LE LANGAGE MACHINE & L'ASSEMBLEUR

Ce chapitre constitue une étape importante, car c'est le premier qui traite du langage. Il décrit le plus rudimentaire des langages informatiques, le langage machine, ainsi que son *alter ego* humain, l'assembleur.

RECAPITULONS

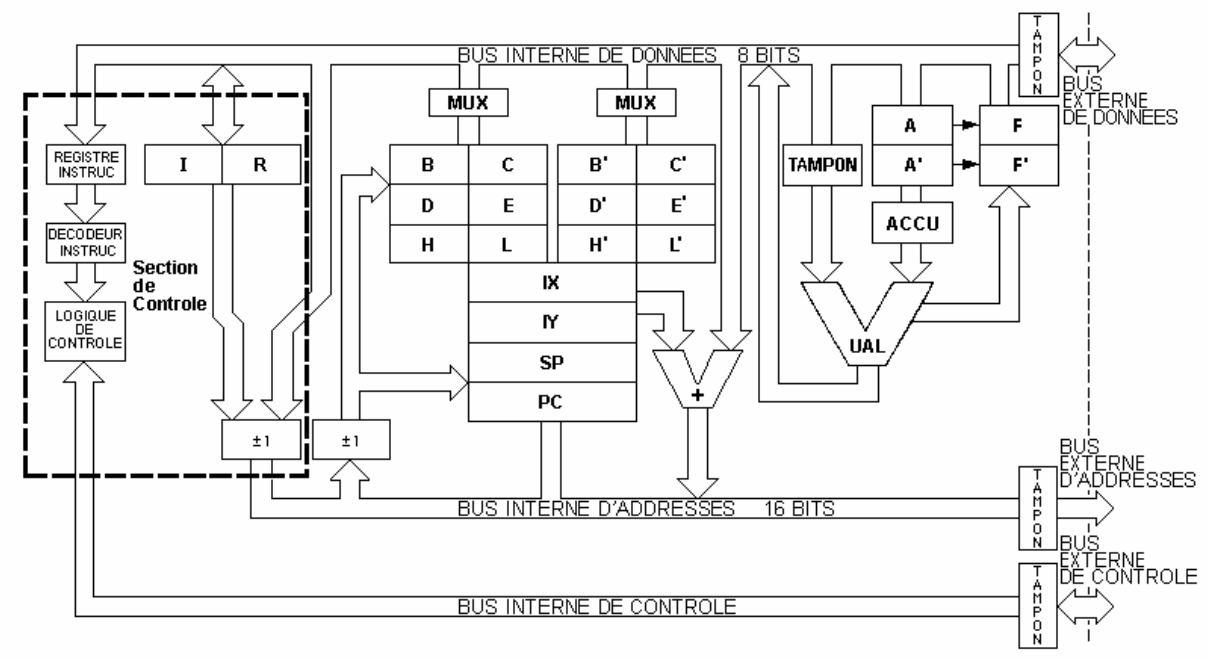
En fait, à la manière de Monsieur Jourdain, nous avons déjà utilisé ces deux langages, sans le savoir : le *langage machine* et l'*assembleur*. Si nous reprenons le tableau indiquant le contenu de la ROM1 dans le sous-chapitre *Optimisation de la ROM / 2 ROM* du chapitre précédent, le lien entre langage machine et l'assembleur est le suivant :

Langage machine	Assembleur
00h	NOP
01h	ADD Acc,M
02h	MOV Acc,D
03h	MOV D, Acc
04h	AND Acc,H
05h	etc…

Nota : NOP signifie *No OPeration* (pas d'opération). Cette instruction n'exécute aucune tâche, si ce n'est de passer à la suivante en réalisant uniquement le cycle fetch.
L'expression en langage machine d'une instruction en assembleur est la valeur à inscrire en RAM pour exécuter l'opération (c'est-à-dire le Code Opération), suivie éventuellement de ses opérandes aux adresses suivantes.
Le microprocesseur que nous avons construit présentait l'intérêt d'être simple, donc pédagogique. Pour aborder l'apprentissage du langage dans un monde plus réel, nous allons maintenant utiliser le microprocesseur réel Z80, dont les principes de fonctionnement sont les mêmes que pour notre microprocesseur fantôme, mais qui présente cependant quelques différences, certaines de pure convention (comme le nom donné aux registres), d'autres de fonctionnement (comme le fait que certains résultats d'opérations aillent directement dans un registre déterminé sans passer par Acc), mais dans ce dernier cas, il s'agit souvent de fonctions élaborées qui n'ont pas leur place dans un ouvrage dédié à une initiation.

LE Z80

Sa structure



Ses registres

Les registres du Z80 sont les suivants :

- accumulateur : a.
- registres de données 8 bits : b, c, d, e, h, l.
- registres de fonctionnement 8 bits :
  - f : drapeaux. Le contenu de ses 8 bits est SZXHXPNC où, suivant le résultat de l'opération précédente :
    - S : est à 1 pour un résultat négatif, à 0 sinon. Peu utilisé.
    - Z : est à 1 pour un résultat nul, à 0 sinon. Très utilisé. Attention à l'inversion apparente de logique (on aurait pu croire que Z soit à 0 pour un résultat à 0) : Z prend la valeur booléenne de l'assertion « le résultat est à Zéro », c'est-à-dire 1 si c'est vrai, 0 sinon.
    - X : inutilisés.
    - H : demi-retenue. Utilisé pour le codage BCD (*Binaire Codé Décimal*), que nous ne développerons pas.
    - P : appelé en réalité P/V (*Parity / oVerflow* = parité / débordement), modifié notamment suivant le nombre (pair ou impair) de bits à 1 du résultat. Très peu utilisé.
    - N : indique si la dernière opération était une addition ou une soustraction. Utilisé pour le codage BCD (*Binaire Codé Décimal*), que nous ne développerons pas.
    - C : est à 1 pour un résultat ayant généré une retenue, à 0 sinon. Très utilisé.
  - i : gestion des interruptions. Nous ne détaillerons pas son fonctionnement ici.
  - r : rafraîchissement de la mémoire. Nous ne détaillerons pas son fonctionnement ici.
- registres de données 16 bits :
  - bc, de, hl : registres issus du jumelage des registres 8 bits. Par exemple, si B=9Dh et C=A3h, alors BC=9DA3h. Inversement, si BC=9DA3h, alors B=9Dh et C=A3h.
  - ix et iy : purs registres 16 bits, dont le nom est propre et ne signifie en rien un jumelage des registres i et x d'une part, et i et y d'autre part. ix est utilisé via le préfixe *DDh*, précédant les opérations sur hl et permet de remplacer ce dernier par ix+n. Par exemple, le code assembleur *AEh* réalise l'opération *XOR (hl)*. Le code assembleur *DDh AEh A3h* réalise l'opération *XOR (ix+A3h)*. iy est utilisé de même que ix, mais avec le préfixe *FDh* au lieu de *DDh*.
- registres de fonctionnement 16 bits :
  - sp : Stack Pointer
  - pc : Program Counter
  - af : registre issu du jumelage des registres 8 bits a et f, que l'on retrouve dans quelques instructions sous cette forme. Ce n'est pas un registre ayant une valeur de donnée 16 bits, mais ce jumelage permet de manipuler en une fois tout ce qui touche à un résultat : sa valeur (dans a) et son effet sur les drapeaux (dans f).

