NOM, Prénom :

DS Architecture des Ordinateurs

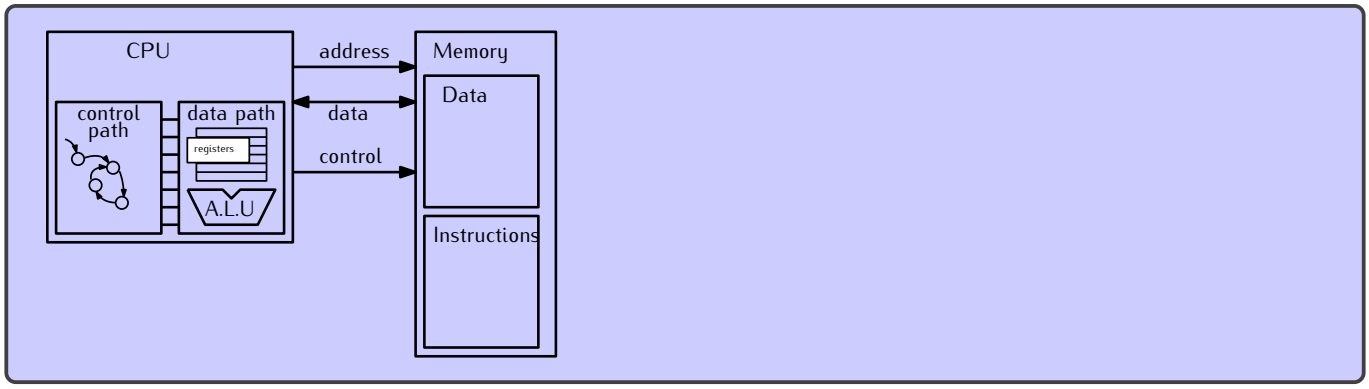
31/01/2023

Durée 1h30.
Répondez sur le sujet.
REMPLISSEZ VOTRE NOM TOUT DE SUITE.
Seul document autorisé : 1 feuille recto-verso manuscrite !
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 Questions de cours - Généralités

Q1.

Dessinez ici une machine de von Neumann :



Q2. Pour chaque affirmation indiquez si vous la considérez vraie ou fausse :

V F a Dans une machine de von Neumann, il y a un bit de chaque case registre qui dit si ce registre contient un entier signé ou non signé.

V F b Dans une machine de von Neumann, un programme peut modifier son propre code.

V F c Dans une machine de von Neumann, il y a un bit de chaque case mémoire qui dit si la case contient une instruction ou une donnée.

V F d Un cycle de von Neumann exécute une instruction et une seule.

F - V - F - V

Q3. On considère un certain processeur qui adresse sa mémoire par octet (comme le pentium et le MSP430). Ce processeur dispose d'une instruction PUSH, et lorsqu'on fait PUSH R3 on observe les changements suivants.

	Registres (16 bits)					Cases mémoires (octets)									
	PC	SP	R3	R5	R6	27	28	29	2a	2b	2c	2d	2e	2f	30
avant PUSH R3	0x0c	0x2c	6	d	e	0	d	e	a	d	b	e	e	f	0
après PUSH R3	0x0e	0x28	6	d	e	0	d	e	a	d	6	0	0	0	0

Que pensez-vous des assertions suivantes ?

V F a La pile monte en mémoire, c'est à dire qu'empiler augmente le pointeur de pile

V F b Les cases de pile font 4 octets

V F c Le pointeur de pile pointe vers la prochaine case vide de la pile

V F d L'instruction PUSH R3 s'encode en 4 octets

F - V - V - F

2 MSP430

On rappelle que le MSP430 est un micro-controlleur contenant un processeur 16 bits, que la mémoire est adressée par octets, que la pile est descendante avec pointeur sur case pleine, etc.

Q4. 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 : 0x7FFF + 0x0001 A l'issue de cette opération, lesquels de ces drapeaux valent 1 ?

Cette somme donne 0xF000, ce qui veut dire :

Z = 0 — N = 1 — C = 0 — V = 1

Q5. On vous donne ci-dessus les informations sur le plan mémoire d'un micro-contrôleur de la famille MSP430 (plus gros que celui manipulé en TP).

