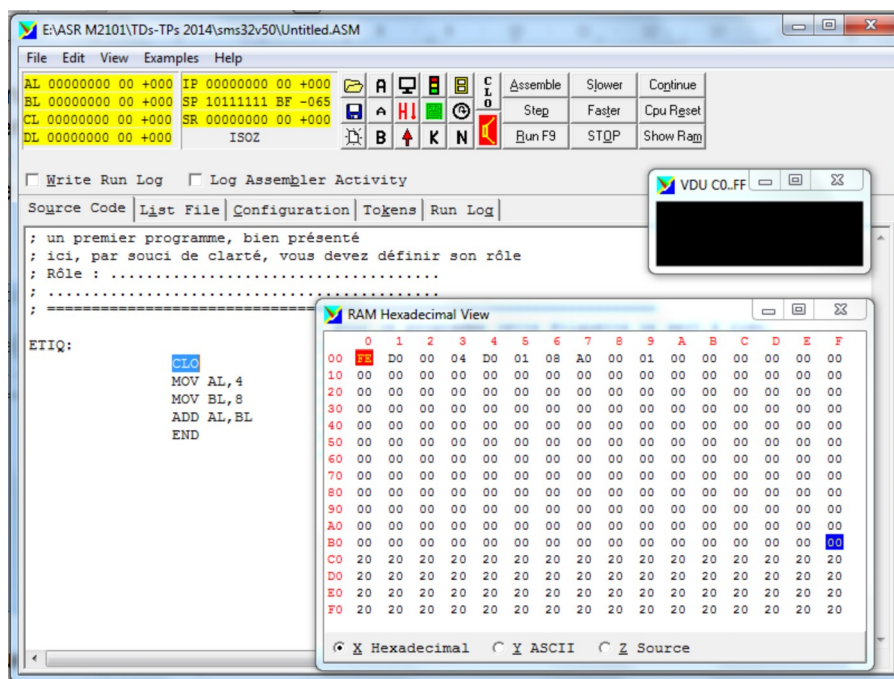


Assembleur



Cours Assembleur (rappels)

D. Bogdaniuk - P. Portejoie

1 - INTRODUCTION**1 - 1 - Langage d'assemblage**

Pourquoi s'intéresser à la programmation en langage d'assemblage alors que des langages évolués tels le C, le Pascal, l'Ada ou le COBOL, pour n'en citer que quelques uns, sont aujourd'hui couramment utilisés ? Il se trouve que, même si les langages existants offrent de larges possibilités et un grand confort, le langage d'assemblage s'impose partout où puissance et précision sont requises. Ceci était d'autant plus vrai à une époque où les Ko de mémoire et les MHz du processeur étaient comptés.

Les programmes en langage d'assemblage peuvent offrir d'immenses possibilités. Ils sont courts et s'exécutent beaucoup plus rapidement que les mêmes programmes écrits dans un langage évolué. Leur précision leur permet de faire des choses impossibles avec n'importe quel autre langage.

Mais ils perdent malheureusement certaines de leurs qualités et soulèvent bien des problèmes dès lors que leur longueur s'accroît. Ce défaut trouve son explication dans le fait que ceux-ci accordent beaucoup d'importance aux détails. En effet, le langage d'assemblage oblige le programmeur à décider des moindres actions de la machine.

Alors, quand faut-il utiliser le langage d'assemblage ? Il est évident que l'on ne doit recourir à ce type de programmation que lorsqu'on ne peut faire autrement ou bien lorsque que la motivation principale est le souci de performance dans son acception générale, c'est à dire "temps d'exécution" ou "encombrement mémoire".

1 - 2 - Langage machine

L'unité élémentaire d'information dans un ordinateur est le bit (0/1) auquel correspond un niveau de tension électrique. On n'a heureusement pas à se préoccuper de tensions lorsqu'on écrit des programmes, mais seulement des nombres. L'information stockée dans un ordinateur est ainsi représentée au moyen du système de numération binaire.

L'unité d'échange élémentaire en mémoire étant le mot, constitué au minimum d'un octet (2 quartets, 8 bits) il est commode de désigner l'ensemble des bits qui le composent par un nombre que l'on exprime en hexadécimal. L'arithmétique hexadécimale pose un léger problème étant donné que les seuls chiffres dont on dispose sont ceux de 0 à 9. Il a donc été convenu que les nombres 10 à 15 seraient représentés au moyen des six premières lettres de l'alphabet A à F.

Un programme informatique est ainsi constitué d'une suite d'instructions représentée par des 0 et des 1 contenus en mémoire. C'est ce qu'on appelle le langage machine.

L'ordinateur s'évertue perpétuellement à rechercher dans sa mémoire (phase *fetch*), d'une manière bien définie, les instructions qu'il exécute (phase *execute*) les unes après les autres. L'élément exécutif dans un ordinateur est le microprocesseur capable de ne comprendre qu'un jeu limité d'instructions codées en binaire. Ce dernier recense les opérations élémentaires qu'un microprocesseur est capable de réaliser. Celles-ci peuvent être classées en trois grandes catégories :

- les opérations d'E/S
- les opérations de calcul (addition, soustraction, multiplication, division)
- les opérations de comparaison

Il est ainsi possible de concevoir un programme en écrivant en mémoire les suites de bits (sous forme hexadécimale par facilité), correspondant aux différentes instructions. On imagine facilement la pénibilité de la tâche et le risque d'erreur inhérent à ce type de travail. En outre, le langage machine présente un caractère fortement ésotérique et n'est porteur que d'une très faible information vis à vis du lecteur. Par exemple l'addition de deux nombres pourrait s'écrire :

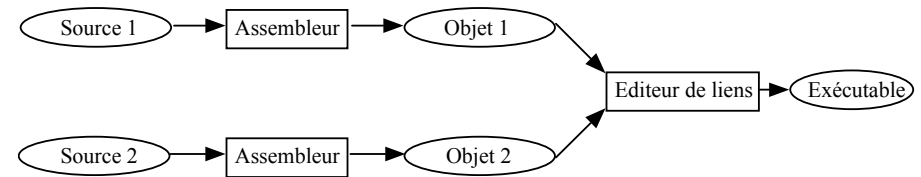
00000011 11000011 ou en hexadécimal 03 C3

Fort heureusement, le langage d'assemblage nous permet de nous affranchir des contraintes du langage machine. Chaque instruction est, dans ce langage, identifiée par un mnémonique. Ainsi l'instruction précédente s'écrirait :

ADD AX, BX (où AX et BX sont des registres)

Le logiciel qui assure la traduction du langage d'assemblage en langage machine s'appelle un assembleur. Il existe en fait autant de langages d'assemblage que de microprocesseurs différents.

