Micro contrôleur embarqué : le PicoBlaze

Nous allons aborder dans ce chapitre les architectures programmables. Par programmable nous entendons ici susceptible d'exécuter un programme en langage de bas niveau (assembleur).

I.1 - La représentation des nombres

La donnée de base de PicoBlaze est l'octet. L'octet contient 8 bits. Le bit 0 est le bit de poids faible et le bit 7 est le bit de poids fort.
De plus, le PicoBlaze connaît des mots de 18 bits. Le bit 0 est aussi le bit de poids faible et le bit 17 est le bit de poids fort. Dans la mémoire l'octet de poids fort se trouve devant l'octet de poids faible.
L'unité centrale connaît trois présentations des nombres :
Nombres entiers non signés (Unsigned Integer): Un octet peut avoir des valeurs entre 0 et 255 ($FF), un mot entre 0 et 65535 ($FFFF).
Nombres entiers signés complément à 2 (Two's complement). Un octet peut avoir des valeurs entre -128 ($80) et +127 ($7F), un mot entre -32768 ($8000) et 32767 ($7FFF). Le bit de poids fort indique toujours le signe.
Décimal codé binaire (BCD Binary Coded Decimal): Un octet contient deux chiffres décimaux. Les bits 7,6,5,4 contiennent le chiffre de poids fort, les bits 3,2,1,0 contiennent le chiffre de poids faible. Un octet peut donc avoir des valeurs entre 0 ($00) et 99 ($99). Cette présentation est peu utilisée. Des **adresses** ont 16 bits et adressent des octets. L'espace d'adressage de l'unité centrale comprend donc 2^{16} = 65536 octets ($0000..$FFFF). On trouve toutes les mémoires comme la RAM, la ROM, l'EEPROM, et les registres d'E/S dans l'espace d'adressage.

Exercice 1
Convert the following decimal numbers to binary : +11, -11, -23 (two's complement representation with 8 bits). Then convert these numbers to hexadecimal.
Convert the following binary numbers to decimal : 111011, 11010101.
Perform the following binary substraction using (i) the « ordinary » substraction technique ; (ii) the two's complement method.
11011 - 10101
11100 - 1001

I.2 - Le modèle de programmation du PicoBlaze

Il peut être présenté de la manière suivante :

7	s0	0	7	s1	0
7	s2	0	7	s3	0
7	s4	0	7	s5	0
7	s6	0	7	s7	0
7	s8	0	7	s9	0
7	sA	0	7	sB	0
7	sC	0	7	sD	0
7	sE	0	7	sF	0

9	Pointeur de pile SP	0

9	Compteur ordinal PC	0

Z	I	C

C : retenue (Carry) Z : zéro
I : masque d'interruption

La mémoire programme est organisée en 1024 instructions sur 18 bits. L'adressage se fait donc sur 10 bits (2^{10}=1024).
La mémoire donnée est organisée en 64 octets. L'adressage se fait donc sur 6 bits (2^6=64)
Par comparaison, les micro-processeurs 8 bits avaient un espace d'adressage sur 16 bits. Cet espace mélangeait données et programme. Ici on a une séparation, une telle architecture est appelée Harvard.

I.3 - Quelques instructions

3.1 - Les modes d'adressage

Le PicoBlaze connaît quatre modes d'adressage. Nous n'en aborderons que deux dans cette section.

- **Adressage registre** : c'est l'adressage le plus simple, il ne peut concerner que des registres.

Op-code	***instruction***	***opérande(s)***	***commentaire***
`0 10 10`	`load`	`s0,s1`	`;charger s1 dans s0`

L'opération qui permet de passer de l'ensemble des instructions et opérandes à l'op-code s'appelle l'assemblage. Le programme qui réalise cette opération est un assembleur. L'opération inverse s'appelle désassemblage.

Remarquez la taille de l'op-code : 18 bits. Ces 18 bits sont composés de 5 caractères hexadécimaux de 4 bits. Ils représentent donc 20 bits. Pour se contenter de 18 bits, le caractère le plus à gauche ne peut dépasser la valeur 3.

- **Adressage immédiat (Immediate)** : l'opérande correspondant se trouve directement dans le programme derrière le code de l'instruction et un premier opérande. Il se fait sur 8 bits.

```
0 00 7F    load s0,7F      ;charger 127 dans s0
1 80 10    add s0,10       ;additionner 16
```

On peut remarquer que les nombres (deuxième opérande) sont systématiquement en hexadécimal.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
LOAD	sX	aaaaaaaa	0	0	0	0	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

xxxx est le code de X sur 4 bits.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADD	sX	aaaaaaaa	0	1	0	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADDCY	sX	aaaaaaaa	0	1	1	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

sX <- sX+aaaaaaaa + C (addition avec retenue)

Avant d'aller plus avant dans la description des modes d'adressage, nous allons nous intéresser à la description du fonctionnement d'une architecture programmée.

I.4 - Bien comprendre le modèle de programmation

Il est important de bien comprendre le fonctionnement général d'un micro-contrôleur pour bien se persuader qu'une connaissance parfaite d'un état permet de trouver les états suivants. Pour un micro-contrôleur on appelle état la connaissance des valeurs des registres et de celles des mémoires. La taille mémoire est de 1024 mots de 18 bits. Cela signifie que toute adresse mémoire est spécifiée sur 10 bits. A titre de comparaison, la majorité des micro-contrôleurs 8 bits ont un espace d'adressage sur 16 bits (mais pour des mots de 8 bits).

Exercice 2 (données en hexadécimal)
On donne les états (connaissance partielle : on ne connaît pas tout, mais c'est suffisant) suivants :

	PC=200	***s0=FF***
adresses	mémoire	prog.
1FF	000C6	
	00104	
	12101	
	12103	
	00001	
204		

PC=300
000C6
001FF
12101
1A101
34300
0001B

C=0	***Z=0***

1°) D'après le schéma donné ci-dessus, désassembler la partie utile du code.
2°) Exécuter ensuite les instructions que vous pouvez, et donner les états successifs.

I.5 - Les modes d'adressage (suite)

Nous avons déjà abordé deux modes d'adressage en section I.3.1. Nous en présentons deux nouveaux maintenant.

- **Immédiat 10 bits** : l'adresse est donnée sur 10 bits. L'instruction JUMP utilise ce mode d'adressage parce que l'adresse fournie est une adresse absolue de programme.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
JUMP	aaaaaaaaaa		1	1	0	1	0	0	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **Direct 6 bits** : l'adresse de la mémoire se trouve dans le dernier octet du code de l'instruction (sur les 6 bits de poids faible). Les instructions STORE et FETCH fonctionnent avec ce mode d'adressage et permettent d'adresser 64 octets (seulement) de la mémoire.

```
0601F           FETCH s0,1F     ;charge le contenu de 1F dans s0
18056           ADD s0,56       ;additionne 56 en hexadécimal
34188           JUMP 188        ;saute à l'adresse 188
                                ;(sur 10 bits)
```

A noter les opcodes sur 18 bits (5 caractères hexadécimaux).

