Aide – Simulateur Dauphin Web

(N. Kunz – 2024)

Table des matières

1. Description du simulateur

1.1. Interface graphique

Nous avons décidé d'avoir une seule interface plutôt que d'avoir les vues multiples présentes dans le « *simulateur de Dauphin* », pour des raisons d'accessibilité et de flux des actions (cf. Figure 1). Toutes les fonctionnalités de *Dauphin Web* sont ainsi disponibles depuis la même page, divisée en sept parties.

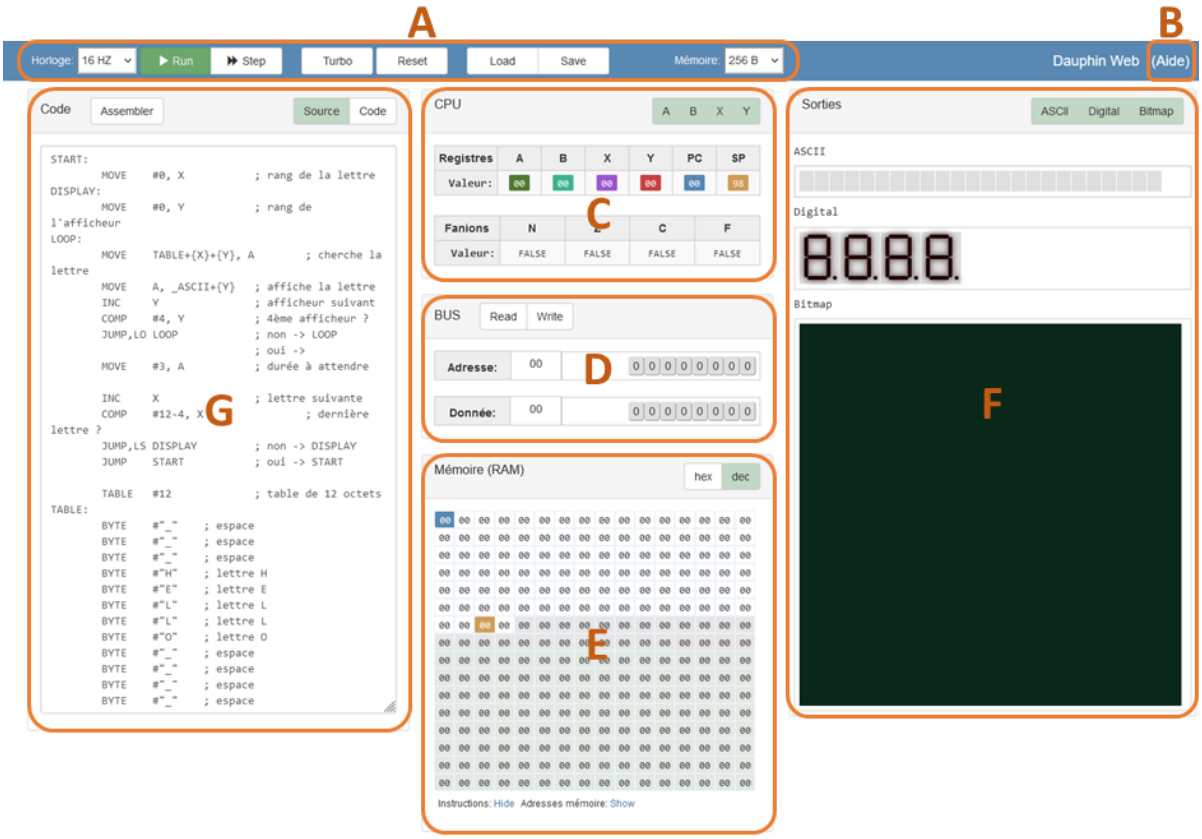


Figure 1: Interface graphique de Dauphin Web divisée en sept parties. A. Contrôles du simulateur B. Aide du simulateur C. Bus d'adresses et de données D. Registres du CPU E. Mémoire (RAM) F. Sorties G. Editeur d'assembleur

A. Contrôles du simulateur

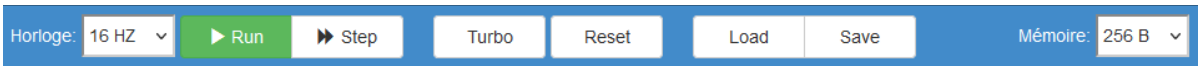


Figure 2 : Contrôle du simulateur

Le ruban d'en-tête en haut de la page permet de contrôler le simulateur. Positionné en haut de la page, il reste toujours visible. De droite à gauche de la Figure 2 on trouve : un menu déroulant donnant le choix, avant de lancer un programme, de la fréquence de l'horloge du CPU et donc de sa vitesse d'exécution. Ensuite, le bouton « Run » lance le simulateur, il est aussi possible de lancer le simulateur dans un mode pas à pas avec le bouton « Step ». Une fois le programme lancé, deux nouveaux boutons apparaissent (cf. Figure 3) permettant soit de mettre le programme en pause et de reprendre son exécution en mode pas-à-pas (« Step ») ou en continu (« Run »), soit de l'arrêter et de réinitialiser la mémoire et les affichages (« Stop »). Le bouton « Reset » remet le simulateur à zéro en effaçant la mémoire et les affichages et ainsi que le programme assemblé. Il est possible d'accélérer l'exécution du simulateur avec le bouton « Turbo » qui masque tous les éléments excepté les sorties.

Les programmes peuvent également être enregistrés sous forme de fichier texte ou chargés dans le simulateur grâce aux boutons « Save » et « Load » respectivement.

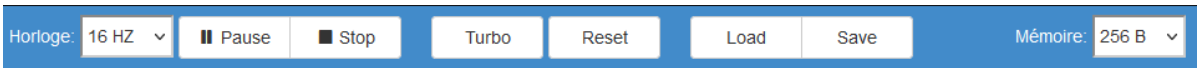


Figure 3 : Contrôle du simulateur une fois le programme lancé

Le bouton de sélection « Mémoire » permet de changer la taille de l'espace mémoire. Par défaut, il est réglé à 256 octets avec un bus d'adresses de 8 bits, ce qui est suffisant pour tous les programmes présentés dans ce document. Pour des usages plus avancés, il est possible d'augmenter la taille de la mémoire jusqu'à 4096 octets avec un bus d'adresses de 12 bits, ce qui correspond aux caractéristiques du « *simulateur de Dauphin* ». Nous avons fait le choix de 256 octets dans la vue par défaut, pour aider les utilisateurs non initiés à suivre ce qui se passe dans la pile. Une fois qu'ils maîtrisent les différents concepts, une mémoire « longue » qui demanderait à se déplacer dans la page n'est plus si problématique.

B. Aide du simulateur

Figure 4 : Bouton d'accès à l'aide du simulateur

Un dernier élément est présent tout à droite du ruban d'en-tête, qui donne accès à une page d'aide sur le simulateur. Cette page contient entre autres des explications sur le langage assembleur utilisé par *Dauphin Web* et la liste de toutes les opérations supportées.

C. Bus d'adresses et de données

Figure 5 : Panneau de contrôle des bus d'adresses et de données. À gauche avec les valeurs en décimales et une mémoire de 256 octets. À droite avec les valeurs en hexadécimales et une mémoire est de 4096 octets.

Une addition importante au simulateur de base sont les bus d'adresses et de données présents dans Dauphin. Une première façon d'utiliser ce simulateur est de passer par le panneau de contrôle des bus d'adresses et de données. Il permet d'accéder à la mémoire de *Dauphin Web* en mode lecture (bouton « Read ») ou écriture (bouton « Write »).