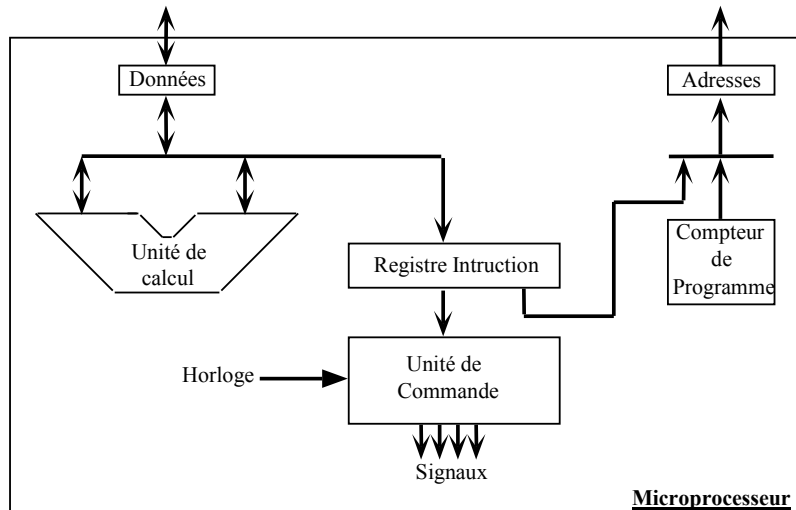
L'assembleur ne produit pas directement du code exécutable, mais seulement un code intermédiaire appelé code objet. Le programme exécutable ne sera obtenu qu'après passage dans l'éditeur de liens :

**1 - 3 - Le microprocesseur**

Le microprocesseur utilise un espace de travail, la mémoire, qui contient à la fois le programme et les données sur lesquelles ce dernier opère. Il s'appuie sur un mécanisme complexe, commandé par une horloge unique, qui assure l'interprétation des codes-instructions et les changements d'états qui en découlent.

On y retrouve les deux parties caractéristiques des circuits séquentiels :

- l'*Unité de Calcul* réalise toutes les manipulations nécessaires aux calculs. Elle n'agit que si elle reçoit des ordres et est incapable d'en générer.
- l'*Unité de Commande* ne sait rien faire d'autre qu'élaborer des ordres pour l'unité de traitement à partir d'informations qu'elle lit en mémoire et qu'elle interprète comme des instructions. Elle contient en particulier :
 - le Compteur Ordinal (CO), encore appelé PC (Program Counter) ou encore IP (Instruction Pointer) qui contient l'adresse de la prochaine instruction à exécuter
 - le registre d'instruction RI (ou IR pour Instruction Register) qui contient l'information à analyser et à exécuter
 - des indicateurs d'états contenus dans le Registre d'Etat (RE) encore appelé SR (Status Register), informant sur l'état de la machine et sur la façon dont s'est déroulée l'instruction précédente
 - des registres généraux



2 - PRESENTATION SIMPLIFIEE DU 8086 et du simulateur

2 - 1 - La mémoire

L'unité adressable est l'octet (8 bits). L'adressage dans le 8086 se fait à l'intérieur de zones de 2¹⁶ octets. Les adresses figurant dans les instructions occupent donc 16 bits. Le 8086 permet ainsi de manipuler en une seule instruction des emplacements mémoire de 16 bits appelés mots.

Le simulateur dispose pour sa part d'une mémoire de 256 octets et ne sait manipuler que des octets.

2 - 2 - L'Unité de Commande

L'unité de commande possède :

- 4 registres généraux de 16 bits notés AX, BX, CX et DX se décomposant chacun en 2 registres de 8 bits notés AH, AL, BH, BL, CH, CL, DH, DL (H pour High et L pour Low)
- un registre d'état (SR) qui regroupe les indicateurs positionnés en fonction du résultat de certaines instructions. Ces indicateurs sont utilisés par les instructions de branchement conditionnel. Les principaux indicateurs sont :
 - OF (O) positionné s'il y a débordement de capacité lors d'un calcul
 - SF (S) positionné si le calcul donne un résultat négatif
 - ZF (Z) positionné si le résultat d'un calcul est nul
 - IF (I) autorise ou non les interruptions
- un registre de pile de 16 bits noté SP qui contient à tout moment l'adresse du sommet de pile. Le principe de celle-ci est de ne permettre l'accès qu'à la dernière information stockée.
- 2 registres d'index de 16 bits notés SI et DI
- le compteur ordinal (IP) qui contient l'adresse de la prochaine instruction à exécuter

Le simulateur simule un microprocesseur 8 bits semblable aux 8 bits de poids faible de la famille 80X86 et disposant de 5 ports d'E/S d'adresses 0 à 4. Il possède 4 registres généraux notés AL, BL, CL et DL et un registre de pile de 8 bits noté SP. Il dispose de 256 octets de mémoire dont les adresses vont de 00 à FF en hexadécimal. La mémoire vidéo s'établit de C0 à FF et la pile croît depuis BF vers 00.

3 - REPRESENTATION DES DONNEES & FORME D'UN PROGRAMME

3 - 1 - Représentation des données

Une information stockée en mémoire n'a a priori aucune signification. Ca n'est qu'une suite de valeurs binaires. C'est l'usage qui en est fait par le programme qui détermine son type.

Ainsi, la suite binaire 1001 1100 notée 93 en hexadécimal peut signifier :

- le caractère "é" codé en ASCII
- le code SHL
- le nombre décimal 156 écrit en binaire
- ...

Représentation des caractères:

Le code ASCII est utilisé pour les représenter. Mais les opérations arithmétiques n'ont aucun sens sur leur représentation. En effet l'addition du code ASCII des caractères "1" et "2" ne fournit pas le code ASCII du caractère "3".

Représentation des nombres:

Les nombres entiers naturels sont représentés en binaire pur tandis que les nombres entiers relatifs le sont en complément à deux.

Cette méthode de représentation sépare l'intervalle de représentation des nombres en deux parties : l'une pour les nombres positifs (le bit de poids fort est à 0) et l'autre pour les nombres négatifs (le bit de poids fort est à 1).

Ainsi se pose la question de la validité du calcul, selon qu'il s'agit d'entiers naturels ou d'entiers. Le résultat sera juste dans le cas d'entiers naturels s'il le bit de débordement (O) n'est pas positionné et il sera juste dans le cas d'entiers relatifs si le bit de débordement (O) est égal au bit de signe (S).

3 - 2 - Forme d'un programme

Un programme en langage d'assemblage est constitué d'une suite de lignes, bien que les instructions puissent être écrites les une derrière les autres sur la même ligne. Une ligne peut être :

- une instruction, qui engendre une instruction machine
- une directive qui n'engendre pas d'instruction machine. C'est une indication fournie à l'assembleur.

Syntaxe d'une ligne d'instruction:

Elle se décompose en trois zones:

zone étiquette	zone commande	zone argument
[identificateur]	mnémotique	[opérandes]

Syntaxe d'une ligne de directive:

Elle se décompose en deux zones :

zone commande	zone argument
<i>directive</i>	<i>[opérande]</i>

Les principales directives sont:

- DB (réservation et initialisation mémoire)
- ORG (implantation mémoire)
- END (fin du programme)

Tout le texte suivant un ";" est considéré par l'assembleur comme un commentaire.

4 - MECANISMES D'ADRESSAGE

La partie adresse d'une instruction peut contenir l'adresse d'un octet. Dans tous les cas, l'adresse est traitée comme un nombre positif.

4 - 1 - Adressage inhérent

On parle d'adressage inhérent lorsque la partie adresse est absente de l'instruction.

Ex:

PUSHF

Place le contenu du registre d'état sur la pile.

4 - 2 - Adressage immédiat

La valeur de l'opérande apparaît directement dans l'instruction.