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
FETCH	sX	aaaaaa	0	0	0	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
STORE	sX	aaaaaa	1	0	1	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Exercice 3 (données en hexadécimal)

On donne les états (connaissance partielle : on ne connaît pas tout, mais c'est suffisant) suivants :

	PC=1FF
adresses	mémoire
1FF	000A1
	2E03F
	18002
	2E03E
	18004
204	2E03D

PC=3000	***s2=3F***	***s0=FD***	***C=1***	***Z=0***

2FF	100FE
	00211
	2A211
	2E21F
	0621E

1°) D'après le schéma donné ci-dessus, désassembler la partie utile du code à l'aide du tableau d'instructions du TD1 et du TD2.
2°) Exécuter ensuite les instructions que vous pouvez et donner les états successifs.

I.6 - Mon premier programme en assembleur

Nous avons déjà eu l'occasion de présenter quelques champs à la section 3-1. Nous allons présenter d'autres possibilités maintenant. Regardez le programme ci-dessous :

```
CONSTANT m,10 ;commentaire
ADDRESS 300
     LOAD s3,m
     JUMP suite
suite:    LOAD ...
```

On distingue dans ce programme une étiquette , une définition symbolique, un commentaire et la définition de l'origine du programme. L'étiquette est un nom suivi de deux points, ici "suite:". La définition symbolique est m pour la valeur 10 (en hexadécimal). L'origine du programme est en adresse 300 (hexadécimal) ; on utilise la directive ADDRESS.

Exercice 4

On donne :

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
JUMP NZ,	aaaaaaaaaa		1	1	0	1	0	1	0	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Calculer la valeur de l'opcode de branchement XXXX dans le programme donné ci-dessous :

Adresse du code	*Code*		*Instruction*
200	00003		LOAD s0,03
201	1C001	Ici:	SUB s0,01
202	XXXX		JUMP NZ,Ici
203			

I.7 - Les structures de contrôle

On présente maintenant les structures de contrôle dans le cas du PicoBlaze.

7.1 - Structure si-alors-sinon

En écriture en langage C cette structure de contrôle s'écrit :

```
if (s0==s1) {
// partie alors
}
else{
// partie autrement
}
```

En assembleur PicoBlaze cela s'écrit :

```
      compare s0,s1
      jump nz, branche_autrement
      ;code pour la branche alors
      jump if_fini
branche_autrement:
      ;code pour la branche autrement
if_fini:
      ;code suivant le if
```

Pour les autres branchements conditionnels voir en fin de polycopié.

7.2 - Structure répéter ... tant que ... et structure tant que...

On donne la structure correspondante dans plusieurs cas particuliers. Remarquez les difficultés pour le passage de s0>=5 à s0>5.

```
; tant que s0==s1 faire action
tque:  COMPARE s0,s1
       JUMP NZ, suite ; s0-s1<>0 on va a suite
       ;action qui modifie s0 et/ou s1
       JUMP tque
suite:     ....
```

La comparaison suivante est basée sur le fait qu'une comparaison est équivalente à une soustraction et qu'ainsi le drapeau C est positionné si et seulement si le résultat de la soustraction est strictement négatif. Attention ce résultat général n'est pas intuitif.

```
; tant que s0>=5 faire action
tque:  COMPARE s0,5 ; si s0-5<0 alors C=1
       JUMP C, suite ; s0-5<0 on va a suite
       ;action qui modifie s0
       JUMP tque
suite:     ....
```

La comparaison suivante est plus complexe à réaliser car aucune instruction de saut n'est conditionnée par les deux drapeaux Z et C simultanément. Un moyen d'y parvenir est d'inverser la comparaison ce qui nécessite de mettre 5 dans un registre !

```
; tant que s0>5 faire action
      LOAD s1,5 ; obligatoire pour inverser comparaison
tque: COMPARE s1,s0
      JUMP NC, suite ; s0-5<=0 on va a suite
      ;action qui modifie s0
      JUMP tque
suite:     ....
```

7.3 - Structure for ...

En utilisant le plus possible les définitions symboliques on peut réaliser ce type de boucle avec un programme du type :

```
; pour i variant de 0 à 15
;(compris) faire action
constant MAX, 15
namereg s0,i
  LOAD i, MAX
boucle:
  ;code de boucle
  ...
  SUB i,01
  JUMP NZ,boucle
  ;code suivant la boucle
  ...
```

Exercice 5

On suppose qu'en adresse $00 se trouvent des valeurs positives (octets) provenant de l'extérieur du PicoBlaze. Les détails sur la façon dont cela peut marcher ne nous intéressent pas ici. La lecture de cette adresse (on parle de port) se fait avec l'instruction INPUT. Faire un programme qui fait sans arrêt l'acquisition de 4 de ces valeurs, fait la somme et divise par 4 le résultat pour le ranger le tout à l'adresse $04 (écriture d'un port avec l'instruction OUTPUT).
Indication : la division par 4 peut être réalisée par deux divisions par deux à la suite. Une division par deux se réalise par un décalage à droite d'un bit. Mais il vous faut décaler un nombre plus grand que 8 bits (s_msb s_lsb). Cela peut se faire avec les instructions SR0 et SRA.

I.8 - L'adressage indexé

Indexé : l'adresse est calculée par le contenu du registre sX utilisé dans l'adressage. Ce registre est mis entre parenthèses dans ce cas. L'espace mémoire étant limité à 64 octets, seuls les 6 bits de poids faibles sont utilisés dans ce mode d'adressage.

```
     00205      LOAD s2,05       ;boucle 5 fois (s2 compteur de boucle)
0188 07010      FETCH s0,(s1)    ;charge le contenu donné par s1 dans s0
     18300      ADD s3,s0        ;additionne 5F en hexadécimal
     18101      ADD s1,01        ;permet d'aller à l'adresse suivante
                SUB S2,01        ;s2 <- s2-1
     34188      JUMP NZ,188      ;saute à l'adresse 188 (sur 10 bits)
```

I.9 - Les tableaux

Les tableaux sont les structures de données correspondant à l'adressage indexé. Contrairement à ce qui se passe en langage C, il faut gérer la mémoire des tableaux vous-même.

I.10 - Exercices

Exercice 6
Déterminer les valeurs de s0, s1 et cases mémoires suite aux instructions suivantes :

Valeur de s0,s1 ?		***Puis ...***	
LOAD	s0,01	ADD	s0,1
LOAD	s1,10	FETCH	s1,(s0)
STORE	s1,(s0)		

Exercice 7
On suppose qu'en adresse $00 se trouvent des valeurs (octets) provenant de l'extérieur du PicoBlaze. Les détails sur la façon dont cela peut marcher ne nous intéressent pas ici. Faire un programme qui fait l'acquisition de 4 de ces valeurs et les stocke dans un tableau. Ensuite on calculera la somme de ces 4 cases divisée par 4 et le résultat sera rangé à l'adresse $04.