Tous les registres 8 bits (sauf r et i qui sont peu utiles au programmeur) ont un registre-doublon (portant le même nom, mais avec une apostrophe : a', b', c', d', e', h', l', f'), auxquels on accède par le biais d'instructions d'échange entre le registre et son doublon.

Ses instructions

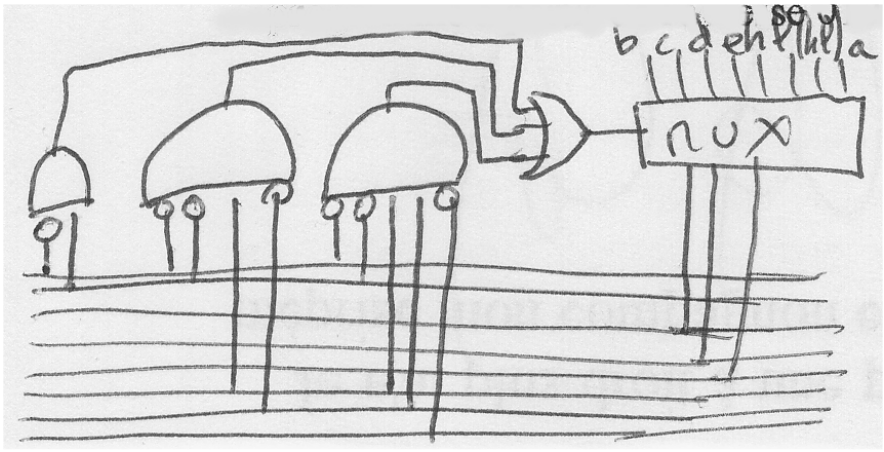
Avec le Z80, les instructions (notamment leurs codes assembleur associés) ne sont plus celles que nous avons élaborées dans le précédent chapitre, mais celles présentées en annexe 1, néanmoins très voisines.
Il existe cinq « jeux » (pour reprendre le vocabulaire du chapitre « Le microprocesseur ») d'instructions :

- colonne « Seul » de l'annexe 1 : jeu de base, comprenant notamment les préfixes CBh, DDh, EDh et FDh qui ouvrent les autres jeux.
- colonne « Après CBh » de l'annexe 1 : instruction exécutées lorsque le Code Opération est précédé de CBh en RAM.
- colonne « Après EDh » de l'annexe 1 : instruction exécutées lorsque le Code Opération est précédé de EDh en RAM.
- colonne « Après DDh » de l'annexe 1 : ce pseudo-jeu qui reprend en fait le 1er jeu d'instructions, en permettant un adressage indexé dans les instructions utilisant l'opérande « hl ». Avec ce préfixe, l'opérande « hl » du premier jeu d'instruction est remplacé par « ix », ou « ix+n » suivant le cas. A ce titre, il ne s'agit pas strictement d'un jeu supplémentaire, car il n'apporte pas de nouvelles instructions : il ne fait que modifier l'une des opérandes d'instructions déjà vues. Le préfixe DDh peut être cumulé avec le préfixe CBh (colonne « Après DDh CBh » de l'annexe 1).
- Un 5ème pseudo-jeu, qui reprend les mêmes instructions que le précédent, mais utilise le registre d'indexation iy au lieu de ix. Cela est réalisé en utilisant le préfixe FDh au lieu de DDh.

La fonction des diverses instructions est décrite en annexe 2.
La liste des instructions peut paraître complètement désordonnée, arbitraire. A y regarder de plus près, on remarque par exemple que les Codes Opération de 40h à 7Fh (c'est-à-dire sur une liste de 40h – qui est un chiffre rond en hexadécimal – Codes Opération) ne tombent pas du ciel tout cuits, mais semblent provenir d'une construction volontaire : la première opérande passe en revue les valeurs *b*, *c*, *d*, *e*, *h*, *l*, *(hl)* et *a*, puis pour chacun d'eux la deuxième opérande aussi.
En effet, une analyse plus fine de l'écriture binaire des Codes Opération du jeu de base révèle la structure suivante :

Code Opération (binaire) (un point signifie : 0 ou 1, en gras : chiffres changeant d'une ligne à la suivante)			Opérateur (points = opérateur et / ou opérande)	Opérateur et/ou opérande(s), selon la valeur de « … », de « .. » ou de « . »							
				000 00 0	001 01 1	010 10	011 11	100	101	110	111
00……	00…**000**	00**0**..000	Divers	NOP	EX af,af'	DJNZ n	JR n				
		00**1**..001	JR .. n	nz	z	nc	c				
	00…**001**	00..**0**001	LD ..,nn	bc	de	hl	sp				
		00..**1**001	ADD hl,..	bc	de	hl	sp				
	00…**010**	00..**0**010	LD ..	bc,a	de,a	(nn),hl	(nn),a				
		00..**1**010	LD ..	a,bc	a,de	hl,(nn)	a,(nn)				
	00…**011**	00..**0**011	INC ..	bc	de	hl	sp				
		00..**1**011	DEC ..	bc	de	hl	sp				
	00…**100**		INC …	b	c	d	e	h	l	(hl)	a
	00…**101**		DEC …	b	c	d	e	h	l	(hl)	a
	00…**110**		LD …,n	b	c	d	e	h	l	(hl)	a
	00…**111**	00**0**..111	Rotations	RLCA	RRCA	RLA	RRA				
		00**1**..111	Divers	DAA	CPL	SCF	CCF				
01……	01**xxxyyy** sauf 01**110110**		LD xxx,yyy où xxx = yyy =	 b b	 c c	 d d	 e e	 h h	 l l	 (hl) (hl)	 a a
	01**110110**		HALT								
10……	10**cccxxx**		CCC a,xxx où CCC = xxx =	 ADD b	 ADC c	 SUB d	 SBC e	 AND h	 XOR l	 OR (hl)	 CP a
11……	11…**000**		RET …	nz	z	nc	c	po	pe	p	m
	11…**001**	00..**0**001	POP ..	bc	de	hl	af				
		00..**1**001	Divers	RET	EXX	JP (hl)	LD sp,hl				
	11…**010**		JP … nn	nz	z	nc	c	po	pe	p	m
	11…**011**	00**000**011	JP nn								
		00**001**011	Préfixe CBh								
		00**01**.011	IN / OUT	OUT n,a	IN a,n						
		00**10**.011	EX .	(sp),hl	de,hl						
		00**11**.011	Interruptions	DI	EI						
	11…**100**		CALL … nn	nz	z	nc	c	po	pe	p	m
	11…**101**	00..**0**101	PUSH ..	bc	de	hl	af				
		00..**1**101 sauf 00**001**101	Préfixes	DDh	EDh	FDh					
		00**001**101	CALL nn								
	11…**110**		CCC a,n où CCC =	 ADD	 ADC	 SUB	 SBC	 AND	 XOR	 OR	 CP
	11…**111**		RST …	00h	08h	10h	18h	20h	28h	30h	38h

Par exemple, le Code Opération 10100010 signifie *AND a,d*, le Code Opération 11011010 signifie *JP c nn*. On distingue bien par exemple que les bits 3 à 5 du Code Opération peuvent être traités via un multiplexeur, situé non pas sur les bits de données du microprogramme, mais directement sur les bits du Code Opération. On peut supposer que ce multiplexeur est par contre activé par un bit du microprogramme, ou par une logique combinatoire sur les autres bits du Code Opération.

#Schéma ? 