Pour écrire une valeur dans la mémoire, il faut tout d'abord sélectionner l'adresse de destination en binaire à l'aide des boutons contrôlant chacun un des douze bits disponibles. L'adresse choisie est ensuite affichée à gauche en notation décimale ou hexadécimale suivant le choix de l'utilisateur (cf. chapitre mémoire (RAM)). Il va de même pour la valeur à enregistrer dans la mémoire. Une fois ces deux opérations effectuées, il suffit d'appuyer sur le bouton « Write ». On pourra aller vérifier dans la mémoire que la valeur a bien été mémorisée (cf. Figure 7).

Pour lire une valeur depuis la mémoire, il suffit de sélectionner l'adresse souhaitée comme précédemment, et d'appuyer sur le bouton « Read ». La valeur sera chargée dans le bus de données et affichée en binaire ainsi qu'en décimale ou hexadécimale (au choix).

Les bus sont aussi mis à jour en cours d'exécution d'un programme. On peut observer les valeurs lues et écrites en mémoire. Cette fonctionnalité a une grande valeur pédagogique et est surtout utile dans le mode d'exécution pas-à-pas.

On peut aussi remarquer que le bus d'adresses peut contenir jusqu'à 12 bits alors que celui de données seulement 8 bits. Cette différence sera expliquée dans le chapitre Mémoire.

D. Registres du CPU

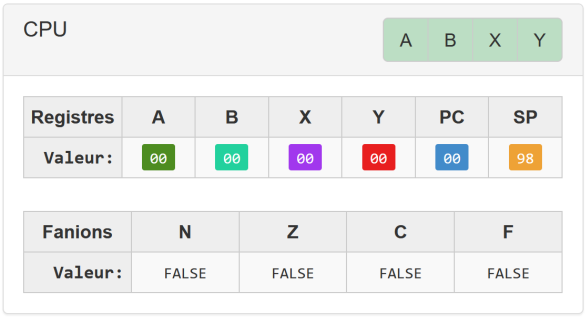


Figure 6 : Valeur des six registres et des quatre fanions du CPU

Ce panneau indique les valeurs des six registres de *Dauphin Web* et les quatre fanions de contrôle. Ces valeurs sont mises à jour à chaque pas d'exécution du programme. Le code couleur des registres permet de visualiser leur valeur dans la mémoire (cf. Figure 7). Il est possible d'activer et de désactiver cette option grâce aux boutons au sommet à droite du panneau pour les registres A, B, X et Y, afin de pouvoir

se concentrer sur un registre en particulier. Ces quatre premiers registres contiennent des valeurs de 8 bits, et donc leur représentation dans la mémoire n'est qu'un rendu visuel de ces valeurs, mais sans réelle signification. Par exemple, si la valeur décimale 12 est mémorisée dans le registre A, le marqueur vert sera positionné sur la case mémoire correspondant à l'adresse 12 (ou H'00C). En revanche, les registres PC et SP désignent bel et bien une adresse mémoire et ont donc leur représentation a une réelle signification mémoire (RAM).

E. Mémoire (RAM)

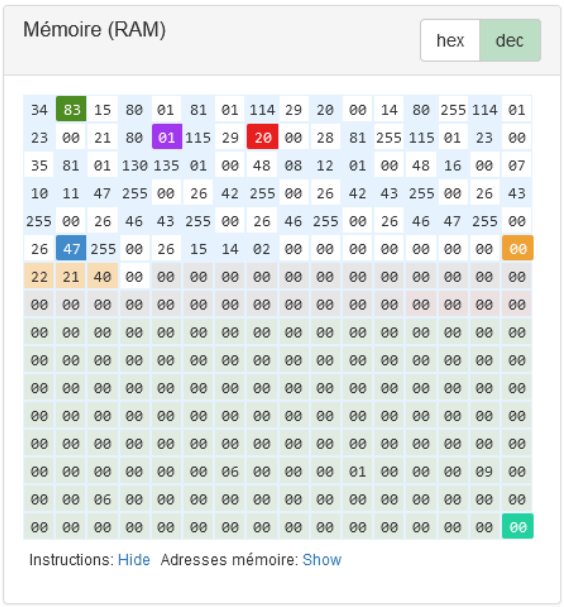


Figure 7 : Panneau d'affichage de la mémoire de *Dauphin Web* d'une capacité de 256 octets. Les valeurs sont représentées en notation décimale . Les registres sont rendus visibles à l'aide du code couleur de la Figure 6 (A, B, X, Y, SP, PC). Les cases mémoires utilisées par la pile sont en orange. Les cases contenant les codes d'instruction, en bleu clair, sont aussi des liens vers le code assembleur. Finalement, la fin de l'espace mémoire correspond aux adresses réservées pour les affichages (ASCII, Digital et Bitmap).

Ce panneau représente l'état de la mémoire du simulateur à chaque étape d'exécution du programme. Plusieurs options d'affichage sont disponibles :
Au sommet du panneau :

- Les boutons «hex» et «dec» *:* permettent de passer en affichage décimal ou hexadécimal des valeurs et des adresses.

En bas du panneau (cf. Figure 7) :

- *Instructions :* les cases mémoires contenant le code d'opération sont colorées en bleu clair. Il est ensuite possible de cliquer sur ces cases pour voir la ligne correspondante dans le code assembleur (cf. Figure 10).
- *Adresses mémoire :* permet d'ajouter une première ligne dans chaque case mémoire indiquant l'adresse de la case (cf. Figure 8 à gauche). Il est aussi possible d'accéder à cette information en laissant la souris immobile sur une case mémoire (cf. Figure 8 à droite).

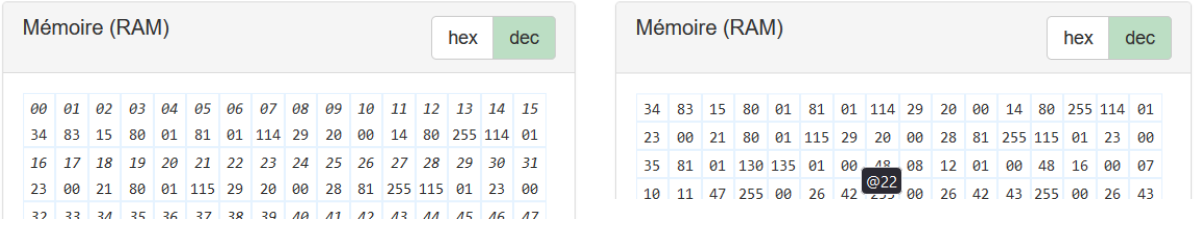


Figure 8 : Accès à l'adresse des cases mémoire.
À gauche, à l'aide du bouton en bas du panneau. À droite, à l'aide de la souris.

F. Sorties

Dauphin Web offre trois possibilités de sortie à travers trois types d'affichage distinct (cf. Figure 9).

- La sortie «ASCII» contient 24 cases permettant d'afficher les symboles de la table ASCII selon leur code d'entrée.
- La sortie «Digital» contient quatre afficheurs composés chacun de huit segments.
- La sortie «Bitmap» est un écran de 32×32 pixels.

Figure 9 : Les 3 types d'affichage de Dauphin Web de haut en bas :

- *Sortie de 24 cellules d'affichage ASCII.*
- *Sortie de 4 afficheurs digitaux à 8 segments.*
- *Sortie écran Bitmap (32×32 pixels).*

Il est possible de les cacher ou de les rendre visibles grâce aux boutons au sommet de ce panneau.

