

NOM, Prénom :

DS Architecture des Ordinateurs

30/01/2024

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.

Ce devoir comporte 13 (treize) questions pour un total de 30 points. Barème donné à titre indicatif.

1 Questions de cours - Généralités

Q1. [/2 pts] Décrivez en une ou deux phrases chacune des étapes du cycle de Von Neumann. Vous décrierez sommairement les interactions entre automate de contrôle (control path) et chemin de donnée (data path) :

Fetch —

Decode —

Execute —

Q2. [/1 pt] Citez les éléments qui permettent de qualifier qu'une machine comporte un processeur 16-bits. Sur le msp430, quelles exceptions connaissez vous à cette règle ?

Q3. [/1 pt] Expliquez en deux phrases la différence entre une machine “load-store” et une machine “register-memory”.

Q4. [/2 pts] Les modes d’adressage sont les différentes façons pour une instruction de demander à accéder à une donnée en mémoire. Expliquez deux modes d’adressage différents.

Q5. [/1 pt] On considère un processeur qui adresse sa mémoire par octet. Il possède une pile dont l’accès en mémoire est permis grâce à un registre spécifique (SP) et à des instructions POP et PUSH. L’instruction POP permet dépiler une valeur de la pile. Lorsqu’on fait POP R8 on observe les changements suivants.

	Registres (16 bits)					Cases mémoires (octets)					
	PC	SP	R6	R7	R8	ccd8	ccd9	ccda	ccdb	ccdc	ccde
avant POP R8	0xa2be	0xccda	0x00d6	0x4ed2	0x0000	0xee	0xd2	0xff	0x54	0x2e	0x63
après POP R8	0xa2c0	0xccd8	0x00d6	0x4ed2	0xff54	0xee	0xd2	0xff	0x54	0x2e	0x63

Que pensez vous des assertions suivantes ?

- V F ☐ a Cette machine accède à la mémoire en “little-endian”
- 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 POP R8 s’encode en 2 octets

2 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 et que le pointeur de pile SP pointe toujours sur la dernière case pleine de la pile. On rappelle en page 6, le jeu d’instruction du msp430. Vous trouverez également le détail des instructions de branchements en page 7.

Q6. [/1 pt] On fait exécuter au processeur les 3 instructions suivantes :

```
.section .init9
main:
    mov #0x8000, R4
    mov #0x0001, R5
    sub R5, R4
```

A l’issue de la dernière instruction, donnez les valeurs des 4 drapeaux Z, N, C et V.

Z : N : C : V :

Quelle est la valeur stockée dans R4 au final :

R4 :

Q7. [/2 pts] On vous donne ci-dessous les informations sur le plan mémoire d'un micro-contrôleur de la famille MSP430 (différent de celui manipulé en TP).

MSP430F5510, MSP430F5509, MSP430F5508
MSP430F5507, MSP430F5506, MSP430F5505, **MSP430F5504**
MSP430F5503, MSP430F5502, MSP430F5501, MSP430F5500



SLAS645L – JULY 2009 – REVISED MAY 2020

www.ti.com

6.4 Memory Organization

Table 6-2 summarizes the memory map of all device variants.

Table 6-2. Memory Organization⁽¹⁾

		MSP430F5504 MSP430F5500	MSP430F5508 MSP430F5505 MSP430F5501	MSP430F5509 MSP430F5506 MSP430F5502	MSP430F5510 MSP430F5507 MSP430F5503
Memory (flash) Main: interrupt vector Main: code memory	Total Size	8KB 00FFFFh–00FF80h 00FFFFh–00E000h	16KB 00FFFFh–00FF80h 00FFFFh–00C000h	24KB 00FFFFh–00FF80h 00FFFFh–00A000h	32KB 00FFFFh–00FF80h 00FFFFh–008000h
RAM	Sector 1	2KB 0033FFh–002C00h	2KB 0033FFh–002C00h	2KB 0033FFh–002C00h	2KB 0033FFh–002C00h
	Sector 0	2KB 002BFFh–002400h	2KB 002BFFh–002400h	2KB 002BFFh–002400h	2KB 002BFFh–002400h
USB RAM ⁽²⁾		2KB 0023FFh–001C00h	2KB 0023FFh–001C00h	2KB 0023FFh–001C00h	2KB 0023FFh–001C00h
Information memory (flash)	Info A	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h
Bootloader (BSL) memory (flash)	BSL 3	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h
Peripherals	Size	4KB 000FFFh–0h	4KB 000FFFh–0h	4KB 000FFFh–0h	4KB 000FFFh–0h