1.11 Memory Map – Uses and Abilities

This memory map represents the MSP430F5438 device. Though the address ranges differs from device to device, overall behavior remains the same.

Can generate NMI on read/write/fetch								
Generates PUC on fetch access								
Protectable for read/write accesses								
Always able to access PMM registers from[1]; Mass erase by user possible								
Mass erase by user possible								
Bank erase by user possible								
Segment erase by user possible								
Address Range	**Name and Usage**	Properties						
00000h-00FFFh	Peripherals with gaps							
00000h-000FFh	Reserved for system extension							
00100h-00FEFh	Peripherals						x	
00FF0h-00FF3h	Descriptor type[2]						x	
00FF4h-00FF7h	Start address of descriptor structure						x	
01000h-011FFh	BSL 0	x				x		
01200h-013FFh	BSL 1	x				x		
01400h-015FFh	BSL 2	x				x		
01600h-017FFh	BSL 3	x			x	x		
017FCh-017FFh	BSL Signature Location							
01800h-0187Fh	Info D	x						
01880h-018FFh	Info C	x						
01900h-0197Fh	Info B	x						
01980h-019FFh	Info A	x						
01A00h-01A7Fh	Device Descriptor Table						x	
01C00h-05BFFh	RAM [redacted]							
05B80-05BFFh	Alternate Interrupt Vectors							
05C00h-0FFFFh	Program	x	x[1]	x				
0FF80h-0FFFFh	Interrupt Vectors							
10000h-45BFFh	Program	x	x	x				
45C00h-FFFFFh	Vacant							x[3]

(1) Access rights are separately programmable for SYS and PMM.
(2) Fixed ID for all MSP430 devices. See Section 1.13.1 for further details.
(3) On vacant memory space, the value 03FFFh is driven on the data bus.

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

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

- RAM = 0x5bff - 0x1c00 = 16ko
- 2 zones :
  - 0xffff-0x5c00 = 41ko
  - 0x10000-0x45bff = 215ko
  - total = 256ko

Q6. Un MSP430 exécute le code de la colonne de gauche (qui n'a pas la prétention de réaliser quelque chose d'utile).
Pour chaque ligne, remplissez la table avec les valeurs des registres et de la mémoire après l'exécution de l'instruction correspondante (vous pouvez laisser en blanc les cases inchangées).
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. L'instruction CALL est codée en 4 octets.

Instruction	Registres (16 bits)					Cases mémoires (1 adresse = 1 octet)					
	PC	SP	R3	R5	R6	0x28	0x29	0x2a	0x2b	0x2c	0x2d
(état initial)	0x0400	0x2c	6	3	2	0	0	0	0	0	0
0x0400 `MOV #4, R3`											
0x0404 `PUSH R5`											
0x0406 `POP R6`											
0x0408 `CALL 0x0500`											

Instruction	Registres					Cases mémoires					
	PC	SP	R3	R5	R6	0x28	0x29	0x2a	0x2b	0x2c	0x2d
(état initial)	0x0400	0x2c	6	3	2	0	0	0	0	0	0
0x0400 `MOV #4, R3`	0x404		4								
0x0404 `PUSH R5`	0x406	0x2a						0x03	0x00	0	0
0x0406 `POP R6`	0x408	0x2c			3						
0x0408 `CALL 0x0500`	0x500	0x2a						0x0c	0x04	0	

Q7. On considère le bout de code MSP430 suivant (en rappelant que `@r5` dénote le contenu de la case mémoire pointée par `r5`). C'est l'implémentation d'une fonction qui reçoit en paramètres `r5`, `r6` et `r7`.

```
mysterycode:
  mov @r5, r10
  add #2, r5
  mov @r5, r11
  mov @r6, r12
  add #2, r6
  mov @r6, r13
  add r13, r11
  add r12, r10
  jnc nocarry                    ; jump if not carry
  add #1, r11
nocarry:
  mov r10, @r7
  add #2, r7
  mov r11, @r7
  ret
```

Que pensez vous des assertions suivantes ?

V F [a] Ce code lit deux cases mémoires successives pointées par r5.

V F [b] Ce code écrit deux cases mémoires successives pointées par r7.