Les deux dernières sorties peuvent aussi être utilisées également comme entrées. En cliquant sur un segment (dans la sortie « Digital ») ou sur un pixel (dans la sortie « Bitmap »), cela active ou désactive le segment/pixel et modifie l'état de la mémoire en conséquence.

Il faut noter que les deux dernières sorties ne sont pas présentes dans le simulateur « *Simple 8-bit Assembler Simulator* » de M. Schweighauser.

G. Éditeur d'assembleur

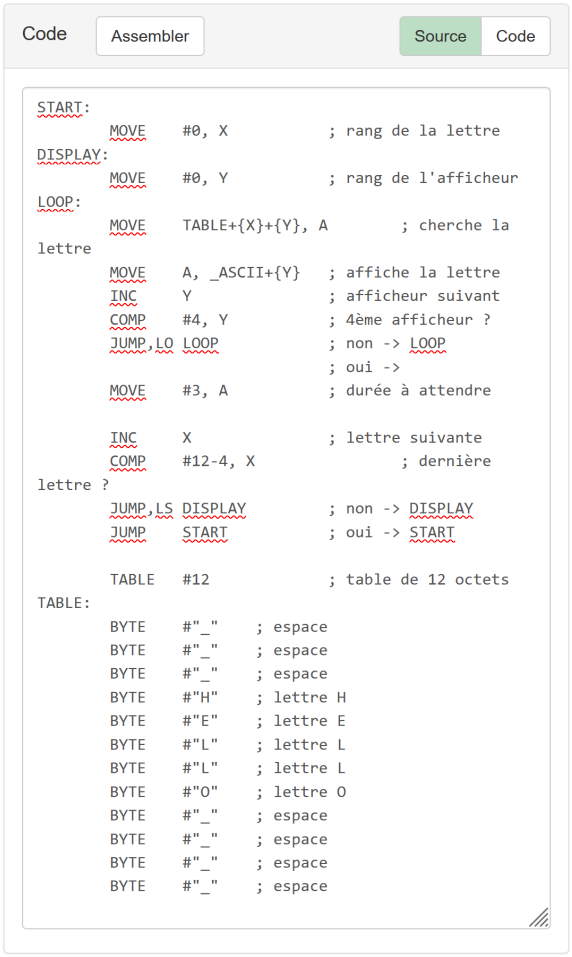


Figure 10 : Éditeur d'assembleur en mode source

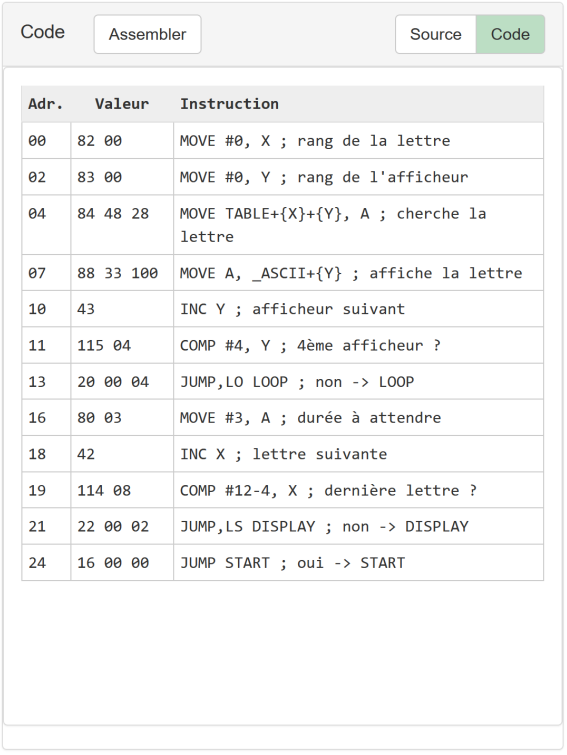


Figure 11 : Éditeur d'assembleur après assemblage en mode code

L'éditeur d'assembleur permet d'écrire un programme ou de modifier un programme existant directement à l'intérieur du simulateur. Il est aussi possible d'utiliser un éditeur de texte externe et d'ouvrir le fichier dans le simulateur. Il faut toutefois faire attention aux symboles spéciaux comme les retours à la ligne ou les simples et doubles guillemets qui peuvent être mal interprétés.

Une fois le programme terminé, il faut l'assembler grâce au bouton du même nom au sommet de ce panneau (cf. Figure 10). Cette opération se fait automatiquement si on lance le programme avec le bouton « Start » du ruban d'en-tête. Si le programme contient une erreur et ne peut être interprété correctement, un message est affiché au sommet de la page et la ligne contenant l'erreur est sélectionnée (cf. Figure 12).

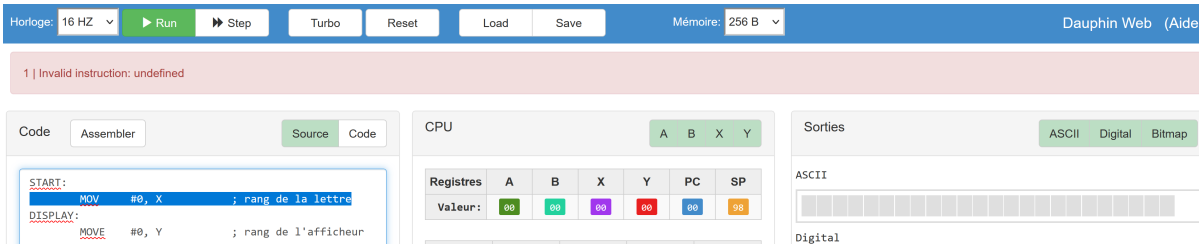


Figure 12 : Message d'erreur dans l'encadré rouge indiquant une instruction pas définie à la 2ème ligne du programme coloriée en bleu

Une fois le programme assemblé, un second mode d'affichage est disponible à travers le bouton « Code » de ce panneau (cf. Figure 11). Dans ce mode, le programme apparait sous forme de tableau. Chaque ligne contient une instruction du programme. La première colonne indique l'adresse mémoire du début de l'instruction. La deuxième contient les codes mémoires de l'opération et des opérandes. La dernière colonne montre l'instruction dans le langage assembleur.

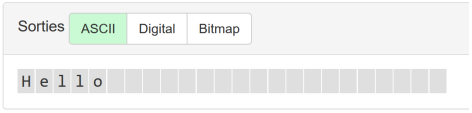
1.2. Bus

Le bus est une fonctionnalité qui a été reprise du « *simulateur de Dauphin* » et ajoutée au «*Simple 8-bit Assembler Simulator*». Il est ainsi possible de charger en mémoire manuellement des instructions en respectant les codes des opérations ou de modifier un des trois affichages de sorties.

Par exemple avec une mémoire de 256 octets, en chargeant les valeurs binaires suivantes aux adresses indiquées, on obtiendra :

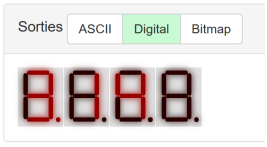
- Le mot «Hello» dans la sortie «ASCII» :

Adresse	Valeur
B'01100100	B'01001000
B'01100101	B'01100101
B'01100110	B'01101100
B'01100111	B'01101100
B'01101000	B'01101111



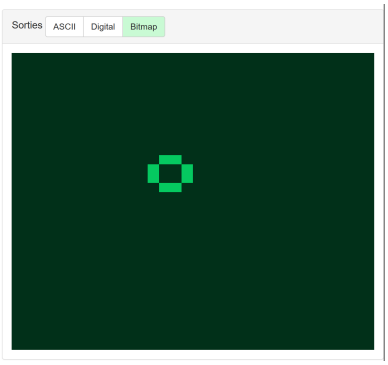
- Le chiffre 3.14 dans la sortie «DIGITAL» :

Adresse	Valeur
B'01111100	B'11100111
B'01111101	B'00000011
B'01111110	B'01010011



- Un cercle dans la sortie «Bitmap» :

Adresse	Valeur
B'10101101	B'00000110
B'10110001	B'00001001
B'10110101	B'00001001
B'10111001	B'00000110



Le fonctionnement des sorties est présenté en détail dans le chapitre «Périphériques de sortie».