(1) N/A = Not available

(2) USB RAM can be used as general purpose RAM when not used for USB operation.

Quelle est la place maximale *réellement* disponible pour stocker du code (en octets) ?

Justifiez votre réponse

NB : On rappelle en page 7 la table des puissances de 2.

Q8. [/3 pts] 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

- l'adresse des instructions successives (à gauche)
- 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.

Vous noterez les valeurs en hexadécimal.

[illegible]

Q9. [/4 pts] 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 6 et 7 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
    mov #0, r12

loop:
    cmp r9, r10
    jeq finish
    add 0(r11),r12
    add #2, r11
    add #1, r9
    jmp loop

finish:
    mov r12, &0x0b0c
end:
    jmp end
  
```

0x0b00	05
0x0b01	00
0x0b02	02
0x0b03	00
0x0b04	0a
0x0b05	00
0x0b06	02
0x0b07	00
0x0b08	11
0x0b09	00
0x0b0a	01
0x0b0b	00
0x0b0c	02
0x0b0d	0e
0x0b0e	01
0x0b0f	03

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

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

R9 : R10 : R11 : R12 :

Expliquez l'objectif du programme. Soyez précis :

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	–	–
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	*	*	*	*

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

$2^0 = 1$	$2^{16} = 65\,536$	$2^{32} = 4\,294\,967\,296$	$2^{48} = 281\,474\,976\,710\,656$
$2^1 = 2$	$2^{17} = 131\,072$	$2^{33} = 8\,589\,934\,592$	$2^{49} = 562\,949\,953\,421\,312$
$2^2 = 4$	$2^{18} = 262\,144$	$2^{34} = 17\,179\,869\,184$	$2^{50} = 1\,125\,899\,906\,842\,624$
$2^3 = 8$	$2^{19} = 524\,288$	$2^{35} = 34\,359\,738\,368$	$2^{51} = 2\,251\,799\,813\,685\,248$
$2^4 = 16$	$2^{20} = 1\,048\,576$	$2^{36} = 68\,719\,476\,736$	$2^{52} = 4\,503\,599\,627\,370\,496$
$2^5 = 32$	$2^{21} = 2\,097\,152$	$2^{37} = 137\,438\,953\,472$	$2^{53} = 9\,007\,199\,254\,740\,992$
$2^6 = 64$	$2^{22} = 4\,194\,304$	$2^{38} = 274\,877\,906\,944$	$2^{54} = 18\,014\,398\,509\,481\,984$
$2^7 = 128$	$2^{23} = 8\,388\,608$	$2^{39} = 549\,755\,813\,888$	$2^{55} = 36\,028\,797\,018\,963\,968$
$2^8 = 256$	$2^{24} = 16\,777\,216$	$2^{40} = 1\,099\,511\,627\,776$	$2^{56} = 72\,057\,594\,037\,927\,936$
$2^9 = 512$	$2^{25} = 33\,554\,432$	$2^{41} = 2\,199\,023\,255\,552$	$2^{57} = 144\,115\,188\,075\,855\,488$
$2^{10} = 1\,024$	$2^{26} = 67\,108\,864$	$2^{42} = 4\,398\,046\,511\,104$	$2^{58} = 288\,230\,376\,151\,711\,744$
$2^{11} = 2\,048$	$2^{27} = 134\,217\,728$	$2^{43} = 8\,796\,093\,022\,208$	$2^{59} = 576\,460\,752\,303\,423\,488$
$2^{12} = 4\,096$	$2^{28} = 268\,435\,456$	$2^{44} = 17\,592\,186\,044\,416$	$2^{60} = 1\,152\,921\,504\,606\,846\,976$
$2^{13} = 8\,192$	$2^{29} = 536\,870\,912$	$2^{45} = 35\,184\,372\,088\,832$	$2^{61} = 2\,305\,843\,009\,213\,693\,952$
$2^{14} = 16\,384$	$2^{30} = 1\,073\,741\,824$	$2^{46} = 70\,368\,744\,177\,664$	$2^{62} = 4\,611\,686\,018\,427\,387\,904$
$2^{15} = 32\,768$	$2^{31} = 2\,147\,483\,648$	$2^{47} = 140\,737\,488\,355\,328$	$2^{63} = 9\,223\,372\,036\,854\,775\,808$
			$2^{64} = 18\,446\,744\,073\,709\,551\,616$

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 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 suite de Syracuse d'un nombre stocké à l'adresse 200^{1 2}. 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  A - A -> B
2
3  *200 -> A
4  B - A?
5  JR -3 IFZ
6  B -> *200
7  *201 -> A
8
9  A -> *255
10 A -> *100
11
12 LSR A -> B
13 JR 6 IFC
14 B -> A
15 A -> *255
16 JA ...
17
18 A -> B
19 B + A -> A
20 B + A -> A
21 A + 1 -> A
22 A -> *255
23 JA ...
24