V F [c] Ce code modifie les registres `r5`, `r6` et `r7`.

V F [d] Ce code contient une boucle.

V — V — V — F

Q8. Décrivez en 2 ou 3 phrases la fonction, fort utile, que réalise le code de la question précédente. Soyez précis et complet.

ce programme calcule la somme de deux entiers 32 bits stockés en mémoire. Si $a1$ (resp. $a2$) est l'adresse contenue dans `r5` (resp. `r6`), alors le premier (resp. second) entier est supposé rangés aux octets d'adresses $a1$ à $a1+4$ (resp. $a2$ à $a2+4$). Le résultat est écrit à l'adresse initialement stockée dans `r7`.

Q9. 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.

```
        .section .init9

main:
        mov.b &0x0a03, R9
        mov #0x0a04, R10
        mov #0, R11
        mov #0, R8
loop:
        cmp R9, R8
        jz end
        mov @R10, R12
        cmp R11, R12
        jl continue
        mov R12, R11
continue:
        incd R10
        inc R8
        jmp loop

end:    jmp end
```

0x0a03	05
0x0a04	01
0x0a05	03
0x0a06	02
0x0a07	0a
0x0a08	02
0x0a09	00
0x0a0a	03
0x0a0b	03
0x0a0c	07
0x0a0d	0a

Le jeu d'instruction du MSP430 est rappelé dans la page suivante. Donnez dans le tableau suivant les différentes valeurs de `R8` et `R10` 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.

R8								
R10								

Quelles sont les valeurs de R8, R9, R10, R11, R12 à la fin du programme ?
R8 : ____ R9 : ____ R10 : ____ R11 : ____ R12 : ____

Expliquez en une seule phrase l'objectif du programme. Ici aussi, soyez concis mais précis :

R8	0	1	2	3	4	5
R10	0x0a04	0x0a06	0x0a08	0x0a0a	0x0a0c	0x0a0e

Quelles sont les valeurs de R8, R9, R10, R11, R12 à la fin du programme ?
R8 : **5** R9 : **5** R10 : **0xa0e** R11 : **0x0a07** R12 : **0x0a07**

Expliquez :
Il calcule le max d'un tableau de 5 éléments. Le tableau est rangé à partir de l'adresse `0xa04`. Le nombre d'élément du tableau est rangé à l'adresse `0xa03`.

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
CLRN		Clear N	0 → N	–	0	–	–
CLRZ		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	*	*	*	*

3 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`	`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.

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	(toujours)	`IFZ` si zéro	`IFC` si carry	`IFN` si négatif

Le programme ci-dessous calcule une multiplication de deux nombres `n` et `m`. Les deux nombres sont stockés initialisés en mémoire au début du programme, aux adresses 162 et 164 (en décimal). Le résultat, initialisé à 0 aux lignes 6 et 7, sera stocké en mémoire à l'adresse 160 (en décimal également).

Le programme est stocké en mémoire à partir de l'adresse 0. Les numéros à gauche sont des numéros de ligne, pas des adresses.

```
 1  mult:
 2          5 -> A                  ; initialize 'n' to 5
 3          A -> *162               ; 'n' is stored at address 162
 4          4 -> A                  ; initialize 'm' to 4
 5          A -> *164               ; 'm' is stored at address 164
 6          0 -> A                  ; initialize 'result' to 0
 7          A -> *160               ; 'result' is stored at address 160
 8
 9  loop:
10          *164 -> A               ; get 'm' into A
11          A-0?                    ; m==0 ?
12          JR 2 IFZ                ; then jump to the end
13          JR 3                    ;
14          JA .....                ;
15
16          *162 -> A               ; get 'n' into A
17          *160 -> B               ; get 'result' into B
18          B+A -> B                ; n+result --> B
19          B -> *160               ; move n+result back to 'result' in memory
20
21          *164 -> A               ; get 'm' into A
22          A-1 -> A                ; decrement A
23          A -> *164               ; write 'm' back to memory
24          JA .....                ; jump back to begining of loop
25
26  end:
27          JA .....                ; loop forever here
```