Ex:

MOV AL, 5

Range la valeur hexadécimale 5 dans le registre AL

AL 5 ← 5

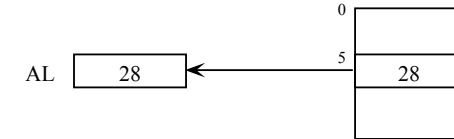
4 - 3 - Adressage direct

L'instruction contient une adresse qui constitue l'adresse finale.

Ex:

MOV AL, ZONE (non disponible sur simulateur)
MOV AL, [5]

Range le contenu de l'emplacement mémoire identifié par ZONE, ou le contenu de l'adresse hexadécimale 5 dans le registre AL.

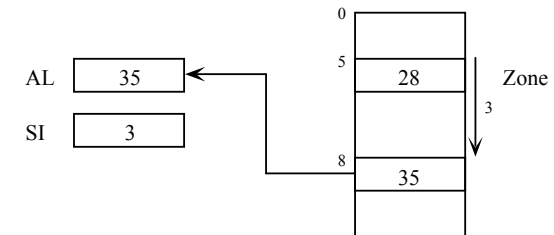
**4 - 4 - Adressage direct indexé** (non disponible sur simulateur)

L'instruction contient une adresse et un déplacement par rapport à cette adresse. La somme des deux constitue l'adresse finale. Le déplacement est exprimé par un registre d'index.

Ex:

MOV AL, ZONE[SI]

Range le contenu de l'emplacement mémoire identifié par ZONE, décalé de la valeur de SI dans le registre AL.

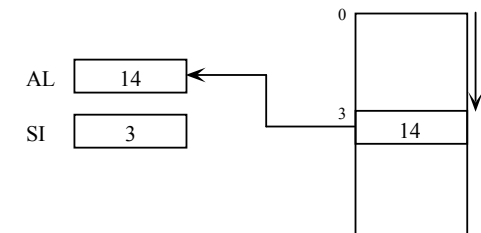
**4 - 5- Adressage implicite** (non disponible sur simulateur)

C'est un cas particulier d'adressage direct indexé dans lequel il n'y a pas d'adresse mais simplement un déplacement.

Ex:

MOV AL, [SI]

Range le contenu de l'emplacement mémoire dont l'adresse est 0 décalé de la valeur de SI dans le registre AL.



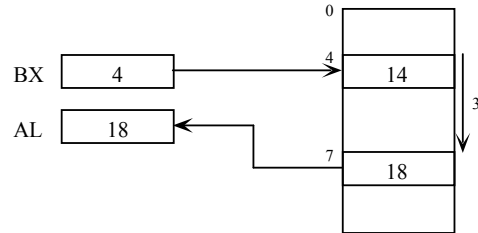
4 - 6- Adressage basé (non disponible sur simulateur)

L'instruction contient une adresse et un déplacement exprimé sous forme d'une constante entière.

Ex:

```
MOV AL, [BX] + 3
```

Range le contenu de l'emplacement mémoire dont l'adresse est dans le registre de base, décalé de 3.

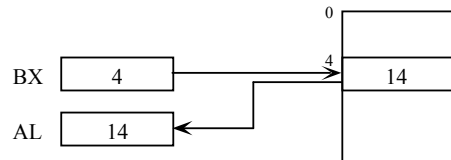
**4 - 7 - Adressage indirect****Indirection par un registre:**

La partie adresse de l'instruction contient un nom de registre de base (BX ou BP). L'adresse finale est égale au contenu du registre de base.

Ex:

```
MOV AL, [BX]
```

Range le contenu de l'emplacement dont l'adresse est dans le registre de base dans AL.

**Indirection par la mémoire:** (non disponible sur simulateur)

La partie adresse de l'instruction contient une adresse dont le contenu est l'adresse finale.

Ex:

```

AdrPGM   RW   1
...
MOV AX,   OFFSET PGM
MOV AdrPGM, AX
...
JMP AdrPGM

PGM:     ...

```

Range l'adresse de la séquence d'instructions démarrant à l'étiquette PGM dans une zone mémoire nommée AdrPGM et effectue un branchement à PGM.

"OFFSET" fait référence à l'adresse de la séquence d'instructions.

L'instruction "JMP AdrPGM" a pour effet de mettre le contenu de l'emplacement mémoire référencé par AdrPGM dans le compteur ordinal.

**4 - 8 - Adressage indirect indexé** (non disponible sur simulateur)

Ce mécanisme est une combinaison de l'indirection et de l'indexation. Ce qui permet de distinguer 2 méthodes :

- la post-indexation
- la pré-indexation

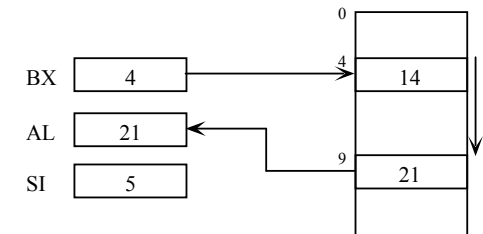
Post-indexation :

L'instruction contient le nom d'un registre de base (BX ou BP) et le nom d'un registre d'index (SI ou DI). L'adresse finale est obtenue en cumulant le contenu des registres de base et d'index.

Ex:

```
MOV AL, [BX+SI]
```

Range le contenu de l'emplacement mémoire dont d'adresse est la somme de BX et de SI dans AX.

**Pré-indexation :**

L'instruction contient une adresse et le nom d'un registre d'index (SI ou DI). L'adresse finale est le contenu de l'emplacement mémoire dont l'adresse est la somme de l'adresse se trouvant dans l'instruction et du registre d'index.

Ex:

```
JMP TabPGM[SI]
```

Effectue un branchement à l'adresse contenue dans l'emplacement mémoire référencé par TabPGM, décalé de la valeur du registre d'index..

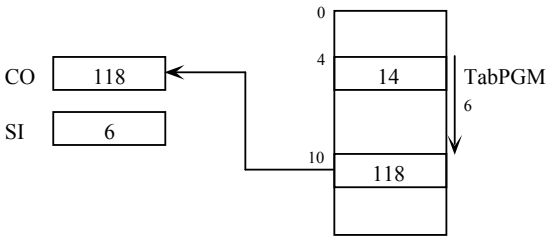


Table des codes ASCII

		Effet de la touche Ctrl							
		000	001	010	011	100	101	110	111
b6	b5	b4	b3	b2	b1	b0			
0	1	2	3	4	5	6	7		
0000	0	0	NUL	DLE		0	@	P	p
0001	1	1	SOH	DC1	!	1	A	Q	a
0010	2	2	STX	DC2	"	2	B	R	b
0011	3	3	ETX	DC3	#	3	C	S	s
0100	4	4	EOT	DC4	\$	4	D	T	t
0101	5	5	ENQ	NAK	%	5	E	U	u
0110	6	6	ACK	SYN	&	6	F	V	v
0111	7	7	BEL	ETB	'	7	G	W	w
1000	8	8	BS	CAN	(8	H	X	x
1001	9	9	HT	EM)	9	I	Y	y
1010	10	A	LF	SUB	*	:	J	Z	z
1011	11	B	VT	ESC	+	;	K	[{
1100	12	C	FF	FS	,	<	L	\	
1101	13	D	CR	GS	-	=	M]	}
1110	14	E	SO	RS	.	>	N	^	~
1111	15	F	SI	US	/	?	O	_	DEL

