

**NOM, Prénom :**

## IF-3-S1-EC-AO Architecture des Ordinateurs

**10/06/2025**

### Consignes

- Durée : 1h30 (une heure trente minutes)
- Tiers-temps : 2h (deux heures)
- Ce devoir comporte **17 (dix-sept)** questions sur **12 pages**, documentations incluses)
- Seul document : 1 feuille recto-verso manuscrite
- Répondez sur le sujet
- REMPLISSEZ VOTRE NOM TOUT DE SUITE
- Crayon à papier accepté, de préférences aux ratures et surcharges.
- Les cadres donnent une idée de la taille des réponses attendues.

## 1 Questions de cours - Généralités

**Q1.** Expliquez la différence entre l'encodage 1-hot et l'encodage logarithmique des états d'un automate. Vous pourrez prendre l'exemple d'un automate à  $n$  états.

**Q2.** Rappelez lequel de ces deux encodages a été choisi dans l'implémentation de la micromachine. Citez un avantage et un inconvénient de ce choix.

**Q3.** Rappelez la définition de **mot** (en anglais **word** dans une architecture). Vous pouvez donner un exemple.

**Q4.** Expliquez en quelques lignes la différence entre instruction machine et instruction assembleur.

**Q5.** Soit un programme exécutable préparé pour s'exécuter sur un processeur msp430. Donnez une raison vue en cours en en TP pour laquelle ce programme binaire ne peut pas être exécuté sur un processeur intel IA32.

**Q6.** Le msp430 est une architecture ... (entourez la bonne réponse)

- a Register-memory
- b Load-Store

Expliquez votre réponse :

**Q7.** Dans la micro-machine, est-il possible d'utiliser l'instruction `JR -20 IFN` ? Pourquoi ?

## 2 MSP430

**Q8.** Sachant que le msp430 utilise un adressage 16-bits, quel est la taille maximale de son espace d'adressage (en octets) ?

Le modèle que nous avons utilisé propose une mémoire vive de 8Ko. Expliquer la différence entre ces deux tailles.

**Q9.** 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 :

- (à gauche) l'adresse de chaque instruction
- sous les colonnes **Registers** et **Memory content** 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**. Les instructions considérées tiennent sur 16 bits, sauf celles manipulant une constante. Dans ce cas la constante est stockée (comme dans la micromachine) à la suite de l'instruction qui occupe ainsi 32 bits en mémoire. Le pointeur de pile pointe sur la dernière case occupée et la pile croît vers les adresses faibles.

Toutes les valeurs en hexadécimal commencent par 0x, vous prendrez soin d'utiliser la même notation.

Address	Instruction	Registers					Memory content							
		PC	SP	R3	R5	R6	0x2c	0x2d	0x2e	0x2f	0x30	0x31	0x32	0x33
	(initial) —→	0x220c	0x34	0x1234	0x5ef2	0x4002	0xde	0x15	0x10	0xee	0xc3	0xa8	0x12	0xd7
0x220c	MOV R3, R6													
...	PUSH R5													
...	PUSH R6													
...	CALL 0x3804													
...	ADD R6, R5													

**Q10.** Un MSP430 exécute le code ci-dessous à gauche. A droite, vous trouverez le contenu d'un extrait de la mémoire **avant** l'exécution du programme.

Rappel : vous retrouvez en pages 10 et 11 les informations indispensables concernant le jeu d'instruction msp430 pour comprendre ce programme.

```
.section .init9

main:
    mov #0, r9
    mov &0x0b00, r10
    mov #0x0b02, r11

loop:
    cmp r9,r10
    jeq finish
    mov r9,r12
    and #0x1,r12
    jnz    pass
    mov #0, @r11

pass:
    incd r11
    inc r9
    jmp loop

finish:
    jmp finish
```

0x0b00 0a
0x0b01 00
0x0b02 0a
0x0b03 00
0x0b04 09
0x0b05 00
0x0b06 08
0x0b07 00
0x0b08 07
0x0b09 00
0x0b0a 06
0x0b0b 00

0x0b0c 05
0x0b0d 00
0x0b0e 04
0x0b0f 00
0x0b10 03
0x0b11 00
0x0b12 02
0x0b13 00
0x0b14 01
0x0b15 00

Donnez dans le tableau suivant les différentes valeurs de R9, R10, R11 et R12 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.