MISE EN JAMBES

Premier exemple

Nous allons écrire un petit programme qui additionne 26h et 1Eh, et met le résultat dans *d*. Nous plaçons notre programme à partir de l'adresse 0100h de la RAM.

Ce qu'il y a dans la machine		Ce que l'humain peut raisonnablement comprendre	
Adresse (hexa)	Code (hexa)	Mnémonique	Commentaire
0100 0101	3E 26	LD a,26h	Octet du Code Opération Octet de l'opérande
0102 0103	16 1E	LD d,1Eh	Octet du Code Opération Octet de l'opérande
0104	82	ADD a,d	Octet du Code Opération (pas d'opérande à expliciter)
0105	57	LD d,a	Octet du Code Opération (pas d'opérande à expliciter)
0106	C9	RET	Octet du Code Opération (pas d'opérande)

La dernière opération spécifie que c'est la fin du programme. Nous y reviendrons plus tard.
Le langage machine est ce qui est directement utilisable par la machine : c'est une suite de nombres dans la RAM (celle de la deuxième colonne), suite ayant certes une signification précise pour le microprocesseur, mais très peu parlante pour l'humain moyen : il ne saute pas aux yeux que la suite hexadécimale 3E / 26 / 16 / 1E / 82 / 57 / C9 réalise une addition. Pour qu'un programme en langage machine devienne présentable, l'humain associe à chacun des Codes Opération un court texte qui résume sa fonction et remplace avantageusement l'hermétique Code Opération hexadécimal : c'est le mnémonique, que nous avons déjà vu. Celui-ci se décompose en une opération, et éventuellement une ou des opérandes.
Ainsi, le Code Opération 3Eh signifie : prendre l'octet suivant dans la RAM et l'écrire dans *a*. Le mnémonique associé est *LD a,n* (*n* signifie : « valeur numérique sur 1 octet, précisée en opérande en adressage immédiat », ce qui est logique puisqu'il s'agit de la copier dans *a*, qui tient sur 1 octet). Reste à inscrire la valeur de *n* (l'opérande) dans l'adresse immédiatement supérieure à celle qui contient le Code Opération 3Eh (puisque c'est là que le microprogramme de l'instruction l'attend) : c'est ce que nous avons fait à l'adresse 0101h. Si cet octet contient 26h, le mnémonique spécifique à ce cas-ci devient *LD a,26h*. L'humain pense ses programmes en assembleur, puis, une fois écrit, le programme doit nécessairement être traduit en langage machine. Cette opération s'appelle l'« assemblage » (l'opération inverse étant le « désassemblage »). Pour bien la comprendre, nous la réalisons ici à la main (grâce aux correspondances Code Opération / mnémonique indiquées dans l'annexe 1), mais c'est une opération aujourd'hui complètement automatisée.
La prochaine instruction doit donc prendre place à l'adresse 0102h.
Le Code Opération 82h signifie : additionner *a* et *d*, puis mettre le résultat dans *a*. Le mnémonique associé est *ADD a,d*. Ce Code Opération se suffit à lui-même, car l'opérande est implicitement comprise dans le Code Opération. L'adresse suivante en RAM contient donc non pas une opérande, mais la prochaine instruction.

Le tableau ci-dessus est plus souvent représenté de la manière suivante :

Adresse (hexa)	Code (hexa)	Mnémonique	Commentaire
0100	3E 26	LD a,26h	
0102	16 1E	LD d,1Eh	
0104	82	ADD a,d	
0105	57	LD d,a	
0106	C9	RET	

Deuxième exemple

Par de simples additions successives, réalisons maintenant la multiplication de 1Dh par 05h.

Adresse (hexa)	Code (hexa)	Mnémonique	Commentaire
0100	3E 00	LD a,00h	Initialisation de *a* à 0
0102	16 1D	LD d,1Dh	Multiplicande dans *d*
0104	06 05	LD b,05h	Multiplicateur dans *b*
0106	82	ADD a,d	Calcul du résultat partiel
0107	05	DEC b	Décrémentation du multiplicateur, avec modification des drapeaux : notamment, z est mis à 0 si b reste différent de 0, mais est mis à 1 si b atteint 0 après la décrémentation.
0108	C2 06 10	JP nz 0106h	Saut à l'adresse 0106h si NZ est vrai, c'est-à-dire si Z est faux, c'est-à-dire si le résultat de la dernière opération n'est pas 0, c'est-à-dire si $b \neq 0$. Comme pour toute valeur 16 bits, l'adresse de saut est inscrite en RAM en mettant l'octet de poids faible avant l'octet de poids fort, ici 06h à l'adresse 0109h, et 10h à l'adresse 010Ah.
010B	57	LD d,a	Enregistrement du résultat dans *d*
010C	C9	RET	Fin

Voici le principe de ce petit programme :

- *a* est destiné à contenir le résultat intermédiaire. Après l'avoir au préalable initialisé à 0, on va, à 05h reprises, additionner 1Dh à *a*.
- *d* contient la multiplicande (la valeur à additionner à *a*)
- *b* contient le multiplicateur, vu comme un compteur (combien de fois a-t-on additionné 1Dh à *a* ?)
- Après avoir additionné *d* à *a*, on décrémente *b*. Si alors *b*>0, cela signifie que nous n'avons pas additionné *d* à *a* suffisamment de fois. Il faut donc répéter l'opération d'addition. C'est le rôle tenu par l'instruction *JP nz 0106h* : si le résultat de la décrémentation de *b* n'est pas nul, on réalise un saut à l'adresse 0106h, ce qui répète le cycle d'addition / décrémentation / saut éventuel. Sinon, on passe à l'instruction suivante dans la RAM, ce qui clôt le cycle.
- Si l'opération est terminée, on transfère le résultat final de *a* vers *d*, pour libérer *a*, qui nous resservira certainement plus tard.

L'ensemble des instructions des adresses 0106h à 010Ah est appelé *boucle* (représentée par la flèche) : un compteur (ici le registre *b*) détermine le nombre de fois où cet ensemble d'instructions doit être exécuté. L'adresse indiquée par l'instruction *JP nz* détermine où la boucle commence. L'adresse de l'instruction *JP nz* détermine où la boucle se termine. Les boucles sont très souvent utilisées dans tous les langages de programmation. Tellement même que le Z80 est doté d'une instruction assembleur spécifique : *DJNZ dis* (*dis* étant l'opérande).
DJNZ dis réalise les opérations suivantes :

- DEC b
- Si Z=1 (c'est-à-dire si *DEC b* a donné un résultat nul, c'est-à-dire si *b*=0), alors *JR dis*. Sinon, le programme continue à l'instruction suivante. Notez que le saut est relatif (JR et non JP).

Le registre b est utilisé comme compteur de la boucle. Il est censé être initialisé à la valeur voulue en amont de la boucle. La construction typique d'une boucle est la suivante :

Adresse	Code (hexa)	Mnémonique	Commentaire
Prog		Diverses opérations…	Comprend notamment les opérations préparatoires à la boucle.
	06 compteur	LD b,compteur	*compteur* est le nombre de fois où la boucle doit être parcourue.
Boucle		Première opération à répéter	Début de la boucle.
		Autres opérations à répéter…	
Fin	10 Boucle-Fin-2	DJNZ Boucle	Le « -2 » provient du fait qu'il s'agit d'un décalage par rapport à PC qui a déjà été mis à jour, par le microprogramme de DJNZ, pour pointer sur l'instruction suivante. Or, *DJNZ dis* occupant 2 octets en RAM, l'instruction suivante se situe à l'adresse Fin+2. PC a donc pour valeur Fin+2, d'où le « -2 » pour compenser cette anticipation.
Fin+2		Reste du programme…	Sortie de la boucle.