32 caractères de contrôle

Effet de la touche Maj

Exemple : la représentation de B

- en binaire : 0100 0010
- en hexadécimal : 4 2

Format des instructions :

[ETIQUETTE :] Commande [Argument1], [Argument2]

Alternatives :

<u>si</u> (A<B) <u>alors</u> <suite d'instructions 1> <u>sinon</u> <suite d'instructions 2> <u>finsi</u> ...	CMP AL, BL JNS SI_ANOTINFB SI_AINFB : <suite d'instructions 1> JMP FINSI SI_ANOTINFB : <suite d'instructions 2> FINSI : ...
---	--

<u>si</u> (A=B) <u>alors</u> <suite d'instructions 1> <u>sinon</u> <suite d'instructions 2> <u>finsi</u> ...	CMP AL, BL JNZ SI_ANOTEQB SI_AEQB : <suite d'instructions 1> JMP FINSI SI_ANOTEQB : <suite d'instructions 2> FINSI : ...
---	---

<u>si</u> (A>B) <u>alors</u> <suite d'instructions 1> <u>sinon</u> <suite d'instructions 2> <u>finsi</u> ...	CMP AL, BL JS SI_ANOTSUPB JZ SI_ANOTSUPB SI_ASUPB : <suite d'instructions 1> JMP FINSI SI_ANOTSUPB : <suite d'instructions 2> FINSI : ...
---	---

Itérations :

<u>pour</u> A:=1 <u>à</u> 5, <u>pas</u> 1 <suite d'instructions> <u>finpour</u> ...	MOV AL, 0 POUR : INC AL CMP AL, 5 JZ FINPOUR <suite d'instructions> JMP POUR FINPOUR : ...
--	--

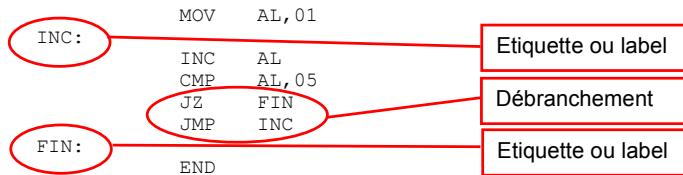
<u>tant que</u> A=B <suite d'instructions> <u>fintantque</u> ...	TANTQUE : CMP AL, BL JNZ FINTANTQUE <suite d'instructions> JMP TANTQUE FINTANTQUE : ...
---	---

<u>répéter</u> <suite d'instructions> <u>jusqu'à</u> A=B ...	REPETER : <suite d'instructions> CMP AL, BL JNZ REPETER FINREPETER : ...
---	---

Calcul des SAUTS

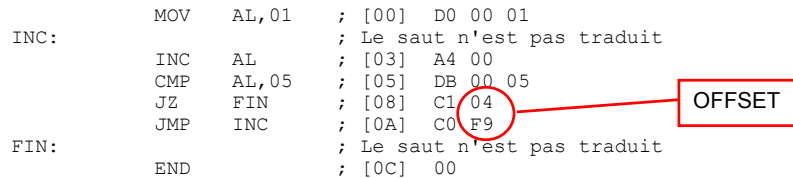
Les sauts en assembleur se font grâce aux instructions telles que JS, JNS, JZ, JNZ, JMP...

Exemple de programme avec des sauts :



Les étiquettes ou labels ne sont pas des instructions. Elles ne sont traduites par rien lors de la phase d'assemblage. Elles permettent uniquement au programmeur de placer les débranchements des sauts. Lors de l'assemblage, les débranchements sont transformés en offset, c'est à dire en déplacements. L'offset est un déplacement en octet vers l'adresse d'implantation de l'instruction à exécuter vers laquelle pointe le saut.

Exemple de programme avec des sauts :



Si l'offset est positif ($\leq 7F$) le déplacement est obligatoirement en direction de la fin du programme. Dans l'exemple $\langle 04 \rangle$ est positif, donc pointe vers un débranchement vers le bas de 4 octets après l'adresse d'implantation de l'instruction de saut qui l'utilise, ici l'instruction $\langle C1 \rangle$ implantée à l'adresse $[08]$. Donc en comptant 4 octets après l'instruction $\langle C1 \rangle$, on arrive à l'instruction $\langle 00 \rangle$ implantée en $[0C]$, autrement dit la fin du programme.

On peut aussi ajouter la valeur de l'offset à l'adresse du saut et on obtient l'adresse de l'instruction du débranchement, $[08] + 04 = [0C]$.

Si l'offset est négatif (≥ 80) le déplacement est obligatoirement en direction du début du programme. Dans l'exemple $\langle F9 \rangle$ est négatif, donc pointe vers un débranchement vers le haut. On peut ajouter la valeur de l'offset à l'adresse du saut et on obtient l'adresse de l'instruction du débranchement, $[0A] + F9 = [03]$. L'adresse $[03]$ est celle de l'instruction $\langle A4 \rangle$.

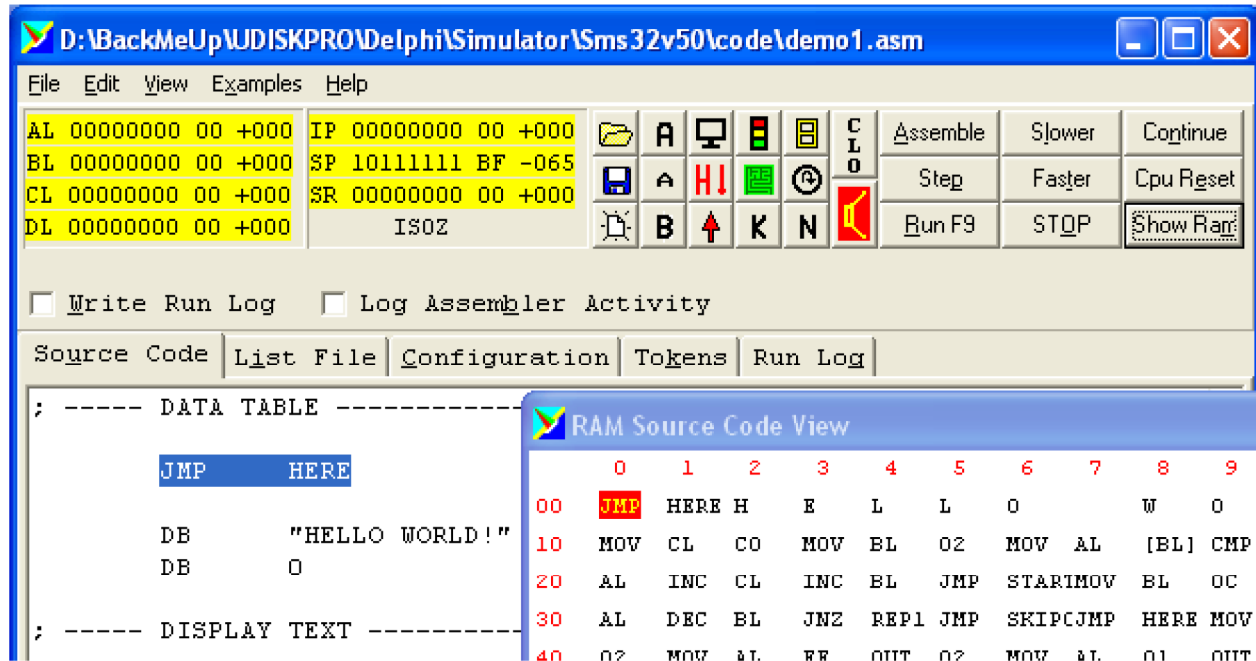
On peut également en complémentant à vrai ($C2$) la valeur de l'offset, obtenir directement l'adresse du débranchement : $(FF - F9 = 06)_{c1} \implies (06 + 1 = 07)_{c2}$ soit un déplacement de 7 octets avant l'instruction $\langle C0 \rangle$ qui amène à l'instruction $\langle A4 \rangle$.



