dst	0	*	*	*
BR	dst	Branch to destination	dst → PC	-	-	-	-
CALL	dst	Call destination	PC+2 → stack, dst → PC	-	-	-	-
CLR (.B)	dst	Clear destination	0 → dst	-	-	-	-
CLRC		Clear C	0 → C	-	-	-	0
CLR N		Clear N	0 → N	-	0	-	-
CLR Z		Clear Z	0 → Z	-	-	0	-
CMP (.B)	src, dst	Compare source and destination	dst - src	*	*	*	*
DADC (.B)	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD (.B)	src, dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC (.B)	dst	Decrement destination	dst - 1 → dst	*	*	*	*
DECD (.B)	dst	Double-decrement destination	dst - 2 → dst	*	*	*	*
DINT		Disable interrupts	0 → GIE	-	-	-	-
EINT		Enable interrupts	1 → GIE	-	-	-	-
INC (.B)	dst	Increment destination	dst + 1 → dst	*	*	*	*
INCD (.B)	dst	Double-increment destination	dst + 2 → dst	*	*	*	*
INV (.B)	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		-	-	-	-
JEQ/JZ	label	Jump if equal/Jump if Z set		-	-	-	-
JGE	label	Jump if greater or equal		-	-	-	-
JL	label	Jump if less		-	-	-	-
JMP	label	Jump	PC + 2 x offset → PC	-	-	-	-
JN	label	Jump if N set		-	-	-	-
JNC/JLO	label	Jump if C not set/Jump if lower		-	-	-	-
JNE/JNZ	label	Jump if not equal/Jump if Z not set		-	-	-	-
MOV (.B)	src, dst	Move source to destination	src → dst	-	-	-	-
NOP		No operation		-	-	-	-
POP (.B)	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	-	-	-	-
PUSH (.B)	src	Push source onto stack	SP - 2 → SP, src → @SP	-	-	-	-
RET		Return from subroutine	@SP → PC, SP + 2 → SP	-	-	-	-
RETI		Return from interrupt		*	*	*	*
RLA (.B)	dst	Rotate left arithmetically		*	*	*	*
RLC (.B)	dst	Rotate left through C		*	*	*	*
RRA (.B)	dst	Rotate right arithmetically		0	*	*	*
RRC (.B)	dst	Rotate right through C		*	*	*	*
SBC (.B)	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC		Set C	1 → C	-	-	-	1
SETN		Set N	1 → N	-	1	-	-
SETZ		Set Z	1 → C	-	-	1	-
SUB (.B)	src, dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC (.B)	src, dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		-	-	-	-
SXT	dst	Extend sign		0	*	*	*
TST (.B)	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR (.B)	src, dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

2 Programmation assembleur micro-machine

On rappelle la description de la micro-machine étudiée en cours et en TD.

Nous travaillons avec un processeur pur 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 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 reçoit 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.

Les instructions sont toutes encodées en un octet comme indiqué ci-dessous. Pour celles qui impliquent une constante (de 8 bits), cette constante occupe la case mémoire suivant celle de l’instruction.

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				

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

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...).

Encodage des conditions du saut relatif conditionnel :

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

Le programme ci-dessous calcule la somme de deux nombres encodés sur 16 bits. Les deux nombres sont stockés initialement en mémoire, chacun occupant 2 octets. X est stocké aux adresses 100 (octet de poids faible) et 101 (octet de poids fort), tandis que Y est stocké aux adresses 102 et 103.

```

1 16bitSum:
2 ; let's start by adding the "first" byte of X and the "first" byte of Y
3 first-half:
4     *100 -> A      ; load right-most byte of X
5     *102 -> B      ; load right-most byte of Y
6     B+A -> A       ; compute right-most byte of the sum
7     JR ... IFC     ; is there a carry ?
8     A -> *104      ; store right-most byte of result
9     0 -> A         ; A contains 0-carry
10    JA second-half
11    A -> *104      ; store right-most byte of result
12    1 -> A         ; A contains 1-carry
13 ; now let's add the "second" byte of X to the "second" byte of Y
14 second-half:
15    *101 -> B      ; load left-most byte of X
16    B+A -> A       ; propagate carry
17    *103 -> B      ; load left-most byte of Y
18    B+A -> A       ; compute left-most byte of result
19    A -> *105      ; store left-most byte of result
20 end:

```

Q5. L'instruction JR de la ligne 7 a été intentionnellement laissée incomplète. La destination du saut est l'instruction de la ligne 11, i.e. immédiatement après le JA second-half. Compléter l'instruction :

JR IFC

Q6. On suppose maintenant qu'on exécute ce programme sur les nombres X = 0x72B1 et Y = 0x2FE6. Le contenu initial de la mémoire est donné ci-dessous. Complétez les cases restantes pour donner le contenu de la mémoire après l'exécution du programme, i.e. lorsque le programme atteint l'étiquette "end".

address	content
100	0xB1
101	0x72
102	0xE6
103	0x2F
104	...
105	...
106	...

Q7. Indiquez le contenu des registres CPU au moins point (une fois le programme terminé) :

— register A =

— register B =

Q8. A l'aide de la description de l'ISA donnée plus haut, donnez ci-dessous l'encodage binaire et hexadécimal des 4 premières instructions du programme.

3 Une pile pour la micro-machine

On considère la micromachine comme elle a été conçue en TP, sans l'extension gérant les interruptions. Les détails nécessaires au bon déroulement des questions sont rappelés plus loin. On souhaite ajouter les éléments suivants permettant l'exécution des fonctions, à savoir :

- une instruction `call` permettant d'appeler un sous-programme ;
- une instruction `return` permettant de revenir depuis à un sous-programme vers le programme appelant ;
- une pile permettant de stocker des contenus (registres ou constantes). Cette pile est une zone mémoire particulière manipulable avec deux instructions `pop` (pour dépiler) et `push` (pour empiler).

Contrairement au msp430, la pile permet de stocker des valeurs sur 1 octet. Autrement dit, l'instruction `push` (respectivement `pop`) a pour effet de décrémenter (respectivement incrémenter) `SP` de 1.

Voici un exemple de programme que ces instructions permettent d'écrire :

```

push A           // empiler registre A, pour le sauvegarder
push B           // empiler registre B, pour le sauvegarder
*21 -> A         // chercher les opérandes de la mémoire
*22 -> B
call addition    // appeler la fonction d'addition
A -> *23         // enregistrer le résultat en mémoire
pop B            // restaurer les registres
pop A
    
```

addition:

```

B+A -> A        // Effectue l'addition
ret              // Retour à la routine 'appelant'
    
```

Cette pile s'inspire de la pile du MSP430, que nous avons aussi vu en TP :

- Pour gérer l'état de la pile, la micromachine aura donc un registre `SP` (*Stack Pointer*) indiquant l'adresse de la dernière valeur empilée.
- Empiler va décrémenter le pointeur de la pile `SP`, donc on va en "sense inverse".
- Au démarrage de la micromachine (reset), `SP` est chargé avec la valeur `0xFF`.
- On procède par pré-décrémentation pour l'empilement (on décrémente `SP` puis on écrit sur la pile) et post-incrémentation pour le dépilement (on lit la valeur de la pile puis on incrémente `SP`).

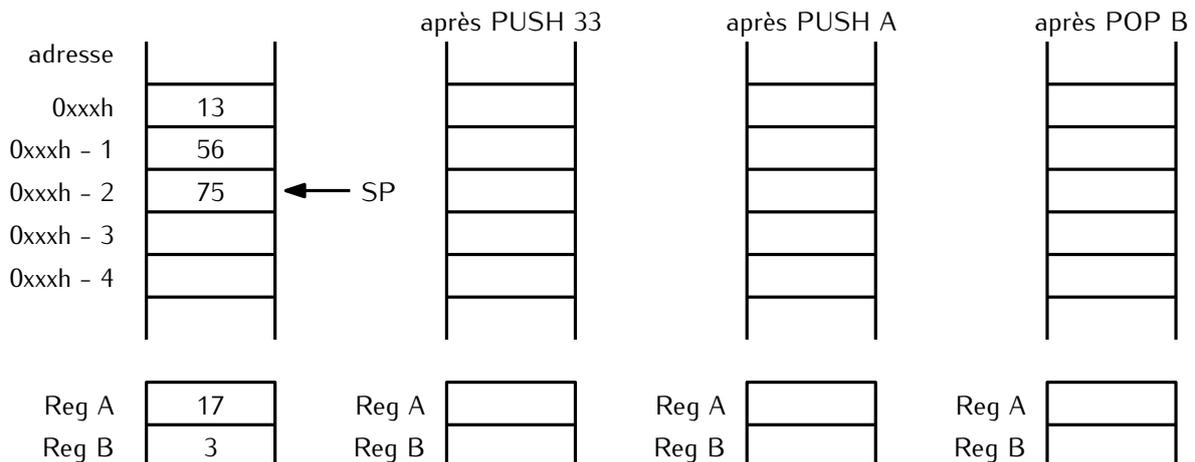
Q9. On fait les hypothèses suivantes quant à l'état de la micro-machine *avant* d'exécuter le petit programme qui suit :