Exercice 8
Modifier ce programme pour qu'il calcule le maximum des 4 valeurs du tableau et le range en mémoire à la suite du tableau.

I.11 - Les sous-programmes le switch et le PicoBlaze

11.1 - Les sous-programmes

Les instructions CALL et RETURN sont destinées à l'appel de sous-programme et son retour. Expliquer le rôle de la pile pour les appels imbriqués. La pile ne comporte que 31 niveaux est n'est manipulable par aucune instruction spécifique. Ceci est inconvénient

pour les langages évolués qui ne peuvent pas réaliser le passage de paramètres en utilisant la pile. Cela veut aussi dire qu'il n'y aura donc pas d'initialisation de la pile en début de programme.

11.2 - La programmation de suivant le cas

La programmation de suivant le cas s'écrit en C :

```
switch (variable)  {  /* variable est de type int ou char */
case valeur_1 : action_1; break;
case valeur_2 : action_2; break;
...
case valeur_n : action_n; break;
default : action_0;
}
```

Remarque : si action_n est un sous programme il faut le remplacer par action_n() en C. Les "break" imposent des conditions exclusives sur la variable. S'ils ne sont pas présents (car non obligatoires) l'algorithme exécuté peut se traduire :
AU CAS OU
condition 1, FAIRE : action_1 et toutes les suivantes
condition 2, FAIRE : action_2 et toutes les suivantes
condition 3, FAIRE : action_3 et toutes les suivantes
condition 4, FAIRE : action_4 et la suivante
DANS LES AUTRES CAS, FAIRE : action_0

```
constant valeur_1, ...
constant valeur_2, ...
...
constant valeur_n, ...

     compare s0, valeur_1
     jump nz case_2
     ...; code pour action_1
     jump case_done
case_2:
     compare s0, valeur_2
     jump nz case_3
     ...; code pour action_2
     jump case_done
case_3:
     ...; etc

case_n:
     compare s0, valeur_n
     jump nz default
     ...; code pour action_n
     jump case_done
default:
     ...; code pour action_0
case_done:
     ...; code suivant le case
```

Remarque : les actions action_x de vrais sous-programmes (appelés avec CALL finissant par RETURN).

11.3 - Comparaison des instructions

Si vous développez autour du PicoBlaze vous serez amené à utiliser plusieurs assembleurs très proches mais pas complètement similaires, celui du KCPSM3 et celui du simulateur PBlazeIDE. A noter que si vous travaillez sous Linux, vous n'aurez pas ce problème. Nous présentons donc maintenant les différents mnémoniques des deux environnements.

KCPSM3	***PBlazeIDE***
addcy	addc
subcy	subc
compare	comp
store sX, (sY)	store sX, sY
fetch sX, (sY)	fetch sX, sY
input sX, (sY)	in sX, sY
input Sx,KK	in sX,$KK
ouput sX, (sY)	out sX, sY
return	ret
returni	reti
enable interrupt	eint
disable interrupt	dint

Il existe d'autres différences avec les directives que nous présentons maintenant.

Fonction	***KCPSM3***	***PBlazeIDE***
positionnement de code	address 3FF	org $3FF
constante	constant MAX, 3F	MAX equ $3F
alias de registre	namereg addr, s2	addr equ s2
alias de ports	constant in_port, 00	in_port dsin $00
	constant out_port 10	out_port dsout $10
	constant bi_port, 0F	bi_port dsio $0F

11.4 - Exercices

Exercice 9

Modifier le premier exemple du « suivant le cas faire ... » pour que ce qui doit être réalisé se trouve dans des sous-programmes.

Exercice 10

Écrire un programme qui lit sans arrêt la mémoire $00 pour la mettre dans s3 et qui suivant la valeur décimale (1, 2, 4, 8, 16, 32, 64, 128) appelle un des sous programme do, re, mi, fa, sol, la, si et do2. On ne vous demande pas de détailler les sous-programmes. Pourquoi ne peut-on pas utiliser directement la méthode du premier exemple du « suivant le cas faire ... » ? On utilisera donc la deuxième technique mais on vous impose d'utiliser des sous-programmes dont le contenu sera NOP (réalisé par un AND s0,s0) suivi de RETURN (c'est le principe qui nous intéresse).

Exercice 11

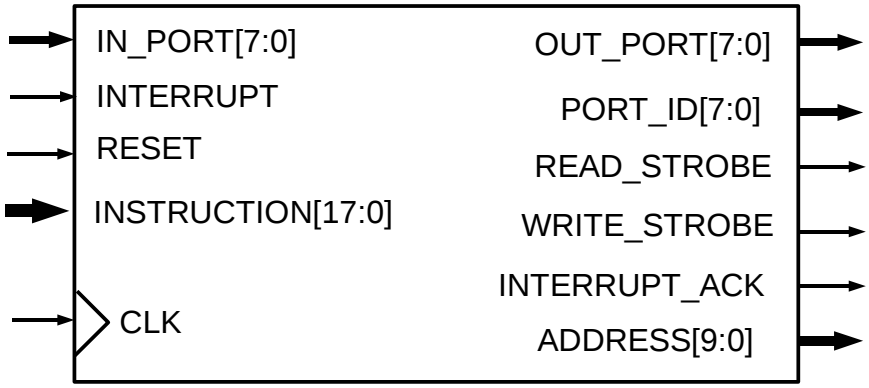
Écrire un programme qui, suivant la valeur présente en adresse $1000 soit fait la somme des cases d'un tableau de 4 cases, soit calcule la moyenne de ces cases, soit cherche le maximum du tableau.