Using the Simulator - Getting Started

[Website](#)

[Home](#) | [Previous](#) | [Next](#)



On Line Help

Press the **F1** key to get on line help.

Writing a Program

To write and run a program using the simulator, select the source code editor tab by pressing **Alt+U**.

Type in your program. It is best to get small parts of the program working rather than typing it all in at once.


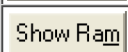
Here is a simple example. Also look at the tutorial example programs. You can type this into the simulator or copy and paste it. The assembly code has been annotated with comments that explain the code. These comments are ignored by the assembler program. Comments begin with a semicolon and continue to the end of the line.

```
; ===== COUNT =====
MOV     AL,0      ; Move 0 into the AL register
REP:    ADD      AL,2    ; Add two to AL
        JMP      REP    ; Jump back to the rep label



        END          ; Program ends here
; =====
```

Running a Program

Step	To run a program, you can step through it one line at a time by pressing Alt+P or by clicking this button repeatedly.
Run F9	You can run a program continuously by pressing F9 or Alt+R or by pressing this button
Slower	To speed up or slow down a running program use these buttons or type Alt+L or Alt+T
Faster	
STOP	To stop a running program press Alt+O or click or press Escape or press this button.
Continue	To restart a paused program, continuing from where it left off, press Alt+N or click this button.

	To restart a program from the beginning, reset the CPU by pressing Alt+E or click this button.
	To re-open the RAM display window, press Alt+M or click this button.

Assembly Code

	The code you type is called assembly code. This human-readable code is translated into machine code by the Assembler . The machine code (binary) is understood by the CPU. To assemble a program, press Alt+A or click this button.
<input type="checkbox"/> Log Assembler Activity	You can see an animation of the assembler process by checking this box.
	When you run or setp a program, if necessary, the code is assembled.

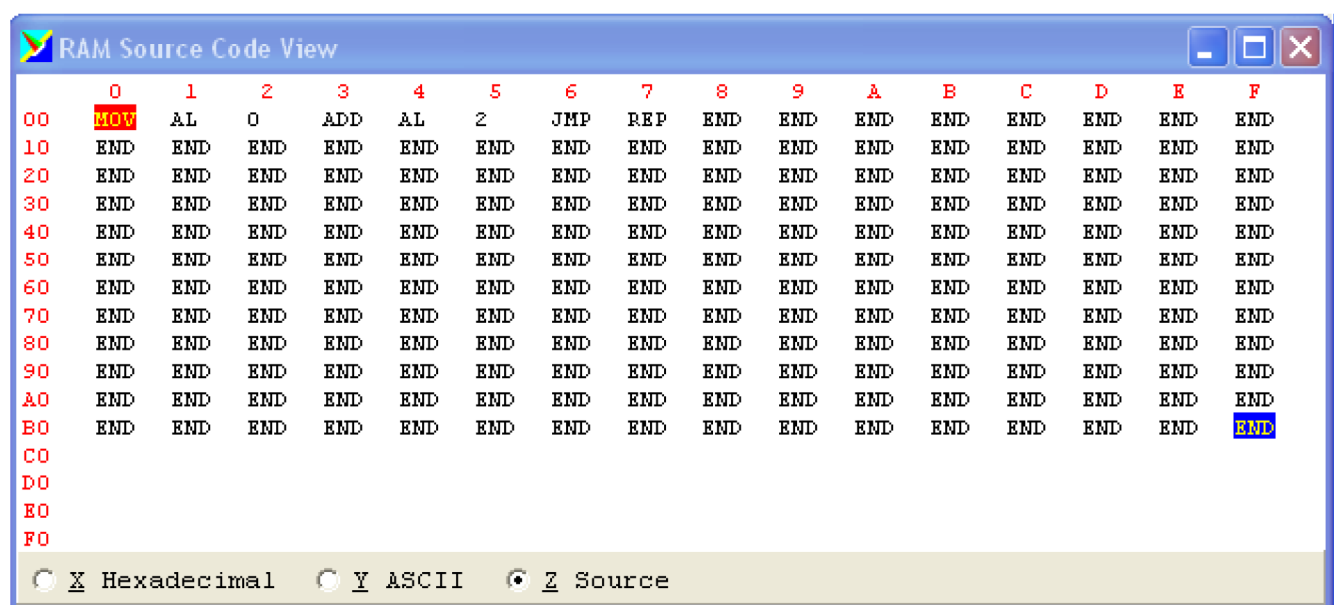
Assembler Phases

There is short delay while the assemblbler goes through all the stages of assembling the program. The steps are

1. **Save** the source code.
2. Convert the source code into **tokens** (this simulator uses human readable tokens for educational value rather than efficiency).
3. **Parse** the source code and (if necessary) generate error messages. If there are no errors, generate the machine codes. This process could be coded more efficiently. If the tokens representing machine op codes like MOV and JMP were numerical, the assembler could look up the machine code equivalents in an array instead of ploughing through many if-then-else statements. Once again, this has been done to demonstrate the process of assembling code for educational reasons.
4. **Calculate jumps**, the distances of the jump/branch instructions.

Viewing Machine Code

The machine code stored in RAM can be viewed in three modes by selecting the appropriate radio button.



Hexadecimal - This display corresponds exactly to the binary executed by the CPU.

ASCII - This display is convenient if your program is processing text. The text is readable but the machine codes are not.

Source Code - This display shows how the assembly code commands are placed in memory.

Tutorial Examples

The tutorial examples provide a step by step introduction to the commands and techniques of low level programming. Each program has one or more learning tasks associated with it. Some of the tasks are simple. Some are real brain teasers.

[Home](#) | [Previous](#) | [Next](#)



Instruction Set Summary

Move Instructions. Flags NOT set.

Assembler		Machine Code	Explanation	
MOV	AL,15	D0 00 15	AL = 15	Copy 15 into AL
MOV	BL,[15]	D1 01 15	BL = [15]	Copy RAM[15] into BL
MOV	[15],CL	D2 15 02	[15] = CL	Copy CL into RAM[15]
MOV	DL,[AL]	D3 03 00	DL = [AL]	Copy RAM[AL] into DL
MOV	[CL],AL	D4 02 00	[CL] = AL	Copy AL into RAM[CL]

Direct Arithmetic and Logic Instructions. Flags are set.