R9										
R10										
R11										
R12										

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

R9 :  R10 :  R11 :  R12 :

Expliquez l'objectif du programme. Soyez précis :

### 3 Passage de paramètres

On considère le programme C à gauche qui appelle une fonction `sum4` qui réalise la somme de ses 4 paramètres. On trouvera à droite sa traduction en assembleur msp430.

```

1 int sum4(int a, int b, int c, int d){
2     return a+b+c+d;
3 }
4
5
6 int main(){
7     int x1, x2, x3, x4;
8     int y;
9     x1 = 11;
10    x2 = 12;
11    x3 = 13;
12    x4 = 14;
13    y = sum4(x1,x2,x3,x4);
14
15 }
```

```

1 sum4:
2     SUB.W #8, R1
3     MOV.W R12, 6(R1)
4     MOV.W R13, 4(R1)
5     MOV.W R14, 2(R1)
6     MOV.W R15, @R1
7     MOV.W 6(R1), R12
8     ADD.W 4(R1), R12
9     ADD.W 2(R1), R12
10    ADD.W @R1, R12
11    ADD.W #8, R1
12    RET
13 main:
14     SUB.W #10, R1
15     MOV.W #11, 8(R1)
16     MOV.W #12, 6(R1)
17     MOV.W #13, 4(R1)
18     MOV.W #14, 2(R1)
19     MOV.W 2(R1), R15
20     MOV.W 4(R1), R14
21     MOV.W 6(R1), R13
22     MOV.W 8(R1), R12
23     CALL #sum4
24     MOV.W R12, @R1
25     MOV.B #0, R12
26     ADD.W #10, R1
27     RET
```

**Q11.** À votre avis, comment les quatre valeurs de `x1`, `x2`, `x3` et `x4` sont-elles passées de `main` à la fonction `sum4` ?

**Q12.** À quoi correspondent les deux instructions `sub.w #8, R1` et `add.w #8, R1`, respectivement lignes 2 et 11 du code assembleur (à droite).

**Q13.** Dessinez la pile au moment où le processeur s'apprête à exécuter l'instruction de la ligne 7 du code assembleur (à droite).

## 4 Masquage des interruptions

On considère la micromachine comme elle a été conçue en TP, **avec** l'extension gérant les interruptions. Dans cette implémentation, lorsqu'une interruption a été levée, le processeur charge dans PC l'adresse A0. A cette adresse est rangée un ensemble d'instructions qui réalise le traitant d'interruption. Les détails nécessaires au bon déroulement des questions sont rappelés plus loin.

On propose dans cette partie d'étendre cette micromachine afin de permettre le masquage des interruptions. Ce masquage a deux buts :

- Premièrement, il doit empêcher la prise en compte d'une interruption dès lors qu'une première interruption est déjà en cours de traitement. Autrement dit (sur l'automate de la figure donnée plus loin), si on a déjà franchi l'état **JumpToISR** une fois mais qu'on n'a pas encore traité d'instruction `reti`, les occurrences de IRQ sont ignorées.
- Deuxièmement, le programmeur peut explicitement demander de masquer les interruptions avec une instruction `maskIT`. Les interruptions sont alors ignorées jusqu'à l'exécution de l'instruction `unmaskIT`.

Ce mécanisme inclut deux parties au fonctionnement distinct pour le programmeur :

- D'un coté, il peut utiliser les instructions `maskIT` et `unmaskIT` pour demander au processeur d'ignorer (resp. prendre en compte) les interruptions. On supposera qu'au démarrage les interruptions sont prises en compte.
- D'un autre coté, les interruptions sont automatiquement masquées par le processeur dès lorsqu'une première interruption a été traitée, et jusqu'à ce que l'instruction `reti` ait été exécutée.