Le bus de contrôle n'est pas représenté dans le simulateur. On peut avoir un aperçu de son fonctionnement grâce aux boutons «Read» et «Write» qui commandent l'écriture ou la lecture d'une cellule mémoire (cf. Figure 13).

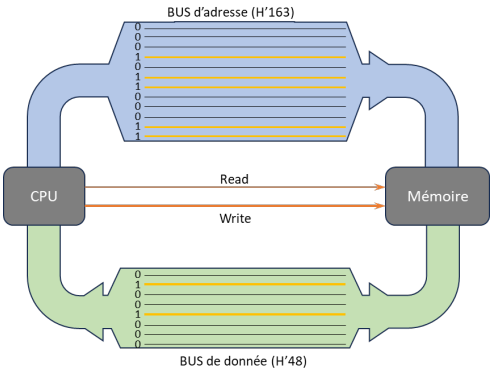


Figure 13 : Fonctionnement des bus de *Dauphin Web* dans la configuration de l'espace mémoire de 4096 octets.

S'il est vrai que l'ALU n'est capable de faire des opérations uniquement sur des valeurs de 8 bits, ce qui équivaut à la largeur du bus de données, l'espace mémoire lui est ajustable avec un minimum de 256 octets, idéal pour les activités proposées dans ce travail, et un maximum de 4096 octets, correspond à l'espace mémoire total (ROM + RAM + périphériques) du « *simulateur de Dauphin* ». Pour cela il est nécessaire d'ajuster la largeur du bus d'adresse de 8 bits à 12 bits respectivement.

1.3. Registres

Dauphin Web reprend les sept registres du « *simulateur de Dauphin* » et ajoute un 4ème fanion (F) présent dans le « *Simple 8-bit Assembler Simulator* ». Le fanion F prend la valeur de 1 lors d'une erreur d'exécution.

Les deux registres spécifiques de pointeur d'instruction et de pile (PC et SP respectivement), ont la propriété de pouvoir être encodés avec un maximum de 12 bits car ils représentent des adresses mémoires. L'adresse qu'ils pointent est mise en évidence dans le panneau mémoire en colorant la cellule en bleu pour PC et en orange pour SP (cf. Figure 7).

Les registres A, B, X et Y sont des registres de nombres entiers et peuvent être utilisés sans distinction. Seules quelques instructions sont limitées à l'une des deux paires de registres. A et B seront toujours utilisés comme registres de nombres entiers et les codes des instructions concernées seront différenciés en utilisant r'. Cela concerne principalement les instructions d'opérations logiques et celles de tests et modifications de bit. Alors que la paire X, Y peut aussi jouer le rôle de registre d'index et leur valeur peut être ajoutée à une adresse fixe, on parlera dans leur cas d'adressage relatif. Cette pratique est utilisée par exemple pour parcourir un tableau comme dans l'exemple de programme «Hello ASCII»:

```
MOVE   TABLE+{X}+{Y}, A    ; cherche la lettre
MOVE   A, _ASCII+{Y}       ; affiche la lettre
```

Le registre X mémorise la première lettre à afficher (tout à gauche) dans la TABLE, quant au registre Y la position à laquelle cette lettre doit être affichée dans la sortie ASCII. Ces registres sont incrémentés à tour de rôle dans le reste du programme. Ainsi la première commande va mémoriser le code ASCII de la lettre à la (X+Y)^{ème} position de la TABLE dans le registre A. Puis, la deuxième instruction va copier la valeur fraichement mémorisée dans A dans le Y^{ème} case de la sortie ASCII.

1.4. Mémoire

L'espace mémoire de *Dauphin Web* est ajustable (cf. Figure 2) partant d'une taille de 256 octets jusqu'à 4096 octets. La première configuration a été ajoutée par rapport au « *simulateur de Dauphin* », pour garantir un affichage complet de l'état du simulateur sur une page web sans avoir besoin de défiler son contenu. De plus, tous les programmes présentés dans ce travail ne nécessitent pas plus de mémoire. Ces 256 octets permettent en outre de construire des séquences pédagogiques couvrant tous les aspects du CPU décrit en annexe (cf. chapitre **Erreur ! Source du renvoi introuvable.**). Cependant, il était important de garder la plus grande compatibilité possible avec le « *simulateur de Dauphin* » et donc d'offrir une version qui conserve sa taille d'espace mémoire d'origine. Dans cette deuxième configuration, l'adaptation de la largeur du bus d'adresses est aussi intéressante à des fins pédagogiques illustrant le lien direct avec la taille de l'espace mémoire. Il faut toutefois noter que la mémoire ROM contenant les routines intégrées au simulateur (cf. Routines) n'est pas accessible dans *Dauphin Web* contrairement à son ainé.

L'état de la mémoire est mis à jour à chaque pas d'exécution du programme. Il est possible d'afficher les valeurs stockées soit au format décimal soit au format hexadécimal (cf. Figure 7, *View*). L'adresse de chaque case mémoire est accessible en tout temps soit en positionnant le curseur de la souris au-dessus, soit en activant le double affichage (cf. Figure 7, *Adresses mémoire*).

Les données enregistrées dans la mémoire peuvent être accédées visuellement dans le panneau « Mémoire (RAM) » (cf. Figure 7) ou chaque ligne représente 16 octets de la mémoire. Autrement, les données peuvent être lues séparément en utilisant la commande « Read » du bus (cf. Figure 5).

Pour écrire dans la mémoire du simulateur, il existe trois possibilités :

- Premièrement, en utilisant la commande « Write » du bus (cf. Figure 5).
- Deuxièmement, les instructions de l'assembleur permettent de modifier l'état de la mémoire.
- Finalement, à travers les sorties « DIGITAL » et « BITMAP », où il est possible d'allumer ou d'éteindre les différents éléments de ces affichages en cliquant dessus.

1.5. Périphériques de sortie

Les deux périphériques de sortie du « *simulateur de Dauphin* » ont été implémentés, avec une augmentation pour l'affichage Bitmap de 24 lignes à 32 lignes pour permettre des dessins plus complets. L'affichage de caractères ASCII de « *Simple 8-bit Assembler Simulator* » a aussi été conservé.

L'espace mémoire réservé aux sorties est situé à la fin de la RAM :

Tableau 1 : Espace mémoire réservé aux sorties

Mémoire de 256 octets	
Adresses	**Contenu**
H'64 … H'7B	Affichage ASCII
H'7C … H'7F	Affichage DIGITAL
H'80 … H'FF	Affichage BITMAP

Mémoire de 4096 octets	
Adresses	**Contenu**
H'F64 … H'F7B	Affichage ASCII
H'F7C … H'F7F	Affichage DIGITAL
H'F80 … H'FFF	Affichage BITMAP

Pour faciliter l'accès mémoire aux différents affichages et permettre d'utiliser le même programme indépendamment de la taille de la mémoire, une série de constantes est disponible dans le langage assembleur explicitées ci-dessous. Généralement, elles correspondent à l'adresse mémoire de la première cellule de l'affichage et de sa taille. Les valeurs des constantes et présentées dans les figures de ce chapitre sont valables pour une taille de l'espace mémoire de 256 octets.