Assembler		Machine Code	Explanation	
ADD	AL,BL	A0 00 01	AL = AL + BL	
SUB	BL,CL	A1 01 02	BL = BL - CL	
MUL	CL,DL	A2 02 03	CL = CL * DL	
DIV	DL,AL	A3 03 00	DL = DL / AL	
MOD	AL,BL	A6 00 01	AL = AL mod BL	
INC	DL	A4 03	DL = DL + 1	
DEC	AL	A5 00	AL = AL - 1	
AND	AL,BL	AA 00 01	AL = AL AND BL	
OR	CL,BL	AB 02 01	CL = CL OR BL	
XOR	AL,BL	AC 00 01	AL = AL XOR BL	
NOT	BL	AD 01	BL = NOT BL	
ROL	AL	9A 00	Rotate bits left.	LSB := MSB
ROR	BL	9B 01	Rotate bits right.	MSB := LSB.
SHL	CL	9C 02	Shift bits left.	Discard MSB.
SHR	DL	9D 03	Shift bits right.	Discard LSB.

Immediate Arithmetic and Logic Instructions. Flags are set.

Assembler		Machine Code	Explanation	
ADD	AL,12	B0 00 12	AL = AL + 12	
SUB	BL,15	B1 01 15	BL = BL - 15	
MUL	CL,03	B2 02 03	CL = CL * 03	
DIV	DL,02	B3 03 02	DL = DL / 02	
MOD	AL,10	B6 00 10	AL = AL mod 10	
AND	AL,0F	BA 00 0F	AL = AL AND 0F	
OR	CL,F0	BB 02 F0	CL = CL OR F0	
XOR	AL,AA	BC 00 AA	AL = AL XOR AA	

Compare Instructions. Flags are set.

Assembler		Machine Code	Explanation	
CMP	AL,BL	DA 00 01	Set 'Z' if AL = BL	
			Set 'S' if AL less than BL	
CMP	BL,13	DB 01 13	Set 'Z' if BL = 13	
			Set 'S' if BL less than 13	
CMP	CL,[20]	DC 02 20	Set 'Z' if CL = [20]	
			Set 'S' if CL less than [20]	

Branch Instructions. Flags NOT set.

Depending on the size and direction of the jump, different machine codes can be generated. Jump instructions cause the instruction pointer (IP) to be altered. The largest possible jumps are +127 bytes and -128 bytes.

Assembler		Machine Code	Explanation
JMP	HERE	C0 12 C0 FE	Increase IP by 12 Decrease IP by 2 (twos complement)
JZ	THERE	C1 09 C1 9C	Increase IP by 9 if Z flag is set Decrease IP by 100 if Z flag is set
JNZ	A_PLACE	C2 04 C2 F0	Increase IP by 4 if Z flag not set Decrease IP by 16 if Z flag not set
JS	STOP	C3 09 C3 E1	Increase IP by 9 if S flag is set Decrease IP by 31 if S flag is set
JNS	START	C4 04 C4 E0	Increase IP by 4 if S flag not set Decrease IP by 32 if S flag not set
JO	REPEAT	C5 09 C5 DF	Increase IP by 9 if O flag is set Decrease IP by 33 if O flag is set
JNO	AGAIN	C6 04 C6 FB	Increase IP by 4 if O flag not set Decrease IP by 5 if O flag not set

Procedures and Interrupts. Flags NOT set.

CALL, RET, INT and IRET are available only in the registered version.

Assembler		Machine Code	Explanation
CALL	30	CA 30	Save IP on stack and jump to 30.
RET		CB	Get IP from stack and jump to it.
INT	01	CC 01	Run code starting at address in 01 and save return address on the stack.
IRET		CD	Return from interrupt and continue from address saved on the stack.

Stack Manipulation Instructions. Flags NOT set.

Assembler		Machine Code	Explanation
PUSH	BL	E0 01	BL Saved onto the stack
POP	CL	E1 02	CL Restored off the stack
PUSHF		EA	Flags in status register saved
POPF		EB	SR restored from the stack

Input Output Instructions. Flags NOT set.

Assembler		Machine Code	Explanation
IN	0C	F0 0C	Data input from I/O port 0C to AL
OUT	0E	F1 0E	Data output from AL to I/O port 0E

Miscellaneous Instructions. CLI and STI set I flag.

Assembler		Machine Code	Explanation
CLO		FE	Close all visible peripheral windows.
HALT		00	Halt the processor clock
NOP		FF	Do nothing for one clock cycle
STI		FC	Set Interrupts Flag - Enabled
CLI		FD	Clear Interrupts Flag - Disabled
ORG	30	None	Code origin. Start generating code from address 30.
DB	"HELLO"	Define Bytes	Load ASCII codes of HELLO into RAM
DB	0D	Define Byte	Load hexadecimal number into RAM

TD1 Assembleur : premiers pas

P. Carreno - P. Portejoie

Prénom Nom

Groupe

Travail préliminaire

Lisez attentivement la partie cours de ce poly jusqu'au chapitre 4.3 et assurez-vous d'en maîtriser la plupart des concepts (déjà vus en cours d'amphi sous une autre forme). N'hésitez pas à poser des questions à votre enseignant.

Exercice 1-1

Soit le programme donné ci-dessous :

```
ETIQ: CLO MOV AL,4 MOV BL,8 ADD AL,BL ; un exemple de ce qu'il ne faut pas faire !
END
```

- Observez le même programme donné ci-après. Il est présenté différemment mais conserve exactement la même sémantique. Notez bien que son écriture sous cette forme n'est pas indispensable mais fortement recommandée ; sa présentation est basée sur 4 champs :

étiquette: instruction opérandes ; commentaires

```
; un premier programme, bien présenté
; ici, par souci de clarté, vous devez définir son rôle
; Rôle : .....
; .....
; =====
ETIQ: ; pour ce programme cette étiquette ne sert à rien,
      ; elle n'est là que pour l'exemple...
      CLO ; comme dans tout langage de programmation les
      MOV AL,4 ; commentaires sont les bienvenus, à condition
      MOV BL,8 ; d'être pertinents
      ADD AL,BL ;
      END ; directive d'assemblage, pas une instruction
          ; à ne pas confondre avec HALT qui est une instruction
```

- Indiquez ce qu'il fait (son rôle).
- Donnez-en la traduction dans le langage d'assemblage du simulateur sms32 ; vous vous référerez au récapitulatif du jeu d'instructions donné dans ce poly (instruction set summary) ; remarquez que les instructions y sont présentées de façon informelle et non exhaustive (présentation par l'exemple, ce qui est vrai pour un registre ou un emplacement mémoire l'est aussi pour les autres).
- Simulez son exécution (faites-le tourner « à la main »).

- Regardez de plus près le jeu d'instructions en y distinguant en particulier les différentes catégories d'instructions.
- Observez ci-dessous le même programme ré-écrit de façon à ce que l'initialisation des données soit moins figée (données assimilable à des constantes). Notez bien que les adresses y sont données en hexa puisque toutes les informations (adresses et données) manipulées en machine sont en binaire. Distinguez bien **instructions** (utilisées par le processeur) de **directives d'assemblage** (utilisées par l'assembleur), également appelées **pseudo-instructions**.

```
; un deuxième programme, initialisation plus adaptable
; Rôle : exemple d'utilisation des directives d'assemblage
; =====
DEBUT: ; pour ce programme cette étiquette ne sert à rien
        ; elle n'est là que pour l'exemple...
        CLO ; comme dans tout langage de programmation les
        MOV AL,[20] ; commentaires sont les bienvenus, à condition
        MOV BL,[21] ; d'être pertinents
        ADD AL,BL ;
        ORG 20 ; directive d'assemblage, pas une instruction
        DB 4 ; directive d'assemblage, pas une instruction
        DB 8 ;
        END ; directive d'assemblage, pas une instruction
            ; à ne pas confondre avec HALT qui est une instruction
```

- Simulez l'assemblage de ce programme afin d'en donner la représentation en mémoire.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00																
10																
20																
30																
40																
50																

Exercice 1-2

- Ecrivez un programme qui effectue l'addition de deux valeurs contenues aux adresses décimales 100 et 101, et range le résultat à l'adresse décimale 102. Pour l'initialisation des données vous utiliserez les directives d'assemblage ORG et DB.
- Effectuez l'assemblage de ce programme et simulez son exécution.