Dans l'exemple précédent (multiplication de 1Dh par 05h), le programme serait :

Adresse (hexa)	Code (hexa)	Mnémonique	Commentaire
0100	3E 00	LD a,00h	Préparations
0102	16 1D	LD d,1Dh	
0104	06 05	LD b,05h	Initialisation du compteur
0106	82	ADD a,d	Opération à répéter
0107	C2 FD	DJNZ –3d	Décrémentation de b, et, si b≠0, saut relatif de –3d car : 0109h (adresse de l'instruction suivante) -3d = 0106h, et -3d = FDh.
0109	57	LD d,a	Sortie de la boucle
010A	C9	RET	Fin

Importance de l'adresse de départ

Reprenons le premier programme de ce chapitre :

Adresse (hexa)	Code (hexa)	Mnémonique	Commentaire
0100	3E 26	LD a,26h	
0102	16 1E	LD d,1Eh	
0104	82	ADD a,d	
0105	57	LD d,a	
0106	C9	RET	

Supposons que ce programme soit exécuté non pas par CALL 0100h, mais par CALL 0101h.

- Le contenu de l'adresse 0101h est alors interprété comme une instruction. 26h est alors interprété comme *LD h,n*. Le *n* est pris sur l'adresse suivante (0102h), qui contient 16h. Le Z80 réalise alors *LD h,16h*.
- La prochaine instruction est donc à l'adresse 0103h : 1Eh réalise l'opération *LD e,n*. En l'occurrence, puisque 0104h contient 82h, le Z80 réalise *LD e,82h*.
- La prochaine instruction est donc à l'adresse 0105h : 57h réalise l'opération *LD d,a*.
- La prochaine instruction est donc à l'adresse 0106h : C9h réalise l'opération *RET*.

Les mêmes codes opération, vus à partir d'une adresse légèrement décalée, réalisent donc le programme suivant, qui n'a rien à voir avec le précédent et ne sert d'ailleurs à rien :

Adresse (hexa)	Code (hexa)	Mnémonique	Commentaire
0101	26 16	LD h,16h	
0103	1E 82	LD e,82h	
0105	57	LD d,a	
0106	C9	RET	

On voit donc deux choses :

- Un octet lambda en RAM n'est pas *a priori* « opérateur » ou « opérande », « programme » ou « donnée ». Ce n'est que l'adresse à partir de laquelle un programme est appelé qui détermine l'état « opérateur » ou « opérande » des octets qui suivent.
- Le décalage d'un seul octet fait perdre toute signification à un programme.

DESSINONS

Affichage d'un pixel sur l'écran

Voyons maintenant, de manière simplifiée, comment éclairer sur l'écran un pixel donné. Supposons que nous ayons une carte graphique ayant les caractéristiques suivantes :

- définition graphique = 256 pixels horizontaux × 192 pixels verticaux en 256 couleurs.
- le "plan" de l'écran se situe directement en RAM, à partir de l'adresse 1000h (par exemple). Il commence en haut à gauche de l'écran.
- Dans cette plage de la RAM, le contenu de chaque adresse spécifie la couleur à afficher, chacune des 256 couleurs possibles étant définie au sein d'une palette (qui précise les 3 composantes RGB associées à chaque code couleur). Par convention, nous supposerons que, dans cette palette, la couleur 0d soit le noir (chaque composante RGB à 0), la couleur 1d soit le blanc (chaque composante RGB à 255d). Voici un exemple de palette (ici stockée à une adresse que nous appelons PALETTE) :

Code couleur (hexa)	Adresse	Composante rouge (décimal)	Composante vert (décimal)	Composante bleu (décimal)	Couleur obtenue
00	PALETTE	0	0	0	noir
01	PALETTE + 01h×3	255	255	255	blanc
02	PALETTE + 02h×3	64	64	64	gris foncé
03	PALETTE + 03h×3	127	127	127	gris
04	PALETTE + 04h×3	192	192	192	gris clair
05	PALETTE + 05h×3	255	0	0	rouge
06	PALETTE + 06h×3	255	0	255	rose
07	PALETTE + 07h×3	127	0	127	violet
08	PALETTE + 08h×3	255	255	0	jaune
09	PALETTE + 09h×3	0	255	255	bleu ciel
0A	PALETTE + 0Ah×3	255	127	0	orange
…	…	…	…	…	…
FF	PALETTE + FFh×3	114	73	10	brun

Le tableau suivant récapitule l'adresse hexadécimale de chaque pixel (en 256 couleurs, 1 pixel = 1 octet, donc 1 pixel = 1 adresse) en fonction de sa position sur l'écran (imaginez que ce tableau soit posé sur l'écran) :

	Gauche de l'écran				Droite de l'écran
Haut de l'écran	1000	1001	…	10FE	10FF
	1100	1101	…	11FE	11FF
	…	…	…	…	…
	CE00	CE01	…	CEFE	CEFF
Bas de l'écran	CF00	CF01	…	CFFE	CFFF

Sur un pixel donné

Pour colorer un pixel, il suffit d'inscrire la valeur de la couleur désirée dans son adresse en RAM. Par exemple, pour mettre en blanc le 3ème pixel en partant de la gauche de la 2ème ligne en partant du haut (leur principe étant désormais bien compris – si, si ! -, les « codes opération » ne sont plus affichés) :

Adresse (hexa)	Mnémonique	Commentaire
0000	LD HL,1102h	Met l'adresse du pixel à modifier dans HL
0003	LD (HL), 1d	Inscrit le code couleur du blanc dans la case mémoire d'adresse HL
0005	RET	

Pour « éteindre » ce pixel, il suffit de lui donner la même couleur que celle de ses voisins. En supposant qu'ils soient noirs :

Adresse (hexa)	Mnémonique	Commentaire
0000	LD HL,1102h	Met l'adresse du pixel à modifier dans HL
0003	LD (HL), 0d	Inscrit le code couleur du noir dans la case mémoire d'adresse HL
0005	RET	

Sur un pixel quelconque

Les sous-programmes précédents ne seront pas souvent utiles : seulement quand on veut blanchir ou noircir le 3ème pixel en partant de la gauche de la 2ème ligne. Il serait plus pratique d'écrire un sous-programme allant inscrire le code couleur donné C à l'adresse correspondant au Xème pixel en partant de la gauche de la Yème ligne (X pouvant être choisi entre 0 et 255 compris, Y entre 0 et 191 compris). Voici comment faire :

- On fixe à trois octets de la RAM la fonction suivante :
  - Le premier (choisissons l'octet d'adresse D000h, juste après la RAM vidéo) est destiné à contenir le code couleur C
  - Le second (D001h) est destiné à contenir la valeur X
  - Le troisième (D002h) est destiné à contenir la valeur Y.
- A partir du quatrième octet, un sous-programme calcule l'adresse correspondant à cet octet (1000h+X+256d×Y), et y inscrit C.