Affichage ASCII

L'affichage ASCII est composé de 24 cellules dans lesquelles il est possible d'afficher 256 symboles du tableau ASCII étendu (Multinational Character Set, 2023). Pour l'utiliser, il suffit de déplacer le code correspondant au symbole choisi à l'adresse de l'affichage souhaité (de H'64 pour la cellule à l'extrémité gauche, jusqu'à H'7B pour la cellule à l'extrémité droite, Figure 14).

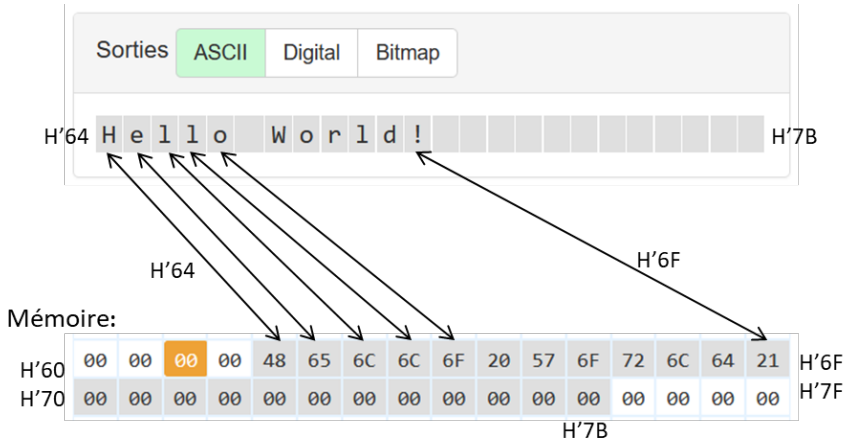


Figure 14 : Lien entre l'affichage ASCII et l'espace mémoire. Les adresses mémoire et les valeurs enregistrées sont affichées en hexadécimal. La première case de l'affichage correspond à l'adresse mémoire H'64 et contient la valeur H'48 qui correspond au caractère «H» du tableau ASCII.

Constantes de l'assembleur :

Nom	Valeur	Définition
_ASCII	H'64	Adresse de la première cellule toute à gauche
_ASCIILENGTH	#24	Nombre de cellules

Affichage DIGITAL

L'affichage Digital est composé de quatre cellules composées chacune de huit segments. Chaque cellule correspond à un octet de la mémoire, et chaque segment à un bit de cet octet. Quand la valeur du bit correspondant est égale à 0, le segment est éteint, quand sa valeur est égale à 1, le segment est allumé.

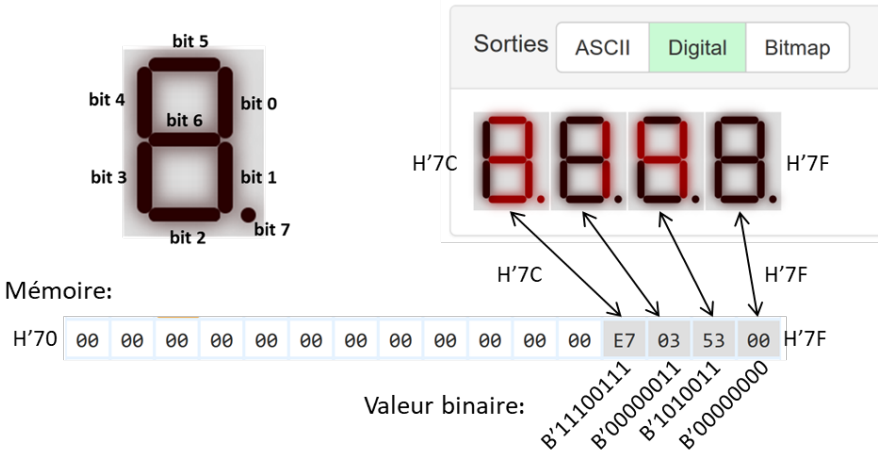


Figure 15 : Lien entre l'affichage Digital et la mémoire. Le coin supérieur gauche donne la correspondance entre chaque segment et son bit mémoire correspondant.

Constantes de l'assembleur :

Nom	Valeur	Définition
_DIGIT0	H'7C	Adresse de la première cellule toute à gauche
_DIGIT1	H'7D	Adresse de la deuxième cellule depuis la gauche
_DIGIT2	H'7E	Adresse de la troisième cellule depuis la gauche
_DIGIT3	H'7F	Adresse de la quatrième cellule depuis la gauche
_DIGITLENGTH	#4	Nombre de cellules

Affichage BITMAP

L'affichage Bitmap est composé de 32×32 pixels représentant les 128 derniers octets de l'espace mémoire. Chaque ligne est donc constituée de 4 octets, chaque octet contient 8 bits, et chaque bit contrôle 1 pixel de l'affichage (cf. Figure 16). Quand la valeur du bit correspondant est égale à 0, le pixel est éteint, quand sa valeur est égale à 1, le pixel est allumé.

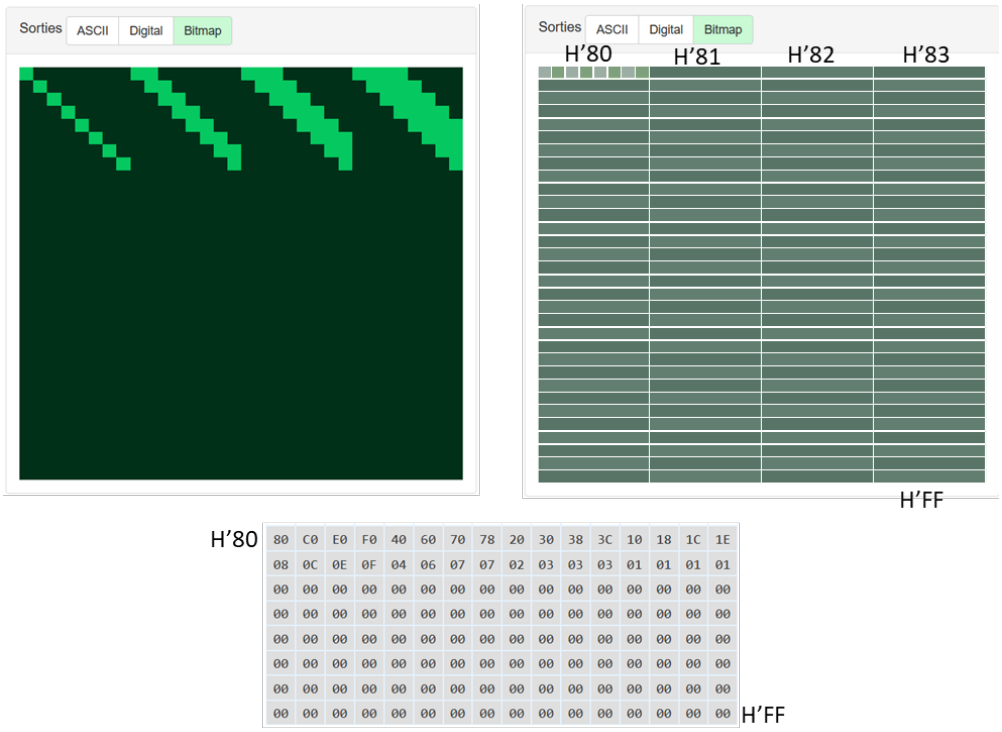


Figure 16 : Lien entre l'affichage Bitmap et l'espace mémoire

Constantes de l'assembleur :

Nom	Valeur	Définition
_BITMAP	H'80	Adresse du premier pixel au sommet gauche
_BITMAPLENGTH	#128	Nombre d'octets attribués à l'affichage
_BITMAPWIDTH	#32	Nombre de colonnes
_BITMAPHEIGHT	#32	Nombre de lignes

1.6. Assembleur

Le langage assembleur choisi pour ce simulateur est celui du « *simulateur de Dauphin* » qui utilise le langage CALM défini au LAMI 1975 (Nicoud, 1975). La syntaxe et les instructions sont identiques et ce qui permet d'exécuter les mêmes programmes dans les deux simulateurs. Il est basé sur la syntaxe AT&T, c'est-à-dire que l'opérande de la source précède celle de la destination. Ce changement de langage a demandé des modifications en profondeurs dans le code du « *Simple 8-bit Assembler*

Simulator » en commençant par l'analyse syntaxique, les codes des opérations, et bien entendu le traitement des instructions par le simulateur.