Q10. Les trois instructions `JA` des lignes 14, 24 et 27 ont été intentionnellement laissées incomplètes. Leurs destinations sont indiquées en commentaires. Calculez les addresses cibles et compléter les 3 instructions, dans l'ordre :

ligne 14 — `JA`	
ligne 24 — `JA`	
ligne 27 — `JA`	

ligne 14 — `JA`	35 / 0x23
ligne 24 — `JA`	12 / 0x0c
ligne 27 — `JA`	35 / 0x23

Q11. A l'aide de la description de l'ISA donnée plus haut, donnez ci-dessous l'encodage binaire et hexadecimal des 5 premières instructions à partir du label "loop" :

*164 -> A		
A-0 ?		
JR 2 IFZ		
JR 3		
JA		

*164 -> A	0 1101 100 10100100	0x6c A4
A-0 ?	0 0110 100 00000000	0x34 0x00
JR 2 IFZ	1 01 00010	0xA2
JR 3	1 00 00011	0x83
JA 36	0 1111 000	0x78
	0010 0100	0x24

4 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 suivant 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.
NB : on s'attend à pouvoir faire `push` et `pop` de registres uniquement.
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 régistres
        pop A

addition:
        B+A -> A            // Effectueur 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`.
- Au démarrage de la micromachine (reset), `SP` est chargé avec la valeur `0xFF`.
- On procède par pré-décrementation 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`).

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				

Q12. Donnez un encodage pour les instructions `pop` et `push`.

`pop` :

`push` :

Q13. Donnez un encodage pour les instructions `pop` et `push`.

`pop` :

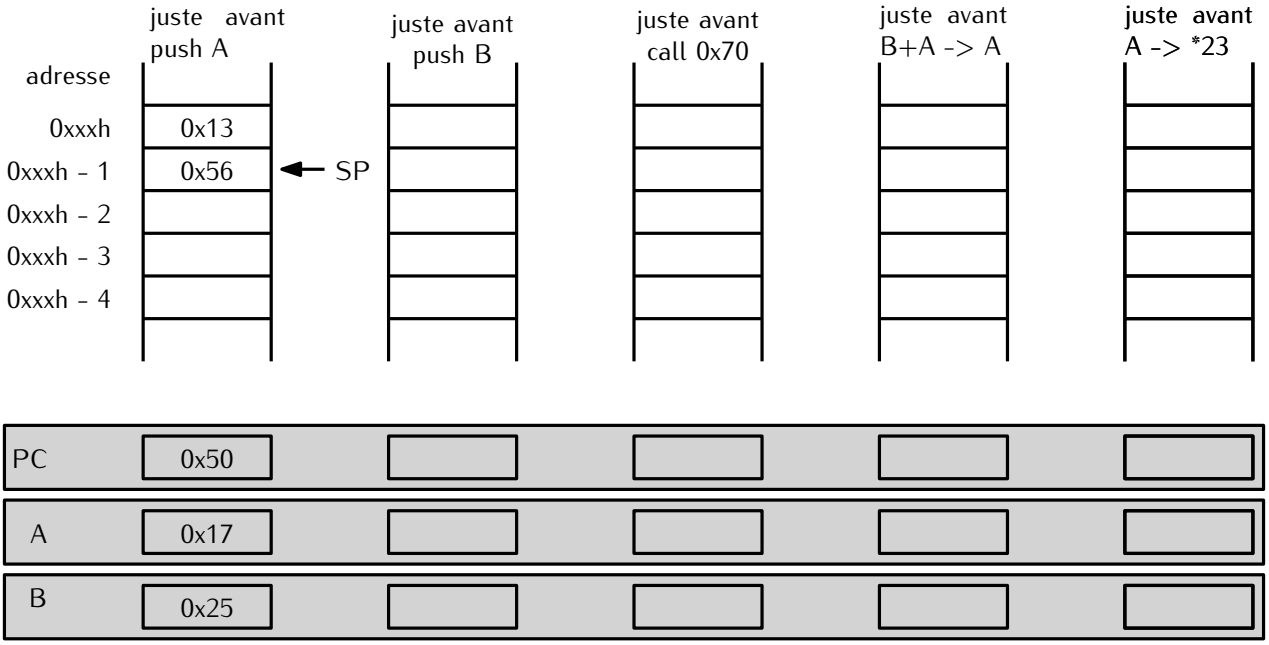
`push` :