- Le registre A contient la valeur 17
- Le registre B contient la valeur 3
- Les dernières valeurs empilées sont 13, 56 et 75 (elles ont été empilées dans cet ordre)

```

push 33
push A
pop B
    
```

Après chaque opération, indiquez l'état de la pile, la position du `SP` (flèche), et le contenu des registres A, B.



Q10. L'instruction CALL sera encodée de manière similaire à une instruction JA (bit de poids fort à 0 et opcode sur 4 bits). CALL aura le code opératoire 1010, la destination sera encodée sur 8 bits dans l'octet qui suit l'instruction (comme c'est le cas pour un saut absolu). On redonne ci-dessous la table détaillant l'encodage des instructions micromachine :

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

On considère l'extrait de programme suivant où la colonne de gauche donne l'adresse de chacun instruction :

<0x10> CALL func

...

<0x30> func:

...

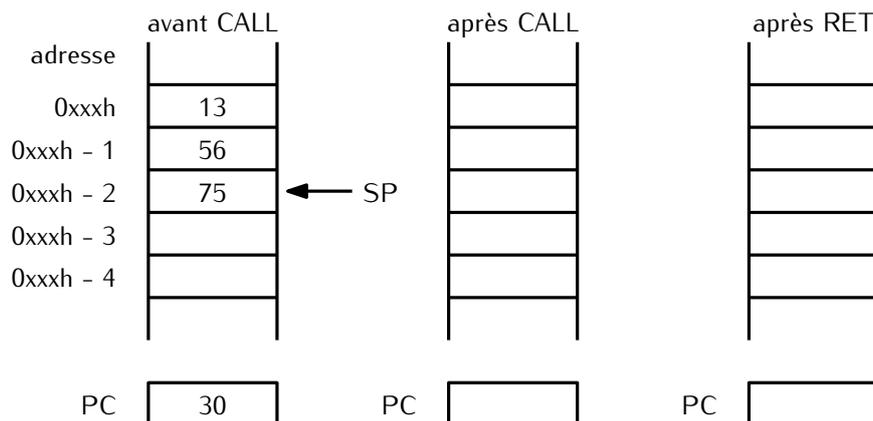
Donnez l'encodage en binaire de l'instruction CALL :

Q11. Soit le fragment de code suivant — ici on donne l'adresse de chaque instruction en début de ligne :