```

Q10. [/1 pt] Les instructions JA des lignes 16 et 23 ont été intentionnellement laissées incomplètes. Leur destination (la même pour les deux JA) est l'instruction LSR A -> B. Calculez l'adresse cible et compléter l'instruction :

JA

Q11. [/2pts] À l'aide de la description de l'ISA donnée plus haut, donnez ci-dessous l'encodage binaire et hexadécimal des instructions des lignes 3 à 7 :

*200 -> A		
B-A ?		
JR -3 IFZ		
B -> *200		
*201 -> A		

1. Merci à Ewan C., 3IF 2022-23

2. https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

4 Gestion des interruptions et pile

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 la gestion des interruptions en suivant une approche différente de celle abordée en TP. Lors d'une interruption, le processeur va exécuter le traitant d'interruption stockée en mémoire à l'adresse 0xA0. L'**adresse** de l'instruction qui devra être exécutée **après** la fin de l'ISR doit être **empilée**.

Il ne s'agit **pas** d'ajouter des instructions permettant au programmeur d'accéder à la pile, mais bien d'ajouter l'usage de la pile pour stocker les adresses de retour lors d'interruptions "imbriquées".

On souhaite ainsi ajouter les éléments nécessaires au chemin de donnée et à l'automate de contrôle pour que :

- **Lors d'une interruption**, l'adresse de l'instruction qui suit l'instruction en cours doit être empilée en mémoire
- une **nouvelle instruction** `reti` permette de revenir d'une interruption. Elle a pour effet de "restituer" l'adresse stockée sur la pile lors de la dernière IT (ie le processeur continuera alors à exécuter le code présent à cette adresse, comme le `reti` vu en TP).