On vous demande dans la suite d'implémenter ce mécanisme en vous basant sur les figures données plus loin et qui donnent le chemin de données et l'automate de contrôle incluant le traitement des interruptions tel qu'étudié en TD.

**Q14.** Proposez un encodage binaire pour les deux instruction `maskIT` et `unmaskIT`.

**Q15.** Complétez le chemin de données ainsi que l'automate de contrôle des pages suivantes pour permettre l'exécution des deux instructions `maskIT` et `unmaskIT`.

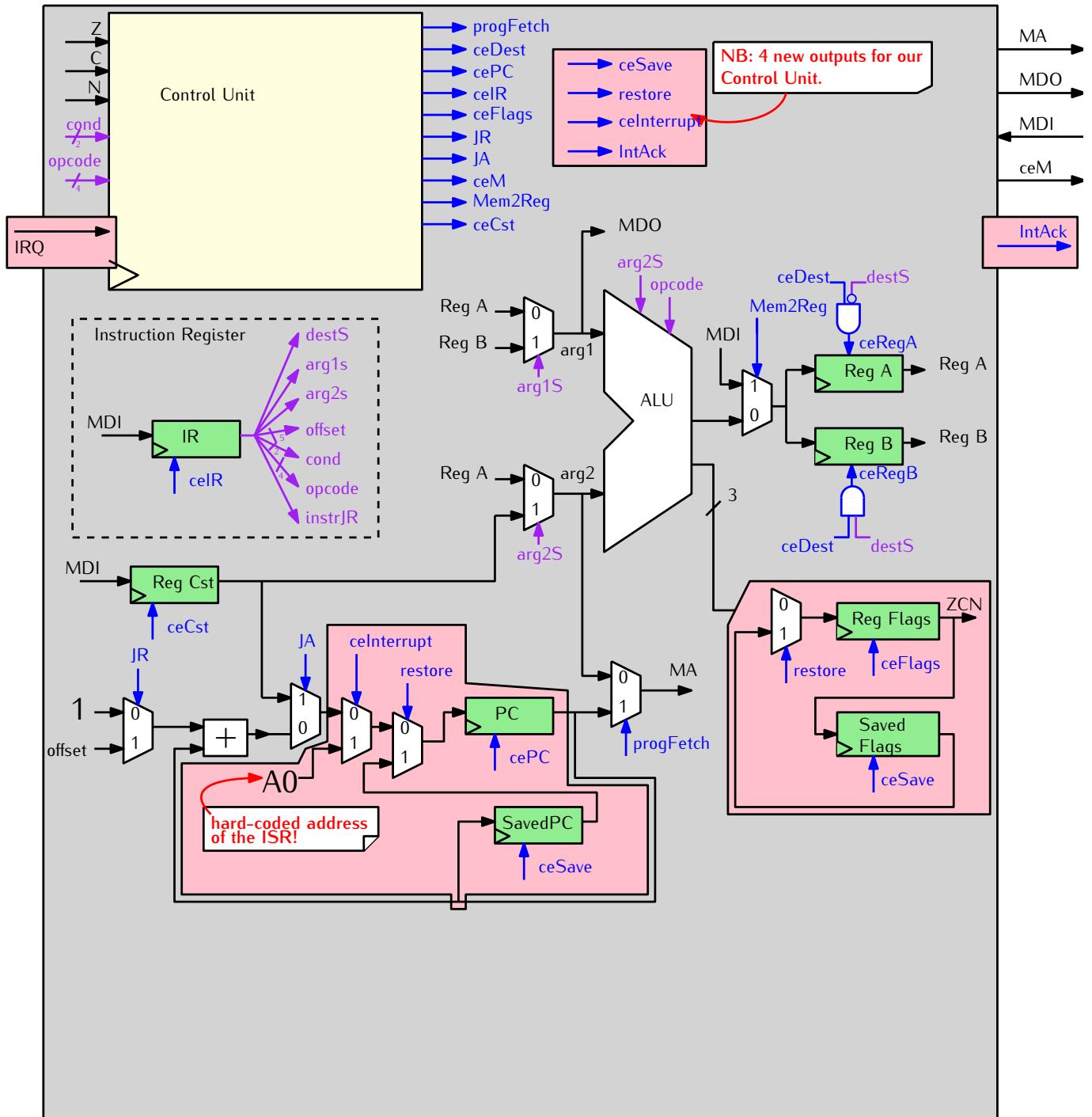
**Répondez sur les schémas des pages suivantes**