Chapitre II La partie matérielle du PicoBlaze

II.1 - Embarquer le PicoBlaze

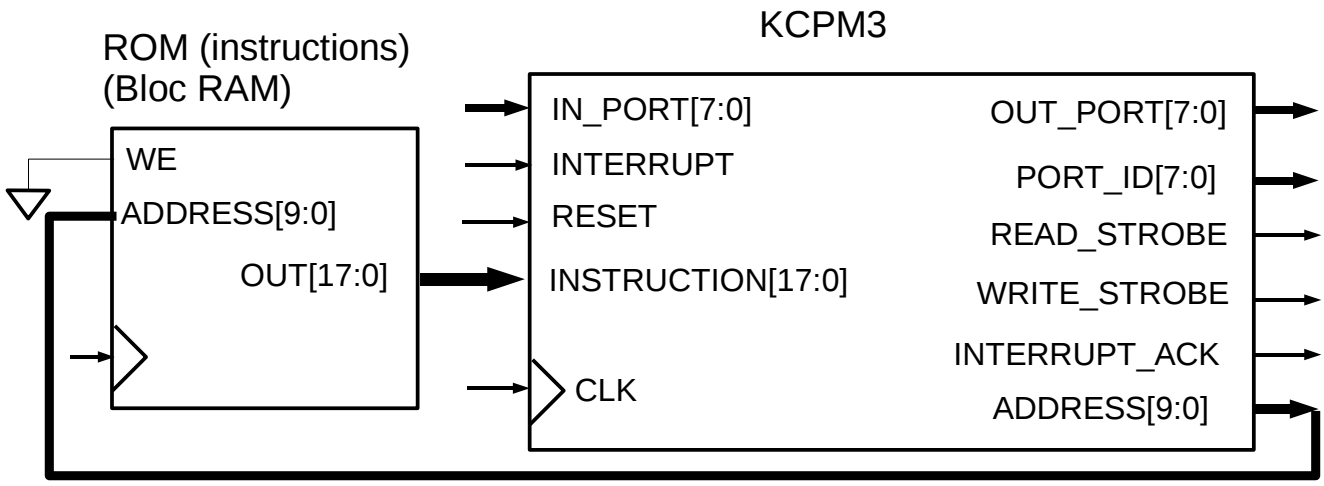
Le PicoBlaze est un micro-contrôleur destiné à être embarqué dans un FPGA. Ce qui caractérise ce type de composant est qu'il comporte en général peu de logique d'interfaçage vers l'extérieur (PORTS). La raison à cela est qu'il va se trouver dans un FPGA où le développement de cette logique externe se fait très facilement. Elle est de plus adaptée à notre propre problème. Cette absence de ports va rendre l'architecture de ce micro-contrôleur facile à comprendre. Commençons par présenter l'aspect fonctionnel du PicoBalze :

Micro contrôleur PicoBlaze



Description fonctionnelle du PicoBlaze

La plus petite application consiste à lui ajouter une mémoire avec les instructions correspondantes dedans. Cela nous donne le schéma ci-dessous où toutes les entrées ainsi que les sorties seront reliées à de la logique du FPGA comme cela sera partiellement détaillé dans la section suivante.



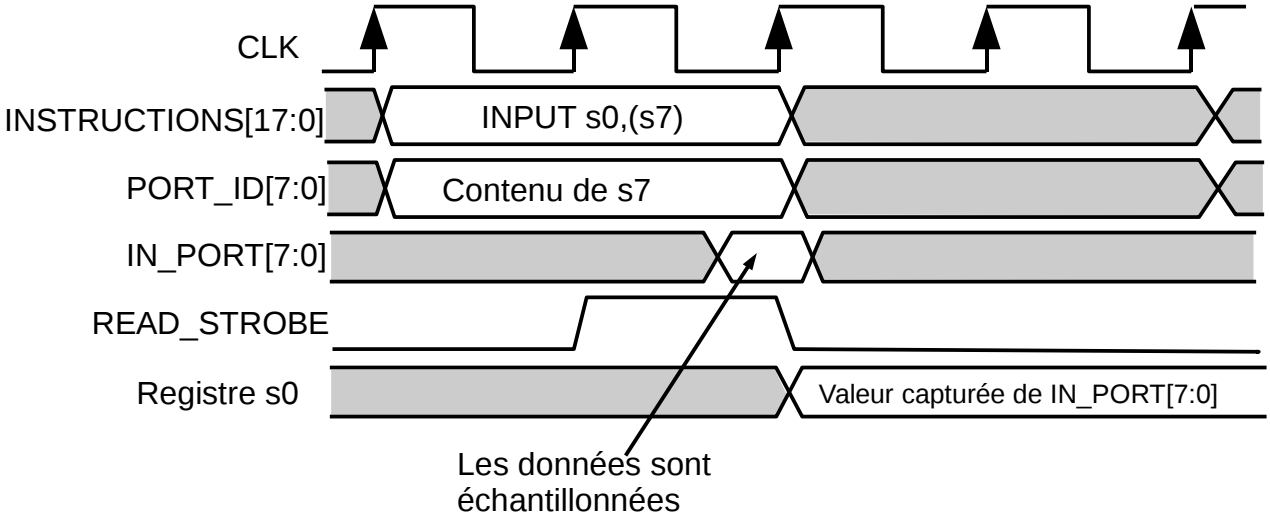
Implantation standard utilisant un bloc RAM 1Kx18 pour stocker les instructions

II.2 - Les entrées sorties

Nous allons apprendre dans ce chapitre à interfacer des ports au PicoBlaze. Commençons par les entrées.

2.1 - Interfacer un PORT d'entrée

Le diagramme temporel correspondant est présenté maintenant.



Timing de l'instruction INPUT

Pour faire l'acquisition d'une valeur externe, il faut réaliser une instruction « INPUT » en précisant un numéro de port sur 8 bits. Il y a donc 256 ports possibles.

On remarque que l'instruction « INPUT » est réalisée en deux fronts d'horloge (comme toutes les instructions) et qu'un signal READ_STROBE est disponible sur un seul front d'horloge. La partie matérielle peut être réalisée comme suit :