Exercice 1-3

- Ecrivez un programme qui compare deux valeurs contenues aux adresses décimales 100 et 101, puis additionne 1 à la plus petite valeur. On ne cherchera pas à traiter avec précision l'égalité (dans ce cas incrément de la deuxième valeur). Pour l'initialisation des données vous utiliserez les directives d'assemblage ORG et DB.
- Effectuez l'assemblage de ce programme et simulez son exécution.

TP1 Assembleur : premiers pas

P. Carreno - P. Portejoie

Les TP se déroulent sous Windows.

Vous utiliserez le simulateur d'assemblage SMS32v50. Vous pouvez y accéder par le lecteur G: (pensez à créer un raccourci y conduisant depuis votre espace personnel sinon vous ne pourrez pas enregistrer vos travaux). Vous pouvez aussi télécharger le dossier :

G:\prof\1tin01\ASR\M2101\ASM/ (récupérable aussi sur Moodle)



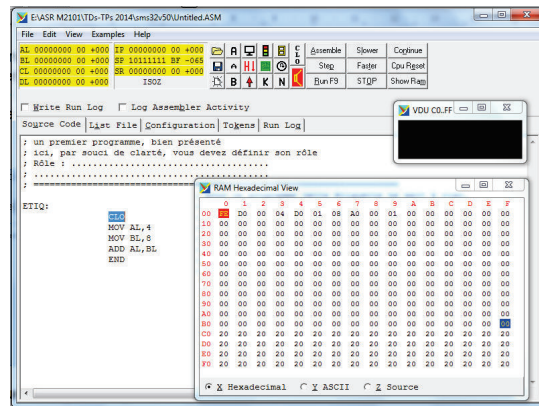
Pour chaque exercice votre compte-rendu devra comprendre une copie écran de votre code source assemblé (vous pourrez utiliser l'onglet `List File` du simulateur) et de son chargement en mémoire (format hexadécimal), ainsi qu'un compte-rendu d'exécution.

NB :

- pour les prochains Tps vos codes sources devront être commentés (pas de `List File`)
- les compte-rendus de TP doivent être rendus sur Moodle au plus tard le dimanche suivant le TP, avant 23h55

Travail préliminaire

Lancez le simulateur et observez-en les caractéristiques ; assurez-vous d'en comprendre la plupart des composants. N'hésitez pas à poser des questions à votre enseignant.



Exercice 1-1

- Reprenez la dernière version du programme de l'exercice 1-1 du TD et faites-le fonctionner ; observez bien le comportement des registres, en particulier IP. Vous vérifierez la conformité de l'assemblage fait manuellement en TP (onglet `ListFile`), ainsi que la conformité des résultats (observez la RAM ainsi que les registres AL et BL).

- Modifiez-le de façon à ce qu'il fasse une division, puis testez le cas d'une division par 0.

Exercice 1-2

Reprenez le programme de l'exercice 1-2 du TD et faites-le fonctionner. Vérifiez la conformité de l'assemblage fait manuellement en TD, ainsi que la conformité des résultats de la simulation (observez le contenu de la mémoire et des registres ; notez en particulier l'utilisation de la base 16 (pour ce faire utilisez par exemple ce jeu de test : 6 et 8).

Exercice 1-3

- Reprenez le programme de l'exercice 1-3 du TD et faites-le fonctionner. Vérifiez la conformité de l'assemblage fait manuellement en TD, surtout en ce qui concerne le calcul des sauts. Faites les tests d'exécution appropriés afin de comprendre le fonctionnement du programme et de pouvoir indiquer ci-dessous l'emplacement du bit S du registre d'état.

SR

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

- Faites un test d'exécution approprié afin de pouvoir indiquer ci-dessus l'emplacement du bit Z du registre d'état (vous indiquerez le test effectué).

Exercice 1-4

- Ecrivez un programme qui permet la saisie au clavier d'un caractère et qui l'affiche sur le terminal virtuel.

- Pour la saisie au clavier vous utiliserez l'instruction `IN n°port` (consultez le récapitulatif du jeu d'instructions), sachant que le port correspondant au clavier est 00
- Pour l'affichage du résultat vous aurez simplement à ranger la valeur à afficher en mémoire, à l'adresse C0

- Testez votre programme en utilisant d'autres valeurs à la place de C0 (par exemple E6 ou encore FF). Qu'observez-vous ? Concluez.

Exercice 1-5

- Modifiez le programme de l'exercice 1-3 précédent (incrément de 1) afin de pouvoir saisir les données au clavier (uniquement des chiffres, mais vous n'avez pas à en programmer la vérification) et afficher le résultat sur le terminal virtuel.

NB : vous aurez beau chercher, vous ne trouverez pas pour ce simulateur d'instruction de transfert entre registres (par exemple `MOV BL, AL`). Vous devrez pour cela utiliser la pile (voir cours). Notez bien que cela n'est pas dû à une limitation due à la simulation, mais bien un choix de construction.

- Testez le programme avec les jeux de données suivants : 4 et 8, puis 9 et 9. Décrivez ce que vous observez dans le dernier cas et donnez-en une explication. Proposez brièvement une solution, mais sans chercher à la coder, cela fera l'objet d'un TD-TP ultérieur.

TD2 Assembleur : boucles et ports

P. Carreno - P. Portejoie

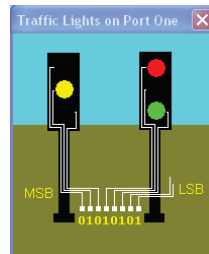
Prénom Nom

Groupe

Exercice 2-1

Observez le programme ci-dessous qui simule succinctement le fonctionnement de feux tricolores.

```
; Rôle : simule succinctement le fonctionnement de feux tricolores  
; =====  
;  
CLO  
Boucle:  
      MOV AL,0      ; Eteint toutes les lampes des feux tricolores  
      OUT 01        ; en mettant le code 00000000 dans AL  
                ; puis en envoyant AL au port correspondant  
  
      MOV AL,FC      ; De la même façon allume toutes les lampes  
      OUT 01        ; des feux tricolores (code FC)  
      JMP Boucle    ; Recommence la séquence précédente  
END
```



- Donnez-en la traduction en langage d'assemblage du simulateur sms32 ; vous vous référerez à l'annexe « Calcul des sauts » pour résoudre le branchement inconditionnel *JMP Boucle* (Aidez-vous de la table des références construite par l'assembleur)
- Complétez le programme de façon à ce qu'il s'arrête si l'utilisateur appuie sur la touche *Entrée* en fin de séquence (vous devrez faire en sorte qu'il soit sollicité).
NB : *Entrée* = *Enter* = *RC* (Retour Chariot) = *CR* (Carriage Return)
- Refaites-en l'assemblage en portant votre attention sur le calcul d'offset ; pensez à construire, au fur et à mesure, la table des références et à l'utiliser pour les résoudre. Comme vu précédemment vérifiez vos résultats.