Syntaxe

Nombre

Les valeurs numériques peuvent être exprimées dans les trois bases les plus communes en informatique :

- Décimal : les nombres décimaux peuvent être écrits sans préfixe, ou précédés de « D' ».
- Binaire : les nombres binaires doivent être précédés du préfixe « B' ».
- Hexadécimal : les nombres hexadécimaux doivent être précédés du préfixe « H' ».

Tableau 2 : Exemples de notation des valeurs numériques

Décimal	***Binaire***	***Hexadécimal***
0	B'00000000	H'00
1	B'00000001	H'01
2	B'00000010	H'02
9	B'00001001	H'09
10	B'00001010	H'0A
11	B'00001011	H'0B

Les valeurs numériques sont encodées sur 8 bits et donc peuvent prendre des valeurs entre 0 et 255 non signées ou entre -128 et 127 pour les valeurs signées.

Adresse

Les adresses sont encodées sur 16 bits quel que soit la taille de l'espace mémoire et donc utilisent deux octets. Les 12 bits de poids faibles sont utilisés pour coder l'adresse. Quand une instruction fait appel à une adresse relative (…+{X} ou {PC}+…), cette information est encodée dans les 4 bits restants de poids le plus fort. Sous format hexadécimal, deux octets sont représentés par quatre symboles : 3 pour l'adresse absolue de 12 bits et 1 pour les adresses relatives.

Tableau 3 : Adresse absolue et relative

[ma] [aa]	**Adresse**	**valeurs**
m=0	Adresse absolue 12 bits	[0, 4096]
m=1	+{X}	[0, 255]
m=2	+{Y}	[0, 255]
m=3	+{X}+{Y}	[0, 510]
m=4	{SP}±valeur signée	[-128, 127]
m=8	{PC}±-valeur signée	[-128, 127]

Exemples :

Instruction	**Mémoire**	
H'1FF	[01] [FF]	Adresse absolue
H'FF+{X}	[**1**0] [FF]	La valeur contenue dans le registre X sera ajoutée à H'FF
H'180+{X}+{Y}	[**3**1] [80]	Les valeurs contenues dans les registres X et Y seront ajoutées à H'180
{PC}+12	[**8**0] [0C]	On ajoutera 12 à l'adresse contenue dans le registre SP

Adresse vs. valeur numérique

Les adresses sont différenciées des valeurs numériques en notant un # devant ces dernières :

Exemple : `MOVE #17, H'FF` → transfert la valeur numérique 17 à l'adresse H'FF

Commentaires

Il est possible de commenter le code en utilisant le symbole « ; ». Tout ce qui suit le symbole sur la même ligne est considéré comme un commentaire.

Exemple : `MOVE #0, X ; Ici commence le commentaire`

Labels

Le langage assembleur permet de définir des labels qui peuvent être utilisés soit comme marqueur pour un saut et se déplacer dans le programme, soit pour définir une routine.

Le label doit commencer par des lettres et finir par le symbole « **:** ». S'il s'agit d'une routine, il faudra la terminer avec l'instruction « RET » qui ramènera le programme à l'instruction suivant l'appel de la routine.

Par exemple, dans le programme Rebond3, le label « LOOP : » est utilisé comme marque pour un saut (cf. ligne 17 : JUMP LOOP), tandis que le label « NOTBALL : » désigne une routine et fini par l'instruction « RET » et est appelée aux lignes 14 et 16 (CALL NOTBALL).

Instructions

Cette partie décrit de manière globale les différents types d'opérations et reprend le deuxième manuel du « *simulateur de Dauphin* » (EPSITEC, Comprendre les microprocesseurs II, 2023). La liste complète des instructions ainsi que leur code d'opération est disponible en annexe. Chaque instruction a un nom qui exprime l'opération effectuée, suivi de l'opérande source et de l'opérande destination séparée par une virgule. Les registres seront désignés par leur abréviation (A, B, X, Y, PC, SP), il en va de même pour les fanions (C, N, Z, F), les valeurs numériques par #VAL et les adresses par ADDR.

Déplacement et opération arithmétiques

Syntaxe :

$$\begin{matrix} MOVE \\ ADD \\ SUB \\ COMP \end{matrix}\left\{\begin{matrix} A \\ B \\ X \\ Y \\ \#\,VAL \\ ADDR \end{matrix}\right\},\left\{\begin{matrix} A \\ B \\ X \\ Y \end{matrix}\right\} \quad \text{ou} \quad \begin{matrix} MOVE \\ ADD \\ SUB \end{matrix}\left\{\begin{matrix} A \\ B \\ X \\ Y \\ \#\,VAL \end{matrix}\right\},ADDR$$

Explications :

- `MOVE #3, A ;` copie la valeur de la source dans la destination. Ici le registre A prend la valeur décimale de 3 (#3→A).
- `ADD H'FF, X ;` Ajoute la valeur contenue à l'adresse H'FF à celle contenue dans le registre X et mémorise le résultat dans ce dernier ([H'FF]+X→X).
- `SUB B, Y ;` Attention l'instruction soustrait la valeur du 2ème opérande à la 1ère (B-Y→Y) !
- `COMP A, B ;` la comparaison effectue une soustraction, mais ne modifie que les fanions. Le résultat n'est pas mémorisé dans la destination (A-B→ fanions).

Fanions

Les fanions sont mis à jour à la fin de la plupart des opérations et peuvent servir de condition pour une future instruction de saut. Si une opération ne modifie pas les fanions, leurs valeurs sont préservées.

Explications :

- Le fanion Z (Zéro) est activé à 1 si le résultat de l'opération est nul.
- Le fanion N (Négatif) est activé si le 7ème bit est à 1.

- Le fanion C (Carry) est activé si l'addition déborde (> 255) ou que la soustraction s'est terminée par un emprunt (< -128).