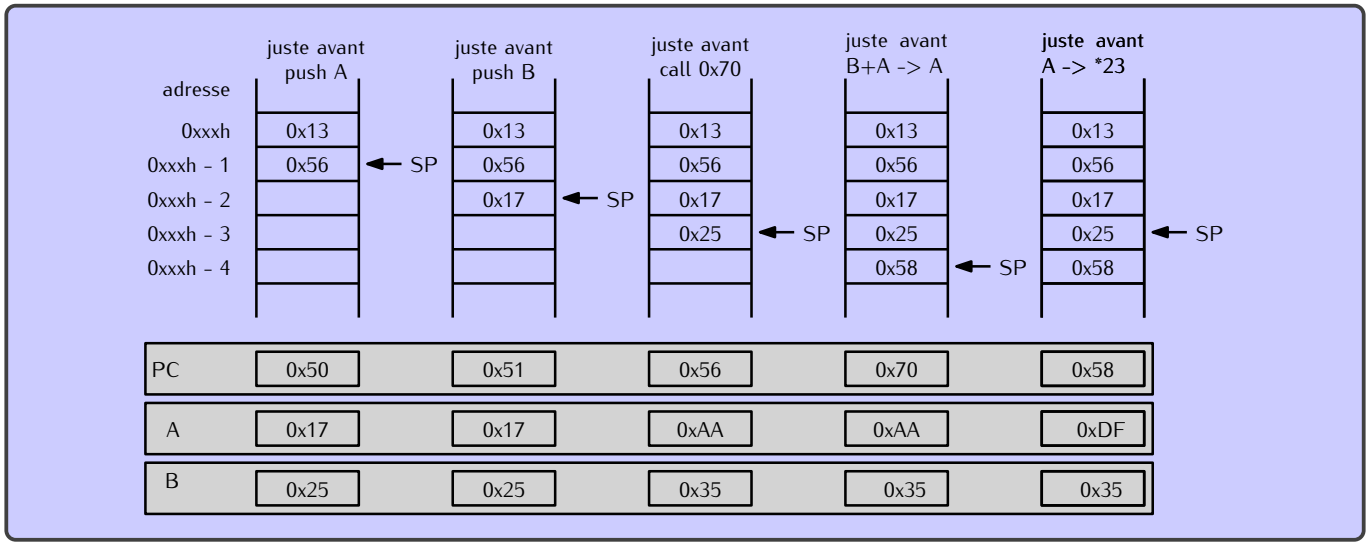
Q14. Soit le fragment de code suivant — ici on donne l'adresse de chaque instruction en début de ligne :

```
0x50    push A
        push B
        *21  -> A
        *22  -> B
        call 0x70
        A -> *23
 ...
 ...
0x70    B+A -> A
        ret
```