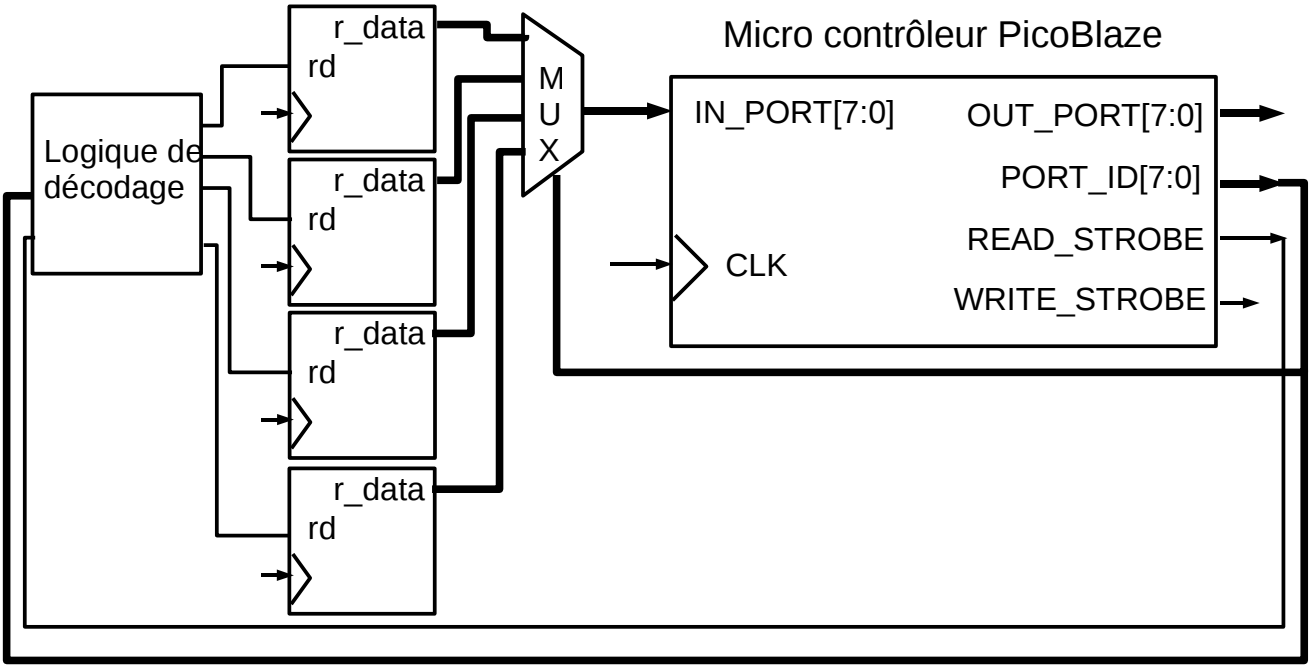


Schéma d'interface entre PORTS d'entrée et PicoBlaze

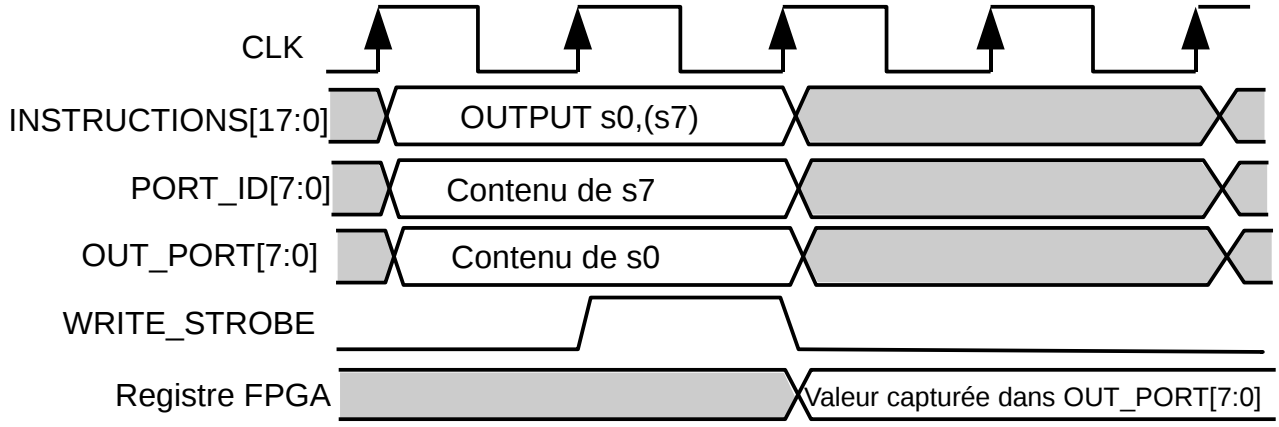
Dans cet exemple on dispose de 4 registres 8 bits pour lesquels on n'a pas dessiné les entrées arrivant de l'extérieur. Le multiplexeur ainsi que la logique de décodage peuvent n'utiliser que les 2 bits de poids faible du signal PORT_ID. Il est possible de rajouter un registre entre le multiplexeur et l'entrée du PicoBlaze pour augmenter les performances (pipeline).
On donne un exemple de programme VHDL correspondant aux deux composants ajoutés, le multiplexeur et la logique de décodage ainsi qu'une partie du registre.

```
-- multiplexeur
with port_id(1 downto 0) select
  data <= indata_0 when "00",
          indata_1 when "01",
          indata_2 when "10",
          indata_3 when "11";

-- circuit décodage
process(read_strobe,port_id) begin
  if read_strobe='0' then
    rv <= "0000";
  else
    case port_id(1 downto 0) is
      when "00" => rv <= "0001";
      when "01" => rv <= "0010";
      when "10" => rv <= "0100";
      when others => rv <= "1000";
    end case;
  end if;
```

```
end process;

-- registre 8 bits
-- A mettre dans une architecture avec l'entité correspondante
-- car on l'utilisera 4 fois
process(clk) begin
  if clk'event and clk='1' then
    if rd = '1' then
      r_data <= d;
    end if;
  end if;
end process;
```

2.2 - Interfacer un PORT de sortie

Le diagramme temporel de l'instruction OUTPUT est donnée ci-dessous.



Timing de l'instruction OUTPUT

On n'a pas représenté les signaux de sorties du circuit de décodage mais ils ont légèrement décalés par rapport à WRITE_STROBE puisqu'ils l'utilisent. Nous montrons ci-dessous un exemple de circuit pouvant réaliser une sortie sur un des quatre ports présents :

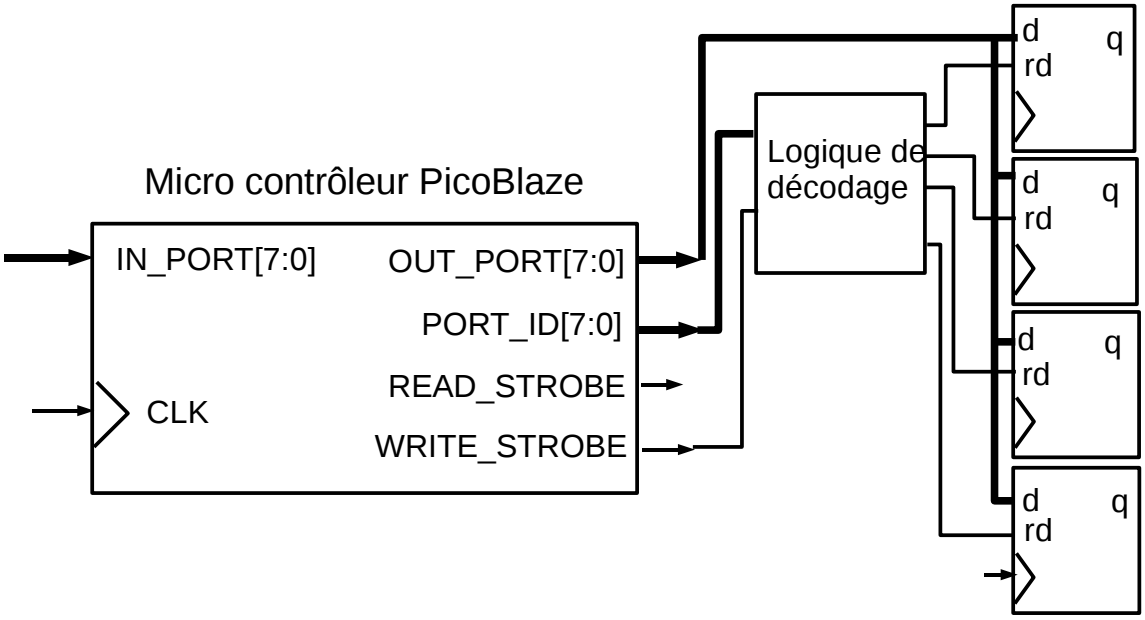


Schéma d'interface entre PORTS de sortie et PicoBlaze

Pour quatre ports seuls les deux bits de poids faible de PORT_ID[7:0] peuvent être utilisés. On donne pour une meilleure compréhension les programmes VHDL correspondants :

```
-- circuit décodage
process(write_strobe,port_id) begin
  if write_strobe='0' then
    en_d <= "0000";
  else
    case port_id(1 downto 0) is
      when "00" => en_d <= "0001";
      when "01" => en_d <= "0010";
      when "10" => en_d <= "0100";
      when others => en_d <= "1000";
    end case;
  end if;
end process;
```