Retenue :	*1*	*1*	*1*	*1*	*1*	*1*	*1*		
		1	0	1	1	0	1	1	0
+		0	1	0	0	1	0	1	0
		0	0	0	0	0	0	0	0

H'B6
+H'4A
H'00

C=1 N=0 Z=1

emprunt :							*1*		
		1	0	1	1	0	1	1	0
-		0	0	1	1	0	1	0	1
		1	0	0	0	0	0	0	1

H'B6
-H'35
H'81

C=0 N=1 Z=0

Opérations logiques

Syntaxe :

$$\left.\begin{matrix} AND \\ OR \\ XOR \end{matrix}\right\} \left\{\begin{matrix} A,B \\ B,A \end{matrix}\right. \quad \text{ou} \quad \left.\begin{matrix} AND \\ OR \\ XOR \end{matrix}\right\} \# VAL, \left\{\begin{matrix} A \\ B \\ X \\ Y \end{matrix}\right.$$

$$\left.\begin{matrix} AND \\ OR \\ XOR \end{matrix}\right\} \left\{\begin{matrix} A \\ B \end{matrix}\right\}, ADDR \quad \text{ou} \quad \left.\begin{matrix} AND \\ OR \\ XOR \end{matrix}\right\} ADDR, \left\{\begin{matrix} A \\ B \end{matrix}\right\}$$

Explications :

Les opérations logiques sont effectuées bit à bit :

- `AND` : donne un bit à 1 dans la destination si les deux bits correspondants sont à 1.
- `OR` : donne un bit à 1 dans la destination si l'un des deux bits correspondants est à 1.
- `XOR` : donne un bit à 1 dans la destination si l'un des deux bits correspondants est à 1, mais pas les deux.

		1	0	1	1	0	1	1	0
AND		0	0	1	0	0	0	1	1
		0	0	1	0	0	0	1	0

H'B6
+H'23
H'22

		1	0	1	1	0	1	1	0
OR		0	0	1	0	0	0	1	1
		1	0	1	1	0	1	1	1

H'B6
+H'23
H'B7

		1	0	1	1	0	1	1	0
XOR		0	0	1	0	0	0	1	1
		1	0	0	1	0	1	0	1

H'B6
+H'23
H'95

Sauts et appels

L'instruction de saut est fondamentale, elle permet d'altérer le déroulement du programme en sautant une série d'instructions, ou en revenant en arrière pour créer une boucle. De plus, il existe des sauts conditionnels qui ne se déclenchent qu'en fonction de l'état des fanions rattachés.

Syntaxe :

Certaines instructions de saut conditionnel sont équivalentes et sont inscrites sur la même ligne ci-dessous :

$$\begin{matrix}JUMP\\CALL\end{matrix}\left\{\begin{matrix}ADDR\\Label\end{matrix}\right\} \quad \text{ou} \quad JUMP,\left\{\begin{matrix}EQ\ ou\ ZS\\NE\ ou\ ZC\\LO\ ou\ CS\\LS\\HI\\HS\ ou\ CC\\NC\\NS\end{matrix}\right\}\left\{\begin{matrix}ADDR\\Label\end{matrix}\right\}$$

Explications :

Le Tableau ci-dessous présente l'état des fanions pour que le saut conditionnel soit effectué :

Tableau 4 : Liste des conditions pour les sauts

Instruction	Condition
`JUMP ADDR`	aucune
`JUMP,EQ` ou `JUMP,ZS`	Z=1
`JUMP,NE` ou `JUMP,ZC`	Z=0
`JUMP,LO` ou `JUMP,CS`	C=1
`JUMP,HS` ou `JUMP,CC`	C=0
`JUMP,LS ADDR`	C=1 OR Z=1
`JUMP,HI ADDR`	C=0 AND Z=0
`JUMP,NS ADDR`	N=1
`JUMP,NC ADDR`	N=0

L'exemple de programme Suite de Fibonacci utilise le saut conditionnel JUMP,CC, qui permet au programme de continuer sa boucle tant que le résultat ne déborde pas (c.-à-d. tant que C=0).

Opérations à un opérande

Syntaxe :

$$\begin{matrix}CLR\\NOT\\INC\\DEC\end{matrix}\left\{\begin{matrix}A\\B\\X\\Y\\ADDR\end{matrix}\right\}$$

Explications :

- `CLR A` ; met le registre A à 0, et par conséquent les fanions N=0 et Z=1
- `NOT B` ; inverse tous les bits un à un de B
- `INC X` ; ajoute 1 au registre X
- `DEC Y` ; soustrait 1 au registre Y

Décalages

Syntaxe :

$$\begin{matrix}RR\\RL\\RRC\\RLC\end{matrix}\left\{\begin{matrix}A\\B\\X\\Y\\ADDR\end{matrix}\right\}$$

Explications :

- `RR A` ; permute les bits de A vers la droite

- `RL B` ; permute les bits de B vers la gauche
- `RRC X` ; permute les bits de X vers la droite en utilisant le carry comme 9ème bit
- `RLC Y` ; permute les bits de Y vers la gauche en utilisant le carry comme 9ème bit

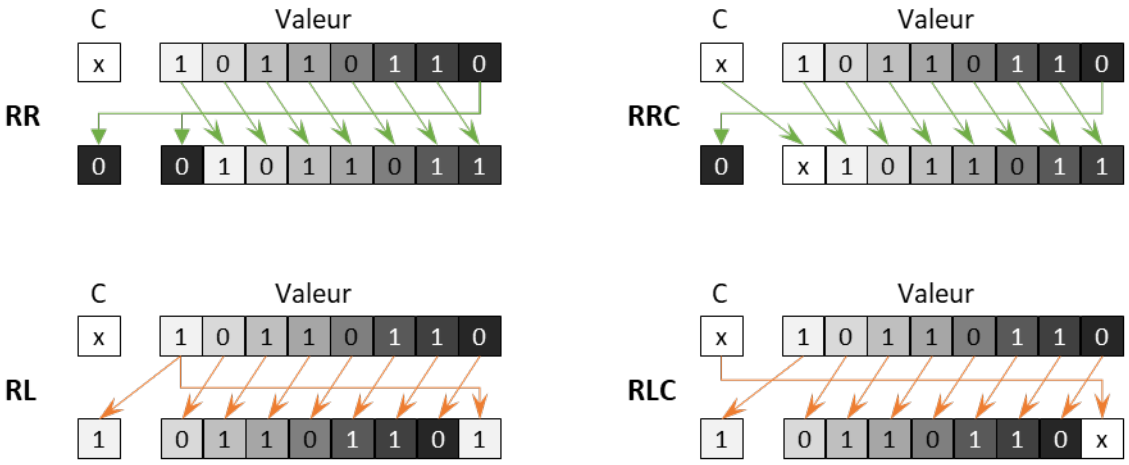


Figure 17 : Illustration des instructions de décalage de bit

L'exemple de programme en annexe « Segment cyclique » permet de visualiser ces instructions dans l'affichage Digital.

Test et modification de bits

Syntaxe :

$$\left.\begin{matrix}TEST\\TSET\\TCLR\\TNOT\end{matrix}\right\}\ \left\{\begin{matrix}A:B\\B:A\end{matrix}\right\}\text{ou}\qquad \left.\begin{matrix}TEST\\TSET\\TCLR\\TNOT\end{matrix}\right\}\ \left\{\begin{matrix}A\\B\\ADDR\end{matrix}\right\}:\#\,VAL\qquad\text{ou}\qquad \left.\begin{matrix}TEST\\TSET\\TCLR\\TNOT\end{matrix}\right\}\ ADDR:\left\{\begin{matrix}A\\B\end{matrix}\right\}$$

Explications :

- `TEST A : #3` ; teste si le 3ème bit de A est égal à 1, si oui Z=0, sinon Z=1
- `TSET B : #1` ; teste si le 1er bit de B et met à jour Z, puis passe le 1er bit de B à 1
- `TCLR A : #0` ; teste si le 0ème bit de A et met à jour Z, puis passe le 0ème bit de A à 0
- `TNOT B : #5` ; teste si le 5ème bit de B et met à jour Z, puis inverse le 5ème bit de B

Pile

Syntaxe :

$$\left.\begin{matrix} PUSH \\ POP \end{matrix}\right\} \left\{\begin{matrix} A \\ B \\ X \\ Y \end{matrix}\right\}$$

Explications :

- `PUSH A ;` place la valeur de A au-dessus de la pile et soustrait 1 au registre SP
- `POP B ;` retire le dernier élément de la pile et l'enregistre dans B et ajuste le registre SP

Routines internes

Le « *simulateur de Dauphin* » possède plusieurs routines facilitant l'utilisation des affichages et qui sont contenues dans la mémoire ROM. Ces routines ont été reprises dans *Dauphin Web*, à la différence qu'elles ne sont plus visibles par l'utilisateur.