**Exercice 2-2**

- Ecrivez un programme qui calcule dans AL, par pas de 1, les nombres entre une borne inférieure et une borne supérieure (à définir à l'assemblage par 2 constantes stockées dans le 31^{ème} et le 32^{ème} mot de la mémoire ; attention, réfléchissez bien à leur adresse).
NB : la borne supérieure peut être égale, mais en aucun cas dépassée.
- Exemple** : 2 et 5 ==> 2 3 4 5 doivent apparaître chronologiquement dans AL
- Vérifiez sa conformité en simulant son exécution (faites-le tourner « à la main »).
- Donnez-en la traduction en langage d'assemblage du simulateur sms32.

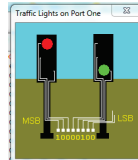
TP2 Assembleur : boucles et ports

P. Carreno - P. Portejoie

Exercice 2-1

- Reprenez le deuxième programme de l'exercice 2-1 du TD et faites-le fonctionner. Vérifiez la conformité des assemblages faits manuellement en TD.

- Complétez cette version de façon à ce que le programme simule le fonctionnement d'un carrefour à feux tricolores français. On donne pour cela l'information suivante concernant le paramétrage des feux (vous commencerez par compléter ce tableau) :



Rouge	Orange	Vert	Rouge	Orange	Vert	Inutilisé	Inutilisé	Hexadécimal	Feux	Temps
1	0	0	0	0	1	0	0	84	R-V	↓
									R-O	
									R-R	
									V-R	
									O-R	
									R-R	

Exercice 2-2

- Reprenez le programme de l'exercice 2-2 du TD et faites-le fonctionner. Vérifiez la conformité des assemblages faits manuellement en TD.
- Modifiez le programme afin de pouvoir en fixer le pas **par initialisation d'une constante en mémoire, à l'assemblage (DB)**. Testez-le avec 2 à 5, pas de 1 (==> observation de 2 3 4 5 dans AL), puis avec 2 à 9, pas de 3 (==> observation de 2 5 8 dans AL).
- NB** : à chaque phase de test observez bien le contenu de AL (c'est là que se construit le résultat) pour en vérifier la conformité avec l'énoncé (dépassement de la borne supérieure interdit). En cas de problème, passez à la question ci-après.
- Dans le cas du pas de 3, le problème de dépassement de la borne supérieure que vous avez dû observer est purement algorithmique : l'algorithme qui ne posait pas de problème par pas de 1 doit être modifié pour permettre un pas supérieur à 1. Pour ce faire il faut ajuster la borne supérieure avant la boucle par : **BL := BL - pas + 1** (hypothèse que BL reçoit RAM[1F] en début de programme).

Corrigez le programme et vérifiez à nouveau son fonctionnement dans les 2 cas.

Exercice 2-3

- Ecrivez un programme qui calcule le PGCD de 2 nombres initialement stockés en RAM aux adresses 100 et 101 et range le résultat à l'adresse 102. L'algorithme vous est donné ci-dessous ; aidez-vous également des exemples de traduction fournis en annexe.

Algorithme

```

TANT_QUE (A<>B)
    SI (A<B) ALORS
        B := B - A
    SINON
        A := A - B
    FINSI
FIN_TANT_QUE
PGCD := A ; ou PGCD := B

```

- Faites-le fonctionner (jeu de tests au choix).
- De façon identique à ce que vous avez fait en TD, justifiez le calcul d'offset pour 2 instructions de branchement de votre choix (un cas de déplacement positif et un cas de déplacement négatif).

TD3 Assembleur : indirection

P. Carreno - P. Portejoie

Prénom Nom
Groupe

Durant cette séance vous allez constater qu'il est courant et souvent indispensable de manipuler des adresses en assembleur (analogie aux pointeurs des langages de plus haut niveau).

Exercice 3-1

- Ecrivez un programme qui, dans un premier temps, compare deux valeurs contenues aux adresses décimales 100 et 101 et mémorise l'adresse de la plus petite. Dans un deuxième temps, la valeur dont l'adresse a été mémorisée est incrémentée de 1. Pour l'initialisation des données vous utiliserez les directives d'assemblage ORG et DB.
- Simulez son exécution.

Exercice 3-2

- Ecrivez un programme qui permet la saisie d'une chaîne de caractères (suite de caractères terminée par *Entrée*) en la rangeant en mémoire à partir de l'adresse décimale 100. Le code ASCII généré par la touche Entrée est un marqueur de fin et ne doit donc pas être stocké. Cependant, avant de se terminer, le programme ajoutera à la chaîne le caractère de fin de chaîne : 0 (octet nul).

Exercice 3-3

- Complétez le programme précédent afin qu'il effectue l'affichage de cette chaîne sur le terminal virtuel dans l'ordre (la condition d'arrêt sera la rencontre de l'octet nul) puis dans l'ordre inverse (la condition d'arrêt sera la reconnaissance de l'adresse du premier caractère).

Exercice 3-4

- Ecrivez un programme qui permet la saisie d'une chaîne de caractères terminée par *Entrée* puis procède à son affichage en ordre inverse. Cette fois-ci en utilisant la pile (à la différence de l'exercice 3-2) on évitera d'avoir à définir explicitement (et artificiellement) une zone de stockage de la chaîne en mémoire et on facilitera l'affichage en ordre inverse. Comme précédemment le caractère correspondant à *Entrée* ne sera pas stocké. En outre on prendra soin d'insérer 0 comme caractère de fin de chaîne, en fond de pile.

TP3 Assembleur : indirection

P. Carreno - P. Portejoie

Exercices 3-1 à 3-4

- Reprenez les exercices 3-1 à 3-4 du TD et faites-les fonctionner.

Exercice 3-5

- Modifiez le programme de saisie d'une chaîne de caractères de l'exercice 3-2 pour qu'il effectue la saisie à l'adresse décimale 192 et détermine si la chaîne saisie est un palindrome ou pas. La réponse affichée sera *OUI* ou *NON à la suite de la chaîne*.

Exemples : LAVAL et ELLE sont des palindromes.

Le résultat pour LAVAL sera : LAVAL OUI

Quelques petits conseils :

- vous pouvez fortement vous inspirer de ce que vous avez fait précédemment pour la saisie et l'affichage (code quasi ré-utilisable au prix de légères modifications) :
 - > saisie (+ mémorisation de la dernière adresse)
 - > palindrome ?
 - > Affichage selon résultat
- pour LAVAL la condition d'arrêt sur l'égalité des pointeurs est suffisante, pas pour ELLE ! Réfléchissez bien pour trouver la solution adaptée. Pensez à tester votre programme dans les 2 cas.
- enfin pour l'affichage vous considèrerez que le principe est unique et que seule change l'adresse d'implantation de la chaîne. Ainsi en sélectionnant le pointeur adéquat la chaîne correspondante sera affichée. Pour ce faire vous utiliserez en fin de code de votre programme :

```
ORG  A0
DB   "NON"
DB   0
DB   "OUI"
DB   0
END
```