Remarquons qu'il est possible de simplifier largement le décodage si on utilise comme numéros de PORT : 1, 2, 4 et 8 (au lieu de 1, 2, 3 et 4) et du coup les 4 bits de poids faible de PORT_ID[7:0] :

```
-- circuit décodage simplifié
process(write_strobe,port_id) begin
  if write_strobe='0' then
    en_d <= "0000";
  else
    en_d <= port_id(3 downto 0);
  end if;
end process;
```

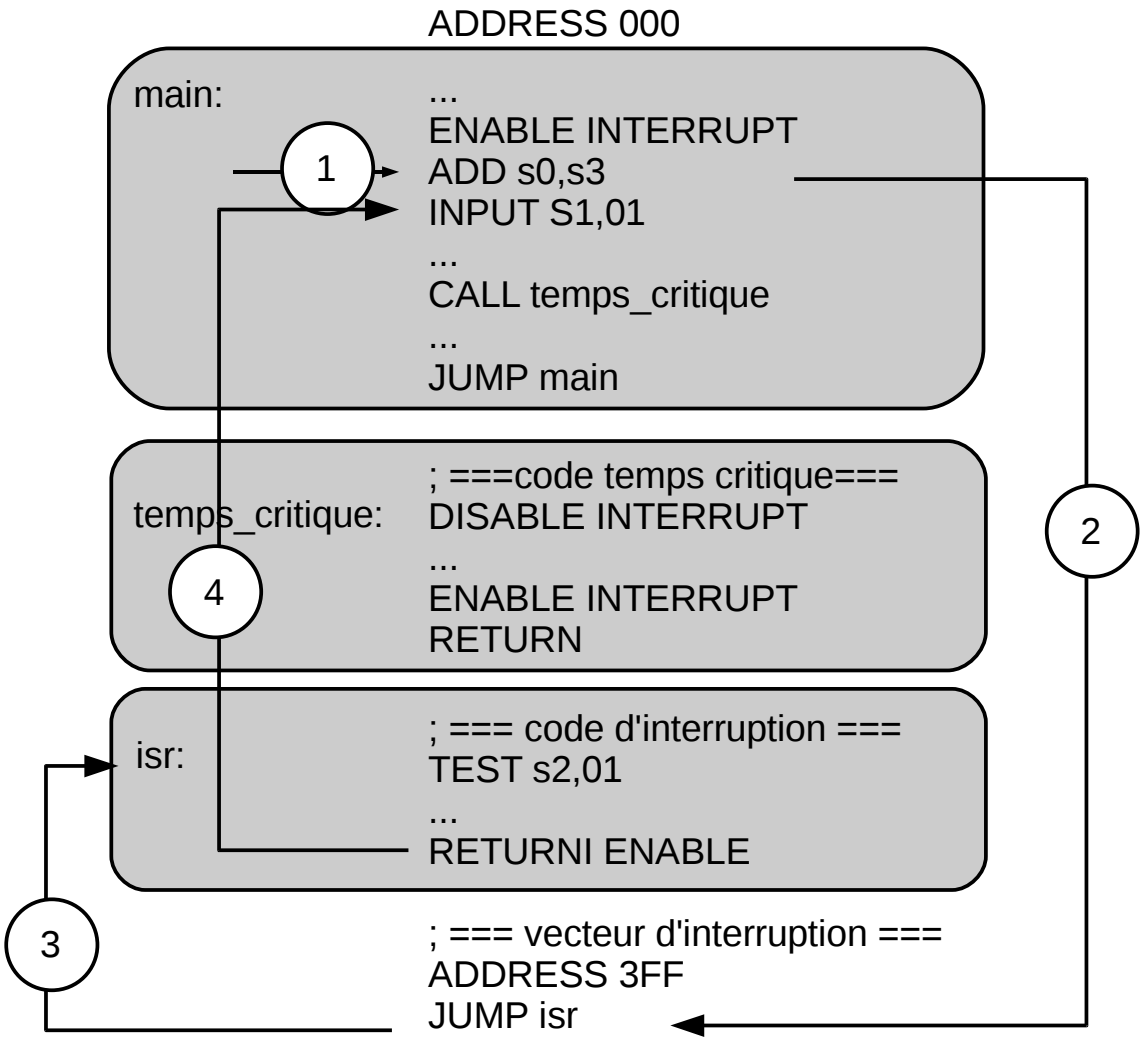
Un autre problème va nous intéresser maintenant : les interruptions.

II.3 - Les interruptions

La gestion des interruptions est en général très dépendante de l'application à réaliser. Encore une fois le PicoBlaze ne comprend par défaut que le minimum de logique pour gérer les interruptions. Quand on dit minimum, c'est le minimum : même pas de gestion de timer par exemple. C'est à l'utilisateur de réaliser la logique correspondante charger de réaliser le seul signal « INTERRUPT » présent (une seule source d'interruption donc par défaut). L'activation de ce signal force le PicoBlaze à réaliser un saut vers l'adresse 3FF (dernière adresse de la mémoire programme). A noter qu'après un RESET le signal d'interruption est inactif : une instruction « ENABLE INTERRUPT » doit être réalisée pour l'activer. A noter que l'exécution d'une interruption préserve les bits C et Z du registre d'état.

3.1 - Partie logicielle

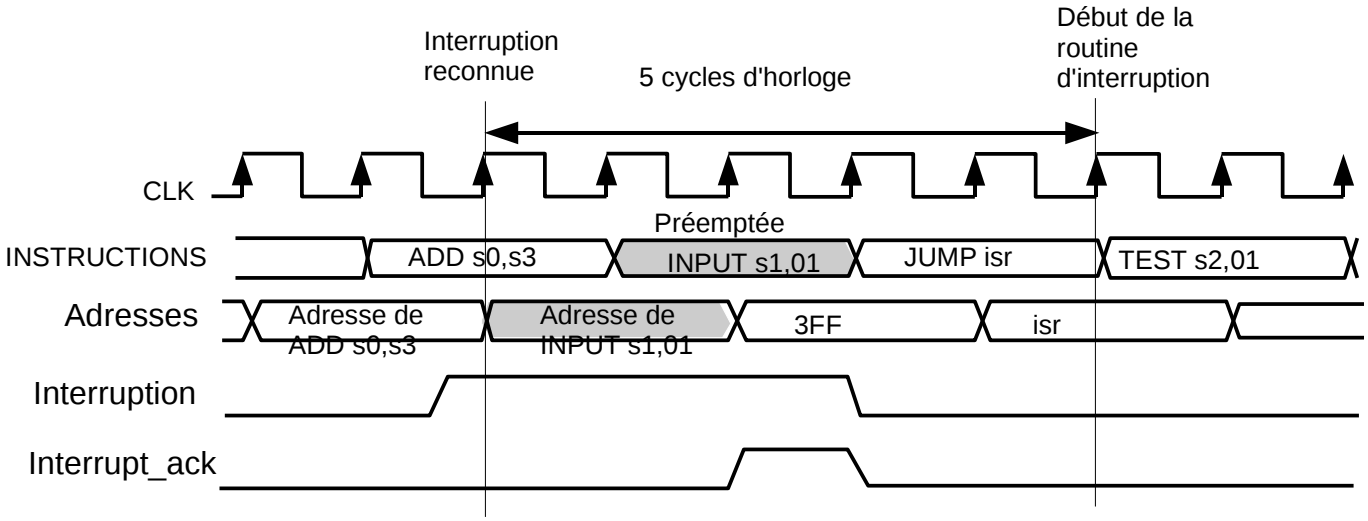
L'architecture d'un programme assembleur gérant une interruption sera comme :