0x20	0xff
0x21	0xaa
0x22	0x35
0x23	0xff
0x24	0xff

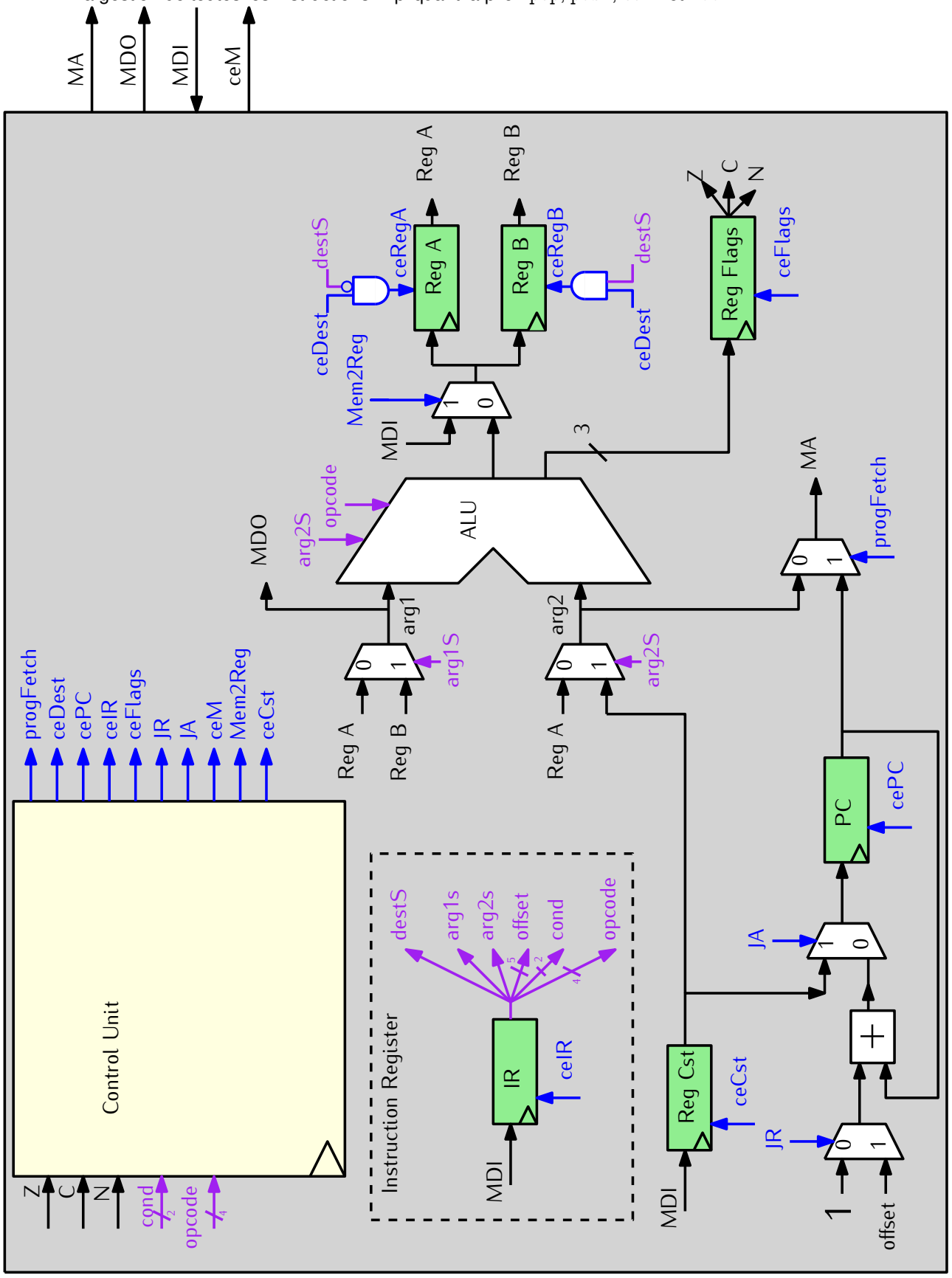
Dans la figure suivante, affichez l'état de la pile **avant** l'exécution de l'instruction indiquée en haut.
Pour chaque cas, indiquez l'état de la pile, la position du SP (flèche), le contenu du registre PC ainsi que le contenu des registres A et B.




juste avant push A
juste avant push B
juste avant call 0x70
juste avant B+A -> A
juste avant A -> *23
adresse
0xxxh
0xxxh - 1
0xxxh - 2
0xxxh - 3
0xxxh - 4
0x13
0x56
0x17
0x25
0x58
SP
PC
0x50
0x51
0x56
0x70
0x58
A
0x17
0xAA
0xDF
B
0x25
0x35

Q15. 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).
Vous penserez spécifiquement :

— Aux signaux de sorties de l'automate de contrôle (pensez à faire le lien avec la question suivante)
— À la gestion de toutes les instructions impliquant la pile : `pop`, `push`, `call` et `ret`



Q16. Dans la figure suivante vous retrouvez l'automate de la micromachine.

Modifiez l'automate pour rendre possible les instructions `push` et `pop` (dans cette question, on ne vous demande **PAS** de traiter `call` et `ret`). 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.