Voici le sous-programme :

Adresse (hexa)	Mnémonique	Commentaire
D000	NOP	Cet octet est réservé à la donnée C
D001	NOP	Cet octet est réservé à la donnée X
D002	NOP	Cet octet est réservé à la donnée Y
D003	PUSH hl	Ce sous-programme allant utiliser le registre *hl* et ne sachant pas si le programme appelant l'utilise, on le sauvegarde dans la pile, pour le récupérer avant le retour au programme appelant.
D004	LD a,(D002h)	Charge Y dans *a*
D007	ADD a,10h	Ajoute 10h à *a*
D009	LD h,a	Charge *a* (c'est-à-dire Y+10h) dans *h*. Ainsi, *h* ayant un poids de 256d, *hl* contient bien (10h+Y)×100h = 1000h+256d×Y
D00A	LD a,(D001h)	Charge X dans *a*
D00D	LD l,a	Charge *a* (c'est-à-dire X) dans *l*. Ainsi, *hl* contient bien 1000h+X+256d×Y
D00E	LD a,(D000h)	Charge C dans *a*
D011	LD (hl),a	Inscrit *a* dans l'octet d'adresse *hl*, c'est-à-dire C en X-Y.
D012	POP hl	On récupère la valeur qu'avait *hl* lors de l'appel par le programme appelant. Ainsi, pour ce dernier, la valeur de *hl* n'a pas changé (peut-être l'utilise-t-il). Il n'y a pas d'incidence sur la commande RET suivante, car le PUSH/POP est réalisé dans le même sous-programme.
D013	RET	

En maintenant ce sous-programme en mémoire, le travail consistant à blanchir le 3ème pixel en partant de la gauche de la 2ème ligne consisterait en :

Adresse (hexa)	Mnémonique	Commentaire
0000	LD a,1d	Charge la couleur dans *a*
0002	LD (D000h),a	L'inscrit en D000h
0005	LD a,2d	Charge X dans *a* (X part de 0 pour le 1er pixel)
0007	LD (D001h),a	L'inscrit en D000h
000A	LD a,1d	Charge Y dans *a* (Y part de 0 pour la 1ère ligne)
000C	LD (D002h),a	L'inscrit en D000h
000F	CALL D003h	Appelle le sous-programme. C'est le fait que l'adresse d'appel soit D003h (et non D000h) qui fait que les octets D000h à D002h sont des données, et non du programme : CALL D000h n'a aucun sens.
0012	RET	Grâce au PUSH hl / POP hl du sous-programme, la valeur de *hl* est conservée, ce qui serait obligatoire si le présent programme l'utilisait.

Cette méthode n'a pas le mérite d'être plus courte, mais de permettre de paramétrer entièrement quel pixel on souhaite colorer, et en quelle couleur, ce que nous allons utiliser à outrance.
Pour des raisons mnémotechniques, nous appellerons désormais PIXEL l'adresse D003h. Il s'agit d'une simple convention entre nous : lorsque nous verrons l'adresse PIXEL, il faudra comprendre D003h. De même, lorsque nous verrons l'adresse PIXEL-3, il faudra comprendre D000h, etc… Les programmes d'assemblage gèrent très bien ce type de convention.

Affichage d'un motif

Supposons que nous voulions afficher une petite voiture en X-Y. Voici la voiture (il suffit d'y croire : en regardant le schéma à 10 mètres de distance, on reconnaît presque une affreuse limousine américaine…) :

	Origine	Colonne +1	Colonne +2	Colonne +3	Colonne +4	Colonne +5	Colonne +6
Origine				1	2	3	
Ligne+1			4	5	6	7	
Ligne+2	8	9	10	11	12	13	14
Ligne+3	15	16	17	18	19	20	21
Ligne+4		22				23	

Chaque pixel est arbitrairement identifié par un numéro. Récapitulons la position de chaque pixel à traiter à partir de l'origine définie qui sera la coordonnée X-Y de la voiture (le gris clair correspondant au code couleur 4d dans notre palette) :

N° pixel	Décalage horizontal	Décalage vertical	Couleur
1	3	0	0
2	4	0	0
3	5	0	0
4	2	1	0
5	3	1	4
6	4	1	4
7	5	1	0
8	0	2	0
9	1	2	0
10	2	2	0
11	3	2	0
12	4	2	0
13	5	2	0
14	6	2	0
15	0	3	0
16	1	3	0
17	2	3	0
18	3	3	0
19	4	3	0
20	5	3	0
21	6	3	0
22	1	4	0
23	5	4	0

L'idée est de

- lire les décalages horizontal et vertical du point 1
- les ajouter respectivement à X et Y
- lire sa couleur
- appeler PIXEL
- recommencer avec le point suivant
- et ainsi de suite jusqu'au dernier…

Il faut savoir combien de point il faut traiter. Pour cela, nous allons inscrire ces données à partir d'une adresse (choisissons D020h) que nous appellerons VOITURE, de la même manière que nous avions défini PIXEL. La mémoire contiendra donc les données suivantes :

Adresse	Libellé du contenu	Contenu
VOITURE	Nombre de points à traiter	23
VOITURE + 1	Décalage horizontal du point 1	3
VOITURE + 2	Décalage vertical du point 1	0
VOITURE + 3	Couleur du point 1	0
VOITURE + 4	Décalage horizontal du point 2	4

VOITURE + 5	Décalage vertical du point 2	0
VOITURE + 6	Couleur du point 2	0
VOITURE + 7	Décalage horizontal du point 3	5
VOITURE + 8	Décalage vertical du point 3	0
VOITURE + 9	Couleur du point 3	0
VOITURE + 10	Décalage horizontal du point 4	2
VOITURE + 11	Décalage vertical du point 4	1
VOITURE + 12	Couleur du point 4	0
VOITURE + 13	Décalage horizontal du point 5	3
VOITURE + 14	Décalage vertical du point 5	1
VOITURE + 15	Couleur du point 5	4
…	…	…
VOITURE + 67	Décalage horizontal du point 23	5
VOITURE + 68	Décalage vertical du point 1	4
VOITURE + 69	Couleur du point 1	0

Ainsi, le petit programme suivant va afficher n'importe quel motif enregistré sous cette forme, à n'importe quel endroit (de même que précédemment, nous nommerons l'adresse de départ MOTIF) :

Adresse (hexa)	Mnémonique	Commentaire	
MOTIF-4	NOP	Sert à inscrire la valeur de X où il faut dessiner le motif	Données
MOTIF-3	NOP	Sert à inscrire la valeur de Y où il faut dessiner le motif	
MOTIF-2	NOP	Sert à inscrire l'octet de poids fort de l'adresse du motif à dessiner (ici, le programme appelant inscrira l'octet de poids fort de VOITURE)	
MOTIF-1	NOP	Sert à inscrire l'octet de poids faible de l'adresse du motif à dessiner	
MOTIF	PUSH hl	Ce sous-programme utilisant *hl*, il sauvegarde *hl* pour le programme appelant	Sauvegarde des registres
	PUSH bc	Idem pour *bc* (le registre *bc* n'est pas utilisé en tant que tel, mais *b* est utilisé, et il n'existe pas d'instruction *PUSH b* : qui peut le plus peut le moins)	
	PUSH de	Idem pour *de*	
	LD hl,(MOTIF-2)	Inscrit dans *hl* l'adresse VOITURE du motif à dessiner	Détermination du nombre de points
	LD a,(hl)	Inscrit dans *a* le contenu de l'adresse VOITURE, c'est-à-dire nombre de points à traiter	
	LD b,a	L'enregistre dans *b* (compteur de DJNZ)	