Lorsqu'une interruption (1) arrive, le programme est dérouté (2) à l'adresse 3FF. Cette adresse branche (3) à la routine d'interruption proprement dite. Puis quand la routine est terminée on revient là où l'on a été interrompu (4).

3.2 - Chronogramme

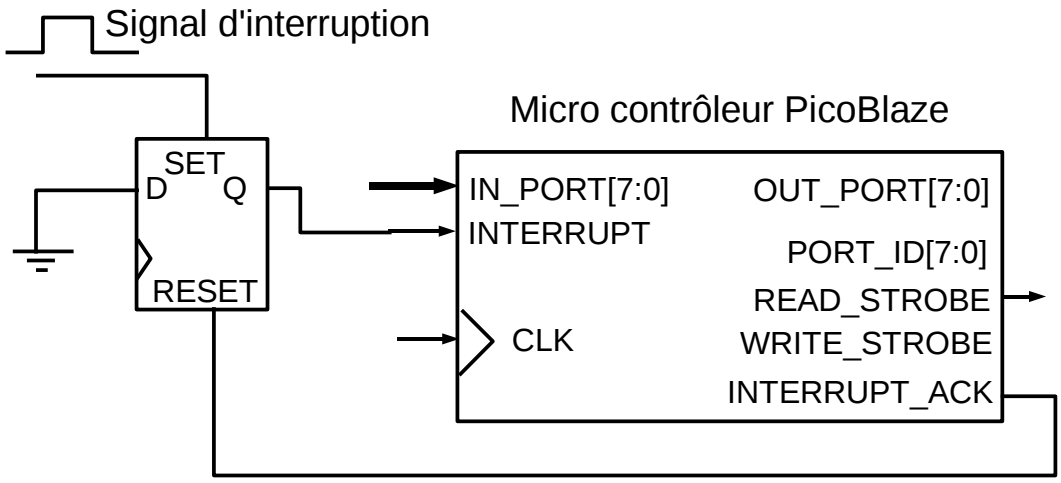
La routine d'interruption n'est pas instantanée dès réception du signal d'interruption. L'instruction en cours continue à s'exécuter puis une fois terminée l'ensemble des opérations décrites dans la section précédente sont synchronisées avec l'horloge comme indiqué sur le chronogramme suivant :



Timing d'un événement interruption

3.3 - Partie matérielle

Un exemple simple comprenant une seule source d'interruption est présenté maintenant.



Logique d'interruption simple

Les signaux SET et RESET sont asynchrones.

Chapitre III Les instructions du PicoBlaze

III.1 - Instructions de contrôle

Les instructions de contrôle peuvent toutes être conditionnelles, dépendant du positionnement ou pas des deux drapeaux du registre d'état, à savoir C et Z.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
JUMP	aaaaaaaaaa		1	1	0	1	0	0	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP Z,	aaaaaaaaaa		1	1	0	1	0	1	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP NZ,	aaaaaaaaaa		1	1	0	1	0	1	0	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP C,	aaaaaaaaaa		1	1	0	1	0	1	1	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP NC,	aaaaaaaaaa		1	1	0	1	0	1	1	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
CALL	aaaaaaaaaa		1	1	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL Z,	aaaaaaaaaa		1	1	0	0	0	1	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL NZ,	aaaaaaaaaa		1	1	0	0	0	1	0	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CALL C,	aaaaaaaaaa		1	1	0	0	0	1	1	0	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CALL NC,	aaaaaaaaaa		1	1	0	0	0	1	1	1	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																
RETURN			1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN Z			1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN NZ			1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN C			1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN NC			1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

III.2 - Instructions de mémorisations

64 octets de RAM sont accessibles à ces instructions soit en adressage direct, soit en adressage indexé. FETCH va chercher les données tandis-que STORE les stocke.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																
FETCH	sX	aaaaaa	0	0	0	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

FETCH	sX	(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																
STORE	sX	aaaaaa	1	0	1	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

STORE	sX	(sY)	1	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

III.3 - Instructions arithmétiques

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADD	sX	aaaaaaaa	0	1	0	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADD	sX	sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADDCY additionne les deux opérandes avec le drapeau de retenue C.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADDCY	sX	aaaaaaaa	0	1	1	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADDCY	sX	sY	0	1	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SUB	sX	aaaaaaaa	0	1	1	1	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SUB	sX	sY	0	1	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SUBCY	sX	aaaaaaaa	0	1	1	1	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SUBCY	sX	sY	0	1	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

L'instruction COMPARE réalise une soustraction mais, contrairement à l'instruction SUB, le résultat n'est pas mis à jour. Seuls les bits du regitre d'état sont positionnés.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
COMPARE	sX	aaaaaaaa	0	1	0	1	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

COMPARE	sX	sY	0	1	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

III.4 - Instructions logiques

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
LOAD	sX	aaaaaaaa	0	0	0	0	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LOAD	sX	sY	0	0	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
AND	sX	aaaaaaaa	0	0	1	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AND	sX	sY	0	0	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
OR	sX	aaaaaaaa	0	0	1	1	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

OR	sX	sY	0	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
XOR	sX	aaaaaaaa	0	0	1	1	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

XOR	sX	sY	0	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

L'instruction de test fait un ET logique entre deux opérandes mais, contrairement à l'instruction AND, le résultat final n'est pas mis à jour. Seuls les bits d'état sont mis à jour.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
TEST	sX	aaaaaaaa	0	1	0	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

TEST	sX	sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

III.5 - Instructions d'entrées sorties

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
INPUT	sX	pppppppp	0	0	0	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INPUT	sX	(sY)	0	0	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
OUTPUT	sX	pppppppp	1	0	1	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

OUTPUT	sX	(sY)	1	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

III.6 - Instructions d'interruptions

Les instructions qui permettent d'autoriser ou non les interruptions sont :

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ENABLE	INTERRUPT		1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

DISABLE	INTERRUPT		1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

L'instruction RETURNI doit être utilisée à chaque fin d'interruption. Elle fonctionne comme RETURN sauf qu'elle restore les bits de retenue et de détection de zéro du registre d'état.