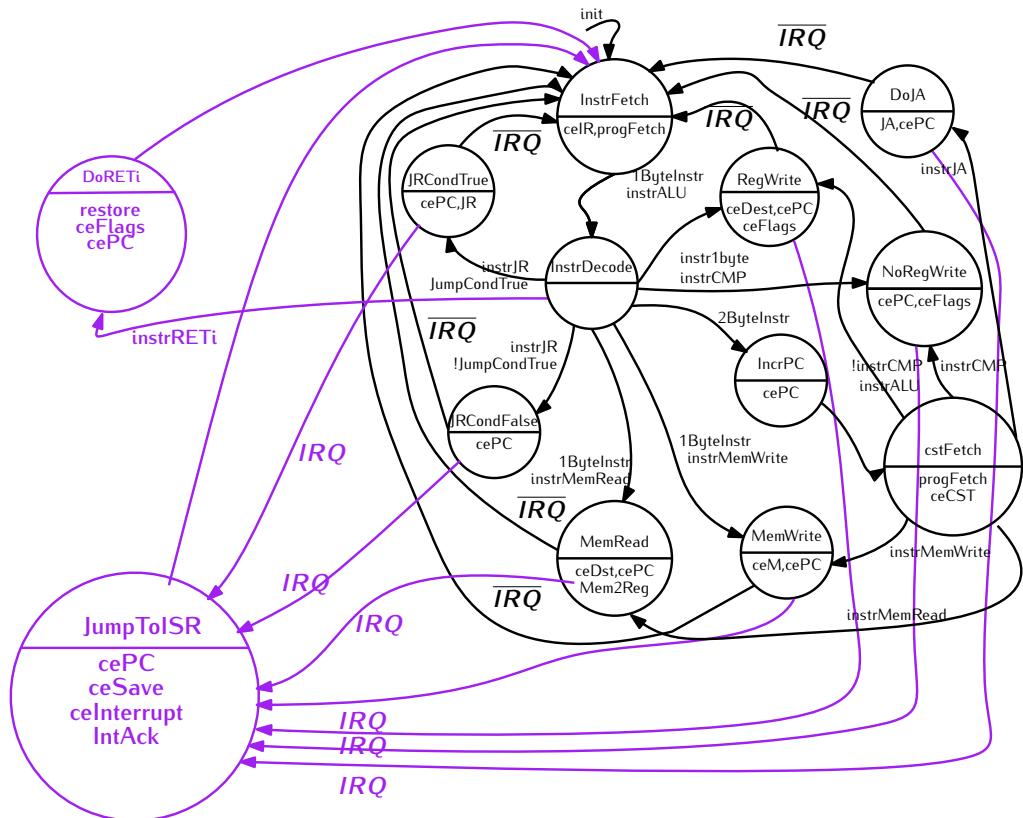
**Q16.** Complétez le chemin de données ainsi que l'automate de contrôle pour prendre en compte le masquage et démasquage des interruptions par l'entrée dans l'exécution d'une ISR et l'exécution de l'instruction `reti`.

**Répondez sur les schémas des pages suivantes**

**Q17.** Si ce n'est pas déjà fait, pensez à faire en sorte que les interruptions soient ignorées dans les bonnes conditions.