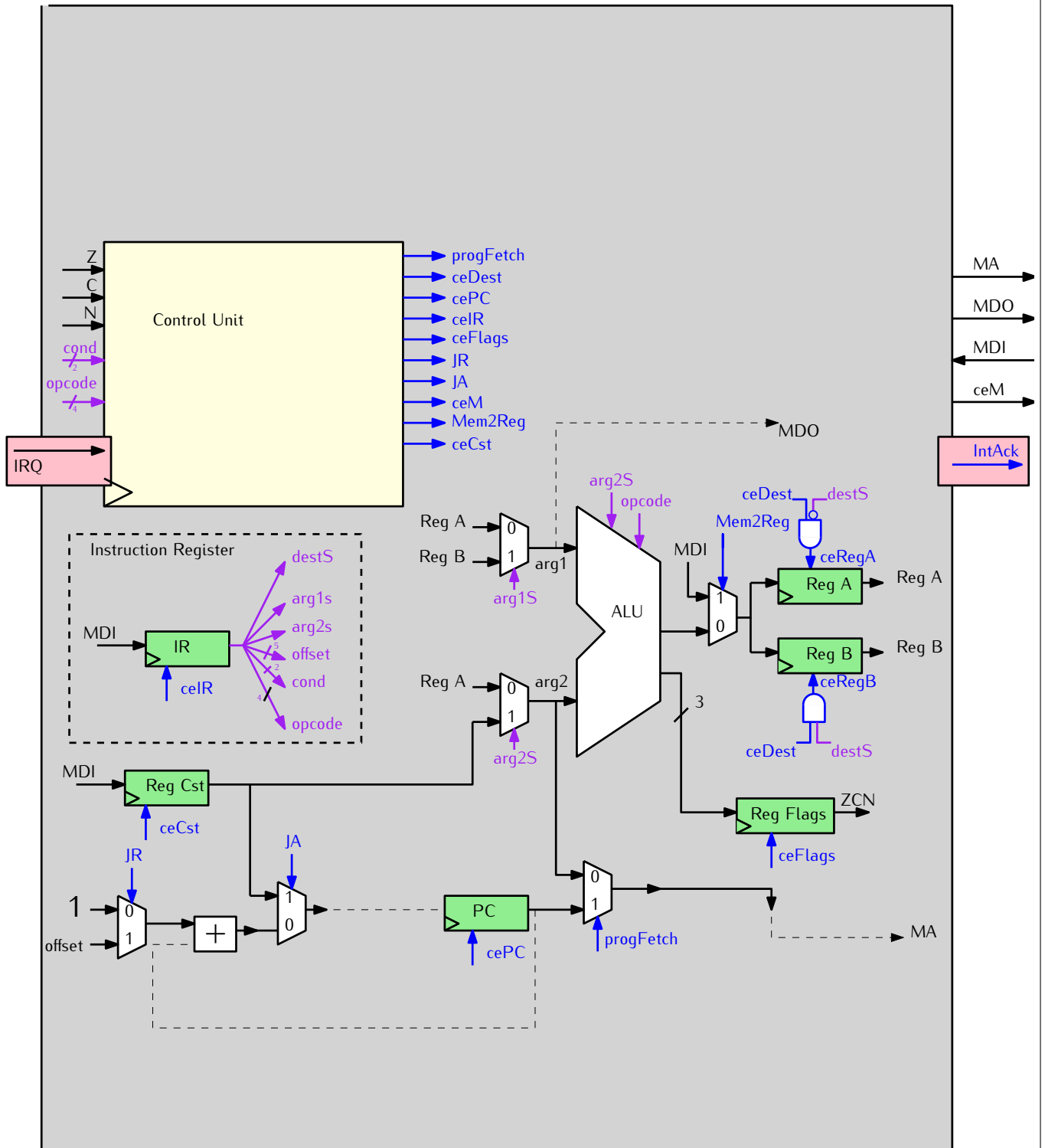
Simplifications volontaires et commentaires complémentaires :

- Le traitant d'interruption exécuté suite à l'occurrence d'une interruption sera toujours le même.
- La gestion des données est entièrement laissée au programmeur qui devra s'assurer que le traitant d'interruption ne "casse rien" de l'état du code (programme principal ou traitant d'interruption) qui est en cours d'exécution lorsque l'interruption survient. NB : **Votre solution ne doit donc PAS** inclure la gestion cet aspect.
- Contrairement au MSP430, la pile permet de stocker des valeurs sur 1 octet. La pile sera "descendante en mémoire", ie pour empiler (respectivement dépiler) on décrémente (respectivement empiler) le pointeur de pile.
- Pour gérer l'état de la pile, la micromachine aura donc un registre SP (*Stack Pointer*) indiquant l'adresse de la dernière valeur empilée.
- Au démarrage de la micromachine (reset), SP est chargé avec la valeur 0xFF. Il revient au programmeur (et donc **pas à vous**) d'éviter dans son programme qu'une instruction `reti` puisse être exécutée avant la moindre interruption.
- En dehors de ces précisions, vous êtes libres d'implémenter votre solution comme bon vous semble.