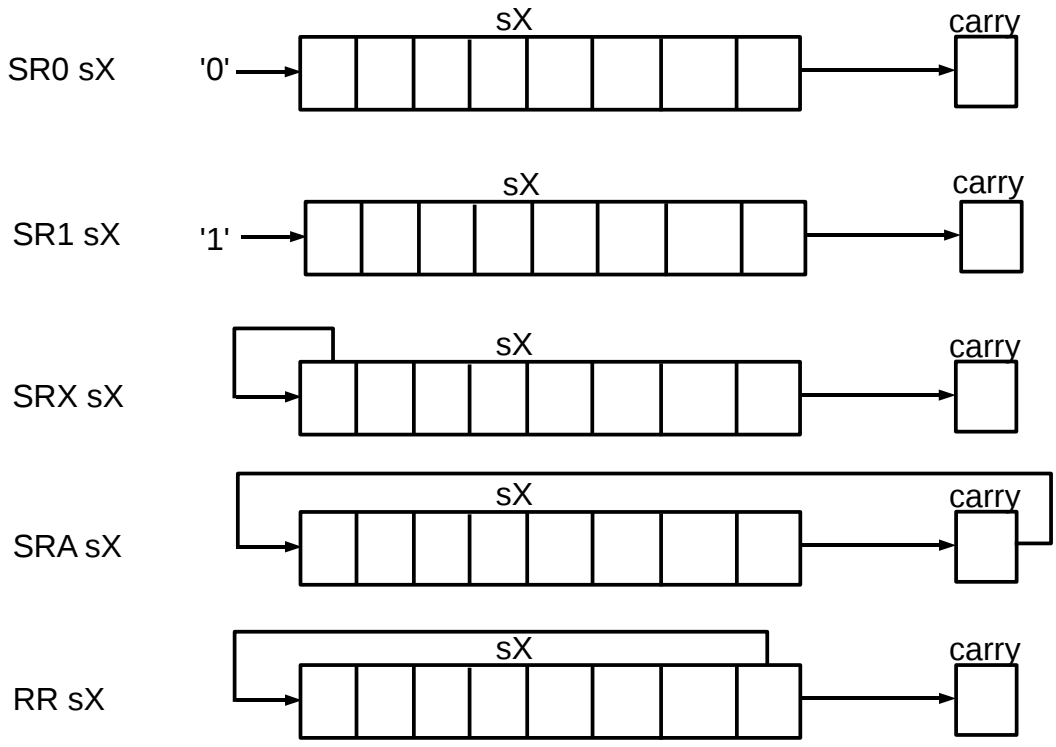
Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
RETURNI	ENABLE		1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURNI	DISABLE		1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

III.7 - Intructions de décalages

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SR0	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	1	0

			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SR1	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	1	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRX	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	0	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRA	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RR	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SL0	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SL1	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SLX	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SLA	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RL	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Chapitre IV SOLUTIONS

Exercice 5 p 7

```
constant MAX, 4
namereg s0,i
NAMEREG s2, s˙lsb  ; rename register s0 as "s˙lsb"
NAMEREG s3, s˙msb  ; rename register s0 as "s˙msb"
debut:
        LOAD i, MAX
        ; initialisation des pds faible (lsb) et fort de somme (msb)
        XOR s˙lsb,s˙lsb
        XOR s˙msb,s˙msb
boucle:
        INPUT s1,0
        ADD s˙lsb, s1
        ADDCY s˙msb, 0
        SUB i,01
        JUMP NZ,boucle
  ;division par 2
        SR0 s˙msb
        SRA s˙lsb
  ;encore division par 2
        SR0 s˙msb
        SRA s˙lsb
  ; sortie en adresse 04
        OUTPUT s˙lsb,04
        JUMP debut
```

Exercice 7 p8

En refaisant le même problème en stockant dans un tableau avant de faire le calcul de la somme :

```
constant MAX, 4
constant ADRTAB,0
namereg s0,i
NAMEREG s2, s˙lsb  ; rename register s0 as "s˙lsb"
NAMEREG s3, s˙msb  ; rename register s0 as "s˙msb"
debut:
        LOAD i, MAX
        ; initialisation des pds faible et fort de somme
        XOR s˙lsb,s˙lsb
        XOR s˙msb,s˙msb
        ; initialisation de l'index s4
        LOAD s4,ADRTAB
boucle:
        INPUT s1,0
        STORE s1,(s4)
        ADD s4,01
        SUB i,01
        JUMP NZ,boucle
        ; on repart pour une boucle
        LOAD i,MAX
        LOAD s4,ADRTAB
boucle2:
        FETCH s1,(s4)
        ADD s4,01
```

```
        ADD s'lsb, s1
        ADDCY s'msb, 0
        SUB i,01
        JUMP NZ,boucle2
  ;division par 2
        SR0 s'msb
        SRA s'lsb
  ;encore division par 2
        SR0 s'msb
        SRA s'lsb
  ; sortie en adresse 04
        OUTPUT s'lsb,04
        JUMP debut
```

Exercice 8 p 8

En refaisant le même problème en stockant dans un tableau avant de faire le calcul de la somme et en gérant le maximum :

```
;;; calcul du maximum
constant MAX, 4
constant ADRTAB,0
namereg s0,i
NAMEREG s2, s'lsb  ; rename register s0 as "s'lsb"
NAMEREG s3, s'msb  ; rename register s0 as "s'msb"
NAMEREG s5, maxi
debut:
        LOAD i, MAX
        ; initialisation des pds faible et fort de somme
        XOR s'lsb,s'lsb
        XOR s'msb,s'msb
        XOR maxi,maxi
        ; initialisation de l'index s4
        LOAD s4,ADRTAB
boucle:
        INPUT s1,0
        STORE s1,(s4)
        ADD s4,01
        SUB i,01
        JUMP NZ,boucle
        ; on repart pour une boucle
        LOAD i,MAX
        LOAD s4,ADRTAB
boucle2:
        FETCH s1,(s4)
        ADD s4,01
        COMPARE i,04
        JUMP NZ, suite
        LOAD maxi,s1
suite:
        COMPARE s1,maxi
        JUMP C,suite2
        LOAD maxi,s1
suite2:
        ADD s'lsb, s1
        ADDCY s'msb, 0
        SUB i,01
        JUMP NZ,boucle2
        STORE maxi,(s4)
  ;division par 2
        SR0 s'msb
        SRA s'lsb
  ;encore division par 2
        SR0 s'msb
        SRA s'lsb
  ; sortie en adresse 04
        OUTPUT s'lsb,04
        JUMP debut
```