Adresse (hexa)	Mnémonique	Commentaire	
BOUCLE1	INC hl	Incrémente *hl*, pour passer à l'adresse suivante (décalage horizontal)	Détermination de l'abscisse du point en cours dans le motif
	LD a,(MOTIF-4)	Inscrit X dans *a*	
	LD d,a	Enregistre dans *d*	
	LD a,(hl)	Inscrit le décalage dans *a*	
	ADD a,d	Somme X et le décalage	
	LD (PIXEL-2),a	Inscrit le résultat dans le X de PIXEL	
	INC hl	Incrémente *hl*, pour passer à l'adresse suivante (décalage vertical)	Détermination de l'ordonnée du point en cours dans le motif
	LD a,(MOTIF-3)	Inscrit Y dans *a*	
	LD d,a	Enregistre dans *d*	
	LD a,(hl)	Inscrit le décalage dans *a*	
	ADD a,d	Somme Y et le décalage	
	LD (PIXEL-1),a	Inscrit le résultat dans le Y de PIXEL	
	INC hl	Incrémente *hl*, pour passer à l'adresse suivante (couleur)	Détermination de la couleur du point en cours dans le motif
	LD a,(hl)	Inscrit la couleur dans *a*	
	LD (PIXEL-1),a	Inscrit la couleur dans le C de PIXEL	
	CALL PIXEL	Dessine le point en cours	
	DJNZ BOUCLE1	Passe au point suivant, jusqu'au dernier (déterminé par la valeur de *b*)	Prochain pixel
	POP de		Restaure les registres pour le programme appelant
	POP bc		
	POP hl		
	RET		

Le programme suivant permet d'exploiter ce sous-programme. Pour dessiner la voiture en X=45d, Y=68d :

Adresse (hexa)	Mnémonique	Commentaire
0000	LD a, 45d	Vise l'adresse MOTIF-4
0002	LD (MOTIF-4),a	Pour y inscrire l'abscisse du motif
0005	LD a, 68d	Vise l'adresse MOTIF-3
0007	LD (MOTIF-3),a	Pour y inscrire l'ordonnée du motif
000A	LD hl,VOITURE	Vise l'adresse MOTIF-2
000D	LD (MOTIF-2),hl	Pour y inscrire l'adresse du motif
0010	CALL MOTIF	Dessine le motif
0013	RET	Fin

Animation

Essai

Dessiner une voiture, même moche, c'est bien, mais la faire avancer c'est mieux ! Pour cela, nous allons appliquer le principe du cinéma : afficher successivement des images décalées de la voiture, afin de donner l'illusion de mouvement… Il s'agit de la dessiner à des endroits successifs et voisins, par exemple : X=45d Y=68d, puis X=44d Y=68d, puis X=43d Y=68d, puis X=42d Y=68d, puis X=41d Y=68d, puis X=40d Y=68d, etc… Ceci n'est qu'une approximation, car une telle opération présenterait le résultat suivant (détail de l'écran) :

Dessin en X=45d Y=68d :

Valeur de X	40d	41d	42d	43d	44d	45d	46d	47d	48d	49d	50d	51d
Y=68d									1	2	3	
Y=68d								4	5	6	7	
Y=68d						8	9	10	11	12	13	14
Y=68d						15	16	17	18	19	20	21
Y=68d							22				23	

Puis dessin en X=44d Y=68d :

Valeur de X	40d	41d	42d	43d	44d	45d	46d	47d	48d	49d	50d	51d
Y=68d								1	2	3		
Y=68d							4	5	6	7		
Y=68d					8	9	10	11	12	13	14	
Y=68d					15	16	17	18	19	20	21	
Y=68d						22				23		

On constate que le dessin laisse une trace : en effet, le pixel (par exemple) X=50d Y=68d a été noirci lors de l'opération précédente, mais n'a pas été reblanchi dans la présente opération. Si on ne provoque pas ce « reblanchiement », il n'a aucune raison de se réaliser tout seul (l'octet correspondant en mémoire contient toujours le même code couleur). Pour illustration, la suite des opérations serait la suivante :

Dessin en X=43d Y=68d :

Valeur de X	40d	41d	42d	43d	44d	45d	46d	47d	48d	49d	50d	51d
Y=68d							1	2	3			
Y=68d						4	5	6	7			
Y=68d				8	9	10	11	12	13	14		
Y=68d				15	16	17	18	19	20	21		
Y=68d					22				23			

Etc…, jusqu'en X=40d Y=68d par exemple :

Valeur de X	40d	41d	42d	43d	44d	45d	46d	47d	48d	49d	50d	51d
Y=68d				1	2	3						
Y=68d			4	5	6	7						
Y=68d	8	9	10	11	12	13	14					
Y=68d	15	16	17	18	19	20	21					
Y=68d		22				23						

En effet, nous n'avons pas effacé l'arrière de la voiture avant chaque affichage de la nouvelle position. Nous allons présenter ici deux possibilités pour le faire.

Première solution

Une première solution consiste à inclure des bords blancs à la voiture :

	Origine	Colonne +1	Colonne +2	Colonne +3	Colonne +4	Colonne +5	Colonne +6	Colonne +7	Colonne +8
Origine					24	25	26		
Ligne+1				42	1	2	3	27	
Ligne+2		40	41	4	5	6	7	28	
Ligne+3	39	8	9	10	11	12	13	14	29
Ligne+4	38	15	16	17	18	19	20	21	30
Ligne+5		37	22	35	34	33	23	31	
Ligne+6			36				32		

Cela se réalise en complétant le tableau VOITURE :

Adresse	Libellé du contenu	Contenu
VOITURE	Nombre de points à traiter	42
VOITURE + 1 à VOITURE + 69	Idem précédemment, en tenant compte du changement d'origine (+1 pour chaque décalage horizontal et chaque décalage vertical)	
VOITURE + 70	Décalage horizontal du point 24	4
VOITURE + 71	Décalage vertical du point 24	0
VOITURE + 72	Couleur du point 24	1
Etc…		
VOITURE + 124	Décalage horizontal du point 42	3
VOITURE + 125	Décalage vertical du point 42	1
VOITURE + 126	Couleur du point 42	1

Cet exemple ajoute des bords blancs dans les quatre directions (gauche, droite, haut, bas), en prévision de déplacements dans les directions opposées, afin d'effacer la traînée. Dans le cas du déplacement précédent (de droite à gauche), les points 27 à 31 et 35 auraient permis d'empêcher la traînée en « reblanchissant » les pixels du dessin précédent. Si on prévoit des déplacements dans huit directions (les quatre précédentes plus leurs combinaisons, comme « en haut à droite »), il faut ajouter des bords de coin (43 à 53) :

	Origine	Colonne +1	Colonne +2	Colonne +3	Colonne +4	Colonne +5	Colonne +6	Colonne +7	Colonne +8
Origine				43	24	25	26	44	
Ligne+1			53	42	1	2	3	27	
Ligne+2	52	40	41	4	5	6	7	28	45
Ligne+3	39	8	9	10	11	12	13	14	29
Ligne+4	38	15	16	17	18	19	20	21	30
Ligne+5	51	37	22	35	34	33	23	31	46
Ligne+6		50	36	49		48	32	47	

L'avantage de la méthode est sa simplicité. Son inconvénient est qu'elle suppose que l'objet se déplace sur un fond blanc (ou tout au moins uni, si on change la couleur blanche du bord pour la couleur de fond), ce qui est rarement le cas.