Q12. [/5 pts] Dans la figure ci-dessous vous trouverez le chemin de données de la micromachine, sans gestion d'interruption. Seuls les signaux IRQ et IntAck ont été positionnés (leur signification est identique à celle vue en cours et en TP). Ajoutez le nécessaire pour permettre l'empilement des adresses d'instructions lors de l'occurrence d'une interruption, ainsi que le traitement de l'instruction `reti`.

Vous penserez spécifiquement :

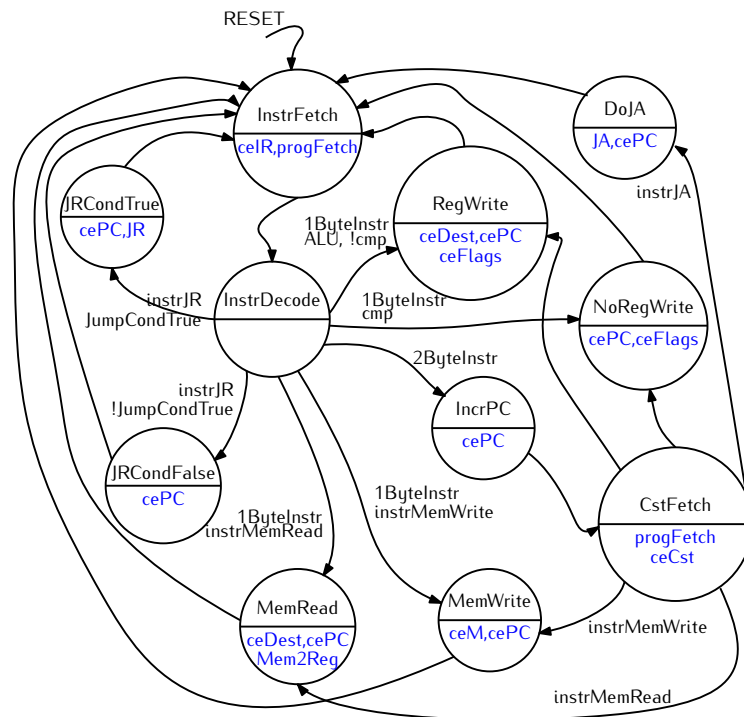
- Aux signaux de sorties de l'automate de contrôle (pensez à faire le lien avec la question suivante).
- À la gestion de toutes les informations impliquant la pile : le pointeur de pile, l'adresse de l'instruction qui suit celle en cours d'exécution lorsque la requête d'interruption apparaît, etc.
- Que les pointillés sont les signaux préexistant à votre modification, autour desquels il faudra travailler, et que vous êtes donc libre de les supprimer/remplacer/modifier.