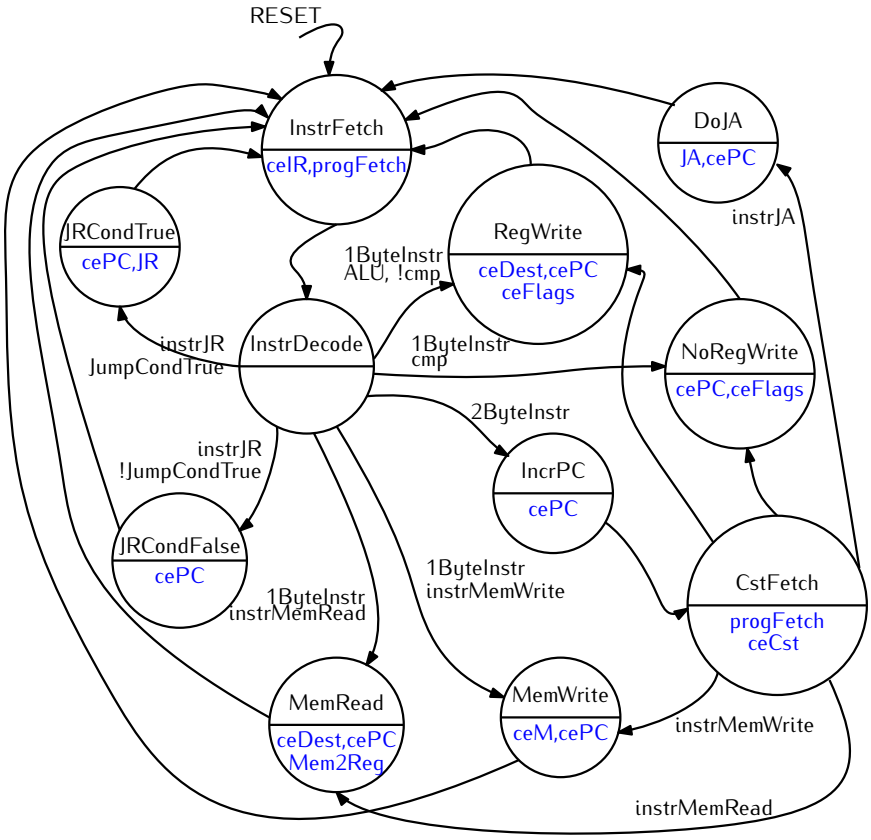
Remarque : nous vous rappelons les opérations effectuées par l'instruction **push A** :

```
<push A>
  SP-1 -> SP
  A    -> @SP                    // @SP est équivalent à Mémoire[SP]
```

... ainsi que par l'instruction **pop A** :

```
<pop A>
  @SP  -> A                    // @SP est équivalent à Mémoire[SP]
  SP+1 -> SP
```

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



Q17. [BONUS] En vous basant sur ce jeu d'instruction "étendu" (ie l'ISA vu en TD auquel on a ajouté `pop`, `push`, `call` et `ret`), proposez une implémentation de la fonction **fibo**.
On rappelle que :

- $fibo(0) = 0$
- $fibo(1) = 1$
- $fibo(n) = fibo(n-1) + fibo(n-2) \forall n \geq 2$

```
*120 -> A ; get n from memory

;; pass A as parameter to fibo
push A
call fibo
pop A ; in the end fibo(n) is in A
```

```
A -> *122 ; we write it back to memory

fibo:
;; (SP+1) -> A ; n is in A
;; un hack pour remplacer *(SP+1)-> A
pop B ; temporarily save return address to B
pop A ; get parameter value from stack to A
push B ; move B (ie return address) back to stack

JR zero IFZ ; n is 0
A-1?
JR un IFZ ; n is 1
A-1 -> A ; n-1 is in A
A-1 -> B ; n-2 is in B
B -> *124 ; une case auxiliaire pour n-2

;; call fibo(n-1)
push A
call fibo
pop A
A -> *125 ; une autre case auxiliaire pour le res de fibo(n-1)

;; call fibo(n-2)
*124 -> A ; get n-2 back from memory
push A
call fibo
pop A


*125 -> B ; now A contains fibo(n-2)
; and B contains fibo(n-1)
A+B -> A ; compute sum fibo(n-2)+fibo(n-1)
push A ; push it back to stack for higher-level rec. call
JA endfibo

zero:
push 0
JA endfibo

un:
push 1
JA endfibo

endfibo:
ret
```