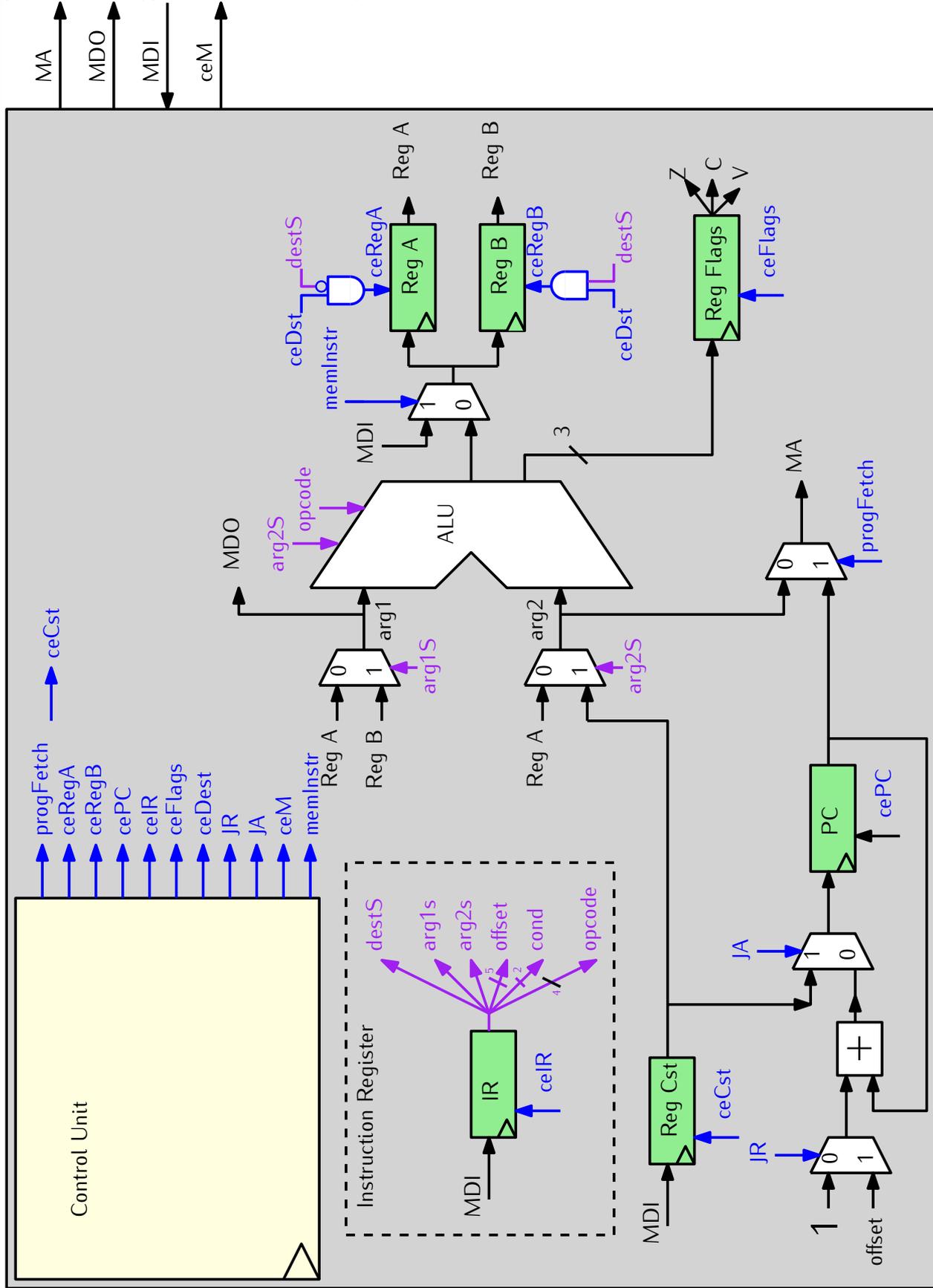
```
0x30    call 0x60
0x32    A -> *23
...
...
0x60    B+A -> A
0x61    ret
```

Dans la figure suivante, affichez l'état de la pile après (i) l'exécution de l'instruction CALL et (ii) après l'exécution de l'instruction RET.

Pour chaque cas, indiquez l'état de la pile, la position du SP (flèche), et le contenu du registre PC.



Q12. Dans la figure ci-dessous vous retrouvez le chemin des données de la micromachine, sans la gestion de la pile. Ajoutez le registre SP gérant la pile et connectez le de telle manière à ce que les opérations du cahier des charges soient possibles (en supposant un automate adapté).



Q13. Dans la figure suivante vous retrouvez l'automate de la micromachine.

Modifiez l'automate pour rendre possible **une seule** des instructions supplémentaires associées à la pile, à savoir l'instruction **CALL** : ajoutez les états nécessaires et connectez les. Ajoutez les signaux **de sortie** nécessaires en veillant à réutiliser les noms que vous avez utilisés lors de la question précédente.

Remarque : rappelons les opérations effectuées par l'instruction **CALL <destination>** :

```

SP-1 -> SP
PC -> @SP // @SP est équivalent à Mémoire[SP]
<destination> -> PC
    
```

Remarque : il convient de réutiliser les états existants quand cela est possible.