**Répondez sur les schémas des pages suivantes**





### Jeu d'instruction de la micromachine

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

$B \rightarrow A$	$21 \rightarrow B$	$B + A \rightarrow A$	$B \text{ xor } -42 \rightarrow A$
$\text{not } B \rightarrow A$	$\text{LSR } A \rightarrow A$	$A \text{ xor } 12 \rightarrow A$	$B - A \rightarrow A$

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  $A+A \rightarrow A$ .
- 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 :

$*A \rightarrow A$	$*A \rightarrow B$	$A \rightarrow *A$	$B \rightarrow *A$
$*\text{cst} \rightarrow A$	$*\text{cst} \rightarrow B$	$A \rightarrow *\text{cst}$	$B \rightarrow *\text{cst}$

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

$\text{JR offset}$	$\text{JR offset IFZ}$	$\text{JR offset IFC}$	$\text{JR offset IFN}$
	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 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.

## Liste compacte des instructions MSP430

Mnemonic	Description	Operation	V	N	Z	C
ADC(.B)	dst	Add C to destination	dst + C $\rightarrow$ dst	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst $\rightarrow$ dst	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C $\rightarrow$ dst	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst $\rightarrow$ dst	0	*	*
BIC(.B)	src,dst	Clear bits in destination	.not/src .and. dst $\rightarrow$ dst	-	-	-
BIS(.B)	src,dst	Set bits in destination	src .or. dst $\rightarrow$ dst	-	-	-
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*
BR	dst	Branch to destination	dst $\rightarrow$ PC	-	-	-
CALL	dst	Call destination	PC+2 $\rightarrow$ stack, dst $\rightarrow$ PC	-	-	-
CLR(.B)	dst	Clear destination	0 $\rightarrow$ dst	-	-	-
CLRC		Clear C	0 $\rightarrow$ C	-	-	0
CLRN		Clear N	0 $\rightarrow$ N	-	0	-
CLRZ		Clear Z	0 $\rightarrow$ Z	-	-	0
CMP(.B)	src,dst	Compare source and destination	dst - src	*	*	*
DADC(.B)	dst	Add C decimaly to destination	dst + C $\rightarrow$ dst (decimaly)	*	*	*
DADD(.B)	src,dst	Add source and C decimaly to dst.	src + dst + C $\rightarrow$ dst (decimaly)	*	*	*
DEC(.B)	dst	Decrement destination	dst - 1 $\rightarrow$ dst	*	*	*
DECD(.B)	dst	Double-decrement destination	dst - 2 $\rightarrow$ dst	*	*	*
DINT		Disable interrupts	0 $\rightarrow$ GIE	-	-	-
EINT		Enable interrupts	1 $\rightarrow$ GIE	-	-	-
INC(.B)	dst	Increment destination	dst +1 $\rightarrow$ dst	*	*	*
INCD(.B)	dst	Double-increment destination	dst+2 $\rightarrow$ dst	*	*	*
INV(.B)	dst	Invert destination	.not.dst $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ dst	-	-	-
NOP		No operation		-	-	-
POP(.B)	dst	Pop item from stack to destination	@SP $\rightarrow$ dst, SP+2 $\rightarrow$ SP	-	-	-
PUSH(.B)	src	Push source onto stack	SP - 2 $\rightarrow$ SP, src $\rightarrow$ @SP	-	-	-
RET		Return from subroutine	@SP $\rightarrow$ PC, SP + 2 $\rightarrow$ 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 $\rightarrow$ dst	*	*	*
SETC		Set C	1 $\rightarrow$ C	-	-	1
SETN		Set N	1 $\rightarrow$ N	-	1	-
SETZ		Set Z	1 $\rightarrow$ C	-	1	-
SUB(.B)	src,dst	Subtract source from destination	dst + .not/src + 1 $\rightarrow$ dst	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not/src + C $\rightarrow$ 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 $\rightarrow$ dst	*	*	*