Deuxième solution

Reprenons la première « voiture » :

	Origine	Colonne +1	Colonne +2	Colonne +3	Colonne +4	Colonne +5	Colonne +6
Origine				1	2	3	
Ligne+1			4	5	6	7	
Ligne+2	8	9	10	11	12	13	14
Ligne+3	15	16	17	18	19	20	21
Ligne+4		22				23	

Pour permettre une animation sur un fond quelconque (notamment pas nécessairement uni), il est nécessaire d'enregistrer celui-ci, aux endroits où nous allons modifier les pixels en dessinant notre voiture. Il s'agit donc de faire le travail inverse du programme MOTIF : passer chaque position pointée par le tableau VOITURE, mais au lieu de copier le code couleur dans le pixel correspondant de l'écran, il s'agit de lire le pixel sur l'écran pour l'enregistrer en mémoire vive, dans une structure du même type que VOITURE que l'on utilisera par la suite dans le programme MOTIF pour « recoller » le fond originel à la place de la voiture. Nous appellerons cette structure *FOND*, et le programme inverse de MOTIF, *ENR_FOND* (pour « ENREGISTRER_FOND »).

1ère image : Situation initiale. L'écran (c'est-à-dire la mémoire vidéo située en RAM) contient une image, la RAM contient la définition d'un dessin à partir de l'adresse VOITURE (ce que, par simplicité de vocabulaire, nous appelons le « tableau VOITURE »). Nous devons d'abord créer la structure FOND, grâce à un programme CREE_FOND. Ici, par soucis de lisibilité, le dessin est volontairement grossi.

2ème image : ENR_FOND enregistre, à l'adresse FOND, les pixels de la mémoire vidéo que l'on prévoit d'écraser par le programme MOTIF affichant VOITURE.

3ème image : MOTIF affiche VOITURE.

4ème image : MOTIF affiche FOND, rétablissant ainsi l'affichage initial.

Nous devons donc créer deux programmes :

- CREE_FOND : copie VOITURE vers FOND, afin de bénéficier de sa structure et des données que nous conserverons (en fait, nous n'en modifierons que les codes couleurs). Cette étape sera articulée autour d'une instruction LDIR (copie de blocs mémoire).
- ENR_FOND : passe en revue chaque position indiquée par FOND. Pour chacune d'entre elles, lit le code couleur sur l'écran et l'enregistre comme code couleur dans FOND.

L'animation consiste, après avoir utilisé CREE_FOND et pour chaque position successive à donner à la voiture dans l'animation, en :

- appeler ENR_FOND
- appeler MOTIF, en indiquant VOITURE comme adresse de motif
- attendre « un certain temps »
- appeler MOTIF, en indiquant FOND comme adresse de motif
- incrémenter la position à traiter le coup suivant

L'attente a pour but de laisser la voiture affichée un temps suffisant pour que la rétine la voie. Cela peut être constitué par un réel programme d'attente, ou bien « naturellement » par d'autres calculs que le programme doit réaliser de toutes manières pour d'autres fonctions (gestion d'autres animations, lectures / écritures sur le disque dur ou en mémoire, détection des touches utilisées du clavier, etc…).

Voici une solution possible du programme CREE_FOND :

Adresse (hexa)	Mnémonique	Commentaire
-4	NOP	Sert à inscrire l'octet de poids fort de l'adresse du motif à copier (ici, le programme appelant inscrira l'octet de poids fort de VOITURE)
-3	NOP	Sert à inscrire l'octet de poids faible de l'adresse du motif à copier
-2	NOP	Sert à inscrire l'octet de poids fort de l'adresse FOND où le MOTIF sera copié (ici, le programme appelant inscrira l'octet de poids fort de FOND)

Adresse (hexa)	Mnémonique	Commentaire	
-1	NOP	Sert à inscrire l'octet de poids faible de l'adresse FOND où le MOTIF sera copié	
CREE_FOND	PUSH hl	Ce sous-programme utilisant *hl*, sauvegarde *hl* pour le programme appelant	Sauvegarde des registres
	PUSH bc	Idem pour *bc*	
	PUSH de	Idem pour *de*	
	LD hl,(CREE_FOND -4)	hl pointe sur VOITURE	
	LD c,(hl)	Inscrit le nombre de points enregistrés par VOITURE (signification du premier octet du tableau VOITURE) dans c.	Calcule la taille, en octets, du tableau VOITURE (nombre d'octets à copier, à inscrire dans *bc* pour l'instruction LDIR)
	LD b,0	Ainsi, *bc* contient (*hl*)	
	LD l,c	Avec l'instruction suivante, réalise LD *hl*,*bc*	
	LD h,0		
	ADD hl,bc	Réalise $hl = 2 \times bc$	
	ADD hl,bc	Réalise $hl = 3 \times bc$	
	INC hl	Réalise $hl = 3 \times bc + 1$	
	LD b,h	Avec l'instruction suivante, réalise LD *bc*,*hl*	
	LD c,l	Ainsi, *bc* contient le nombre d'octets du tableau VOITURE	
	LD hl,(CREE_FOND -4)	*hl* pointe sur VOITURE, source de la copie à réaliser.	Adresse source de la copie
	LD de,(CREE_FOND -2)	*de* pointe sur FOND, destination de la copie à réaliser.	Adresse destination de la copie
	LDIR	Réalise la copie de *bc* octets depuis VOITURE vers FOND.	Copie les bc octets de VOITURE depuis hl (VOITURE) vers de (FOND)
	POP de		Restaure les registres pour le programme appelant
	POP bc		
	POP hl		
	RET		

Voici une solution possible du programme ENR_FOND (avec un sous-programme LIT_PIXEL qui retourne le code couleur du point X,Y de la mémoire vidéo, détaillé après) :

Adresse (hexa)	Mnémonique	Commentaire	
-4	NOP	Sert à inscrire la valeur de X où il est prévu de dessiner le motif	Données
-3	NOP	Sert à inscrire la valeur de Y où il est prévu de dessiner le motif	
-2	NOP	Sert à inscrire l'octet de poids fort de l'adresse où le fond sera enregistré (ici, le programme appelant inscrira l'octet de poids fort de FOND)	
-1	NOP	Sert à inscrire l'octet de poids faible de l'adresse où le fond sera enregistré	
ENR_FOND	PUSH hl	Ce sous-programme utilisant *hl*, sauvegarde *hl* pour le programme appelant	Sauvegarde des registres
	PUSH bc	Idem pour *bc*	
	PUSH de	Idem pour *de*	
	LD hl,(ENR_FOND -2)	Inscrit dans *hl* l'adresse FOND	Détermination du nombre de points
	LD a,(hl)	Inscrit dans *a* le contenu de l'adresse FOND, c'est-à-dire nombre de points à traiter	
	LD b,a	L'enregistre dans *b* (compteur de DJNZ)	