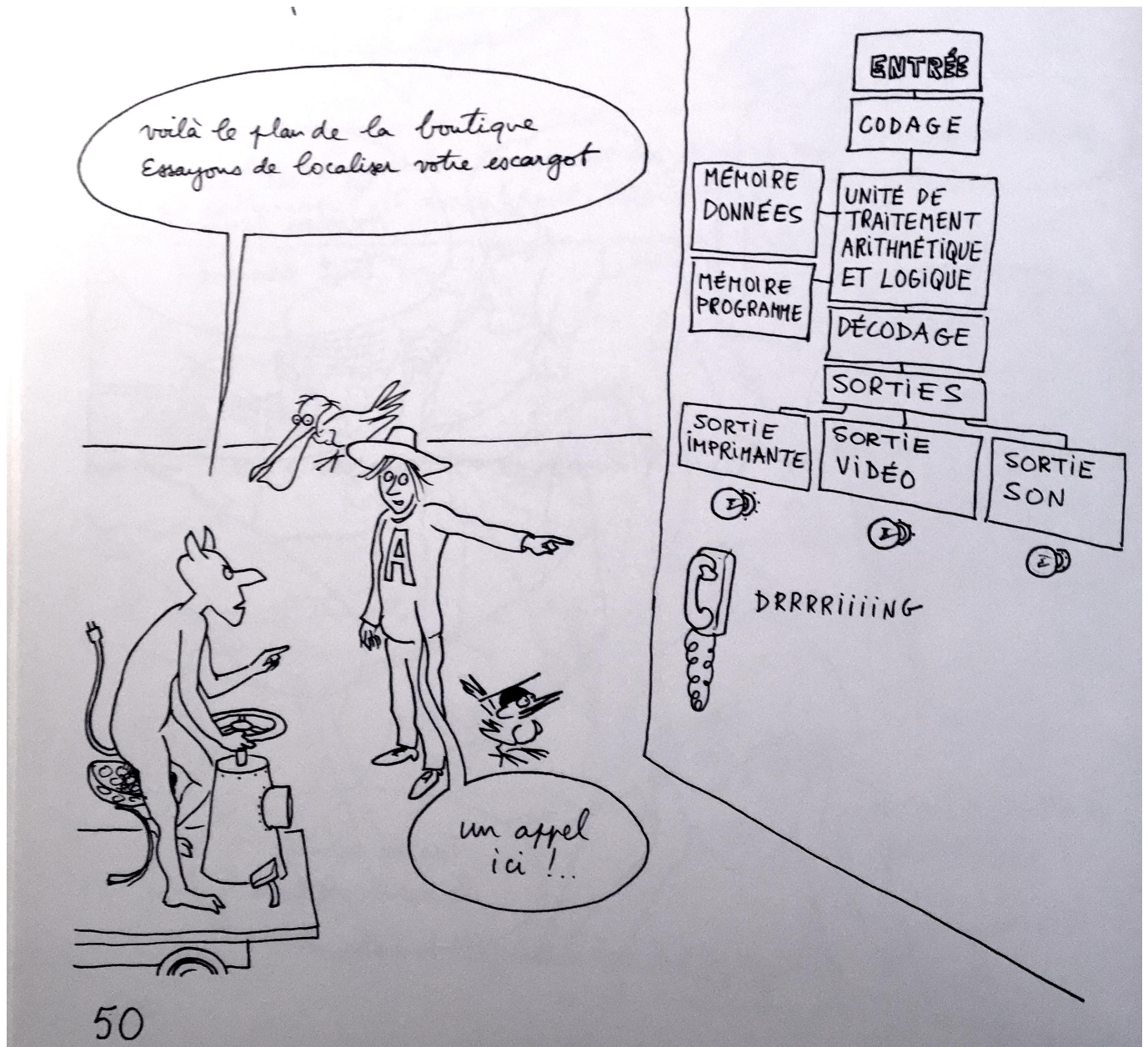


Q13. [/5 pts] Dans la figure suivante vous retrouvez l'automate de la micromachine sans gestion d'interruption.

Modifiez l'automate pour rendre possible la gestion des interruptions telle que décrite plus haut. Il faudra faire apparaître aussi bien le nécessaire pour gérer l'occurrence d'une interruption que la gestion de l'instruction reti.

Ajoutez les états nécessaires et connectez les aux autres en vous assurant que les transitions ajoutées sont telles que l'automate est toujours déterministe. Ajoutez les signaux **de sortie** nécessaires en veillant à réutiliser les noms que vous avez utilisés lors de la question précédente.





*Les Aventures d'Anselme Lanturlu - L'informagique. Jean-Pierre Petit. Éditions Belin, 1980.