### Instructions de saut du MSP430

Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if (N .XOR. V) = 0
JL	Label	Jump to label if (N .XOR. V) = 1
JMP	Label	Jump to label unconditionally

### Modes d'adressage du MSP430

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/-	Indirect register mode	@ Rn	Rn is used as a pointer to the operand.
11/-	Indirect autoincrement	@ Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/-	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

### Puissances de 2

$2^0 = 1$	$2^{16} = 65\ 536$	$2^{32} = 4\ 294\ 967\ 296$	$2^{48} = 281\ 474\ 976\ 710\ 656$
$2^1 = 2$	$2^{17} = 131\ 072$	$2^{33} = 8\ 589\ 934\ 592$	$2^{49} = 562\ 949\ 953\ 421\ 312$
$2^2 = 4$	$2^{18} = 262\ 144$	$2^{34} = 17\ 179\ 869\ 184$	$2^{50} = 1\ 125\ 899\ 906\ 842\ 624$
$2^3 = 8$	$2^{19} = 524\ 288$	$2^{35} = 34\ 359\ 738\ 368$	$2^{51} = 2\ 251\ 799\ 813\ 685\ 248$
$2^4 = 16$	$2^{20} = 1\ 048\ 576$	$2^{36} = 68\ 719\ 476\ 736$	$2^{52} = 4\ 503\ 599\ 627\ 370\ 496$
$2^5 = 32$	$2^{21} = 2\ 097\ 152$	$2^{37} = 137\ 438\ 953\ 472$	$2^{53} = 9\ 007\ 199\ 254\ 740\ 992$
$2^6 = 64$	$2^{22} = 4\ 194\ 304$	$2^{38} = 274\ 877\ 906\ 944$	$2^{54} = 18\ 014\ 398\ 509\ 481\ 984$
$2^7 = 128$	$2^{23} = 8\ 388\ 608$	$2^{39} = 549\ 755\ 813\ 888$	$2^{55} = 36\ 028\ 797\ 018\ 963\ 968$
$2^8 = 256$	$2^{24} = 16\ 777\ 216$	$2^{40} = 1\ 099\ 511\ 627\ 776$	$2^{56} = 72\ 057\ 594\ 037\ 927\ 936$
$2^9 = 512$	$2^{25} = 33\ 554\ 432$	$2^{41} = 2\ 199\ 023\ 255\ 552$	$2^{57} = 144\ 115\ 188\ 075\ 855\ 488$
$2^{10} = 1024$	$2^{26} = 67\ 108\ 864$	$2^{42} = 4\ 398\ 046\ 511\ 104$	$2^{58} = 288\ 230\ 376\ 151\ 711\ 744$
$2^{11} = 2048$	$2^{27} = 134\ 217\ 728$	$2^{43} = 8\ 796\ 093\ 022\ 208$	$2^{59} = 576\ 460\ 752\ 303\ 423\ 488$
$2^{12} = 4\ 096$	$2^{28} = 268\ 435\ 456$	$2^{44} = 17\ 592\ 186\ 044\ 416$	$2^{60} = 1\ 152\ 921\ 504\ 606\ 846\ 976$
$2^{13} = 8\ 192$	$2^{29} = 536\ 870\ 912$	$2^{45} = 35\ 184\ 372\ 088\ 832$	$2^{61} = 2\ 305\ 843\ 009\ 213\ 693\ 952$
$2^{14} = 16\ 384$	$2^{30} = 1\ 073\ 741\ 824$	$2^{46} = 70\ 368\ 744\ 177\ 664$	$2^{62} = 4\ 611\ 686\ 018\ 427\ 387\ 904$
$2^{15} = 32\ 768$	$2^{31} = 2\ 147\ 483\ 648$	$2^{47} = 140\ 737\ 488\ 355\ 328$	$2^{63} = 9\ 223\ 372\ 036\ 854\ 775\ 808$
			$2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616$

### Encodage du jeu d'instruction de la micro-machine

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
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