Adresse (hexa)	Mnémonique	Commentaire	
BOUCLE2	INC hl	Incrémente *hl*, pour passer à l'adresse suivante (décalage horizontal)	Détermination de l'abscisse du point en cours dans le motif
	LD a,(ENR_FOND - 4)	Inscrit X dans *a*	
	LD d,a	Enregistre X dans *d*	
	LD a,(hl)	Inscrit le décalage dans *a*	
	ADD a,d	Somme X et le décalage	
	LD (LIT_PIXEL-2),a	Inscrit le résultat dans le X de LIT_PIXEL	
	INC hl	Incrémente *hl*, pour passer à l'adresse suivante (décalage vertical)	Détermination de l'ordonnée du point en cours dans le motif
	LD a,(ENR_FOND - 3)	Inscrit Y dans *a*	
	LD d,a	Enregistre Y dans *d*	
	LD a,(hl)	Inscrit le décalage dans *a*	
	ADD a,d	Somme Y et le décalage	
	LD (LIT_PIXEL-1),a	Inscrit le résultat dans le Y de LIT_PIXEL	
	INC hl	Incrémente *hl*, pour passer à l'adresse suivante (couleur)	Enregistrement de la couleur du point en cours dans le motif
	CALL LIT_PIXEL	Inscrit le code couleur du pixel pointé dans *a* (voir ci-dessous)	
	LD (hl),a	L'inscrit alors dans le tableau FOND.	
	DJNZ BOUCLE2	Passe au point suivant, jusqu'au dernier (déterminé par la valeur de *b*)	Pixel suivant
	POP de		Restaure les registres pour le programme appelant
	POP bc		
	POP hl		
	RET		

Comment modifier légèrement PIXEL pour qu'il réalise la fonction LIT_PIXEL qu'on attend de lui ? Voici la réponse :

Adresse (hexa)	Mnémonique	Commentaire
-2	NOP	Cet octet est réservé à la donnée X
-1	NOP	Cet octet est réservé à la donnée Y
LIT_PIXEL	PUSH hl	Idem PIXEL
	LD a,(LIT_PIXEL-1)	
	ADD a,10h	
	LD h,a	
	LD a,(LIT_PIXEL-2)	
	LD l,a	
	LD a,(hl)	Inscrit le code couleur du pixel pointé dans *a*
	POP hl	Idem PIXEL
	RET	

Désormais, nous avons tous les outils pour réaliser notre animation :
Causer du temps de chaque instruction################
CREE_FOND (VOITURE,FOND)
Initialise X
Début de boucle :
ENR_FOND (X,Y,FOND)
MOTIF (VOITURE)
Attente
MOTIF (FOND)
X <- X+1
Retour boucle

Adresse (hexa)	Mnémonique	Commentaire	
-6	NOP	Sert à inscrire la valeur de X_d de départ pour l'animation	Données
-5	NOP	Sert à inscrire la valeur de X_f de fin pour l'animation	
-4	NOP	Sert à inscrire l'octet de poids fort de l'adresse VOITURE	
-3	NOP	Sert à inscrire l'octet de poids faible de l'adresse VOITURE	
-2	NOP	Sert à inscrire l'octet de poids fort de l'adresse FOND	
-1	NOP	Sert à inscrire l'octet de poids faible de l'adresse FOND	
ANIMATION	PUSH hl	Ce sous-programme utilisant *hl*, sauvegarde *hl* pour le programme appelant	Sauvegarde des registres
	PUSH bc	Idem pour *bc*	
	PUSH de	Idem pour *de*	
	LD hl,(ANIMATION-2)	*hl* contient l'adresse du tableau FOND	Initialisation des données pour l'ensemble des sous-programmes
	LD (CREE_FOND-2),hl	L'adresse de FOND est copiée comme donnée du sous-programme CREE_FOND	
	LD (ENR_FOND-2),hl	L'adresse de FOND est copiée comme donnée du sous-programme ENR_FOND	
	LD hl,(ANIMATION-4)	*hl* contient l'adresse du tableau VOITURE	
	LD (CREE_FOND-2),hl	L'adresse de VOITURE est copiée comme donnée du sous-programme CREE_FOND	
	LD (MOTIF-2),hl	L'adresse de VOITURE est copiée comme donnée du sous-programme MOTIF	
	LD a,100d	On fixe arbitrairement la valeur de Y à 100d	
	LD (ENR_FOND-3),a	Qu'on inscrit en donnée fixe de ENR_FOND…	
	LD (MOTIF-3),a	…et de MOTIF	
	CALL CREE_FOND	Crée la structure FOND	
	LD a,(ANIMATION-6)	Charge le X_d de début dans *a*	Calcule le nombre de fois où la boucle d'animation devra être exécutée = X_f - X_d
	LD b,a		
	LD a,(ANIMATION-5)	Charge le X_f de fin dans *a*	
	SUB a,b	Inscrit X_f - X_d dans a	
	LD b,a	Transfère le résultat dans *b*, car *b* est le registre de contrôle de la future boucle *DJNZ*.	
	LD a,(ANIMATION-6)	Recharge le X_d de début dans *a*	Initialise la première position de X = X_d

Adresse (hexa)	Mnémonique	Commentaire	
BOUCLE3	LD (ENR_FOND-4),a	Transmets la valeur en cours de X au sous-programme ENR_FOND	Enregistre le fond.
	CALL ENR_FOND	Enregistre le fond d'écran à la position X en cours	
	LD hl,VOITURE	Inscrit l'adresse VOITURE dans *hl*	Dessine la voiture
	LD (MOTIF-2),hl	Inscrit l'adresse VOITURE à l'adresse MOTIF-2	
	CALL MOTIF	Dessine la voiture à la position X en cours.	
	LD hl,5555d	Initialise *hl* à 5555d	Boucle d'attente…
BOUCLE4	DEC hl	Décrémente *hl*. Le drapeau Z est à 1 si le résultat est nul, Z = 0 sinon.	Pour laisser le temps à la voiture d'être affichée assez longtemps pour être vue.
	JR NZ BOUCLE4	Retourne en BOUCLE4 si Z = 0.	Cette boucle ne fait rien d'autre que d'être exécutée 5555d fois
	LD hl,FOND	Inscrit l'adresse FOND dans *hl*	Réaffiche le fond
	LD (MOTIF-2),hl	Inscrit l'adresse FOND à l'adresse MOTIF-2	
	CALL MOTIF	Réaffiche le fond à la position X en cours.	
	INC a	Incrémente *a*	Donne à X la valeur suivante
	DJNZ BOUCLE3	Décrémente *b*, et retourne en BOUCLE3 si *b* est non nul.	Passe au traitement du pixel suivant jusqu'au dernier (déterminé par la valeur de *b*).
	POP de		Restaure les registres pour le programme appelant
	POP bc		
	POP hl		
	RET		Retourne à l'appelant.

Boucle d'attente : dans le Z80,

- *LD hl,nn* utilise 10 cycles d'horloge
- *DEC hl* utilise 6 cycles d'horloge
- *JR NZ n* utilise 12 cycles d'horloge quand le branchement a lieu, 7 sinon.

Ainsi, l'ensemble de la boucle utilise 10 + 5555 × (6+12) – (12-7) ≈ 100 000 cycles d'horloge. Avec un Z80 travaillant à 1 MHz, on force ici une attente de 0,1 seconde, permettant d'avoir un temps d'affichage de la voiture suffisant pour qu'elle soit vue par la rétine, et en tous cas bien plus important que le temps entre son effacement et son affichage à la position suivante.

LECTURE DES TOUCHES DU CLAVIER

##########

Commander le déplacement de la voiture en fonction de la touche enfoncée. Genre :
Ca serait pas mal si la voiture se déplaçait en fonction des touches qu'on presse (flèches gauche ou droite).

Afficher la voiture
Attendre qu'une touche soit pressée (attente pour rendre la voiture visible)
Effacer la voiture
Afficher la nouvelle voiture
Attendre que la touche ne soit plus pressée

ECRIVONS

Affichage d'un caractère

Le code ASCII

Affichage d'un texte

CALCUL DU TEMPS DE TRAITEMENT#

Sous-programmes

#RAM = programme
parler de SP#

La pile

MACRO-ASSEMBLAGE