Leur utilisation et description est accessible dans les annexes (cf. Routines), nous nous focaliserons uniquement sur deux d'entre elles qui permettent d'afficher une partie des 127 symboles de la table ASCII dans les afficheurs Digital et Bitmap.

_DisplayCharDigit

Cette routine permet d'afficher facilement l'une des 26 lettres de l'alphabet (majuscule ou minuscule quand cela est possible) ou l'un des dix chiffres dans l'afficheur Digital. Pour cela, il suffit de placer dans le registre A le code ASCII du symbole souhaité et dans le registre B la position de la cellule choisie (de 0 à 3 en partant de la gauche). Une fois ces deux opérations effectuées, il ne reste plus qu'à appeler la routine avec l'instruction «CALL _DisplayCharDigit». Voici quelques exemples :

```
MOVE #72, A     ; sélectionne la lettre H
MOVE #0, B      ; sélectionne la première cellule de l'afficheur Digital
CALL _DisplayCharDigit      ;affiche la lettre H dans la cellule tout à gauche

MOVE "h", A
MOVE #1, B
CALL _DisplayCharDigit      ;affiche h dans la 2ème cellule depuis la gauche
```

_DrawChar

Cette routine permet d'afficher facilement, avec une représentation 3×3, l'une des 26 lettres de l'alphabet (majuscule ou minuscule) ou l'un des dix chiffres dans l'afficheur Bitmap et quelques symboles spéciaux (+, -, (,), [,], !, ?, =, /, %, ., , , ;, <, >). Pour cela il suffit de placer dans le registre A le code ASCII du symbole souhaité, dans le registre X la position horizontale de la lettre (de 0 à 7 en partant de la gauche) et dans le registre Y la position verticale de la lettre (de 0 à 3). Une fois ces trois opérations effectuées, il ne reste plus qu'à appeler la routine avec l'instruction « CALL _DrawChar ». Voici un exemple (cf. Figure 18) :

```
MOVE #67, A     ; sélectionne la lettre C
MOVE #0, X      ; sélectionne la première position tout à gauche
MOVE #0, Y      ; sélectionne la première ligne
CALL _DrawChar  ; affiche la lettre C au sommet gauche de l'afficheur Bitmap
MOVE "i", A
INC X           ; se déplace d'une case vers la droite
INC Y           ; se déplace d'une ligne vers le bas
CALL _DrawChar;
MOVE "a", A
INC X
INC Y
```

```
CALL _DrawChar;
MOVE "o", A
INC X
INC Y
CALL _DrawChar;
MOVE #H'21, A   ; sélectionne le symbole ! du tableau ASCII
INC X
INC Y
CALL _DrawChar;
Halt            ; arrête l'exécution du programme
```

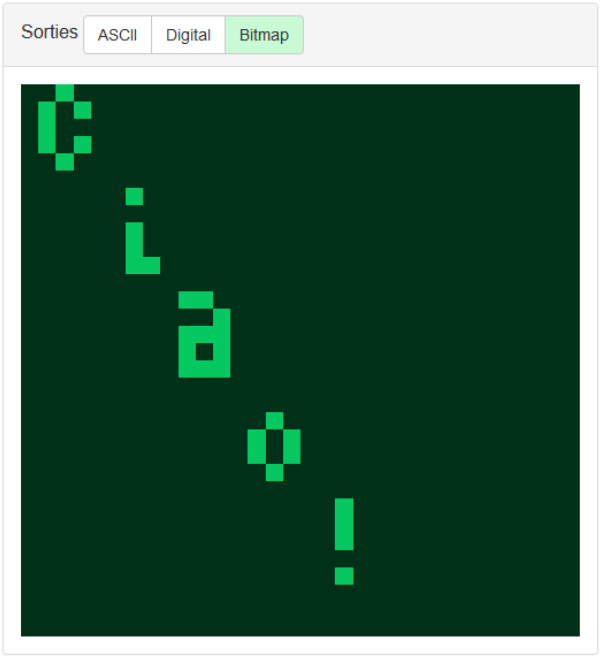


Figure 18 : Résultat de l'exemple d'utilisation de la routine «_DrawChar»

2. Listes des instructions et routines

2.1. Liste des instructions

Ces informations sont disponibles depuis *Dauphin Web* par le lien Aide.

Chaque instruction est codée sur un octet. Ce code est donné en hexadécimal dans la liste qui suit.

Registre

[xx+r]	r
r=0	A
r=1	B
r=2	X
r=3	Y

[xx+r']	r'
r'=0	A
r'=1	B

Valeur immédiate

[vv]	#val
vv	Valeur positive 8 bits

Si la valeur est plus grande que 128, cela met le fanion N à 1.

Adresse

Les adresses sont mémorisées sur 2 octets en mémoire. Les 12 bits de poids les plus faibles mémorisent l'adresse, les deux derniers bits sont utilisés pour les adresses relatives.

[ma] [aa]	Adresse	
m=0	Adresse absolue12 bits	
m=1	+{X}	[0, 255]
m=2	+{Y}	[0, 255]
m=3	+{X}+{Y}	[0, 510]
m=4	{SP}±valeur signée	[-128, 127]
m=8	{PC}±-valeur signée	[-128, 127]

Exemples

MOVE H'107, A ; [54] [0C] [07] adresse absolue, lit la Valeur mémorisée à l'adresse H'107 et la copie dans le registre A

MOVE A, H'100+{X}; [58] [1C] [00]

JUMP {PC}+10 ; [10] [80] [10] (saute 10 octets)

JUMP {PC}-3 ; [10] [8F] [FD] (boucle infinie)

MOVE {SP}+2, A ; [54] [40] [02]

Transferts

Code	Instruction	Fanions
[40+r]	MOVE A, *r*	(N, Z)
[44+r]	MOVE B, *r*	(N, Z)
[48+r]	MOVE X, *r*	(N, Z)
[4C+r]	MOVE Y, *r*	(N, Z)
[50+r] [vv]	MOVE *#val*, *r*	(N, Z)
[54+r] [ma] [aa]	MOVE *ADDR*, *r*	(N, Z)

[58+r] [ma] [aa]	MOVE *r*, *ADDR*	(N, Z)
[DC] [vv] [ma] [aa]	MOVE #val, ADDR	(N, Z)

Exemple :

MOVE #12, B ; [51] [0C] Copie la valeur 12 dans le registre B

Additions

Code	Instruction	Fanions
[80+r]	ADD A, *r*	(N, Z, C)
[84+r]	ADD B, *r*	(N, Z, C)
[88+r]	ADD X, *r*	(N, Z, C)
[8C+r]	ADD Y, *r*	(N, Z, C)
[A0+r] [vv]	ADD *#val*, *r*	(N, Z, C)
[B0+r] [ma] [aa]	ADD *ADDR*, *r*	(N, Z, C)
[B8+r] [ma] [aa]	ADD *r*, *ADDR*	(N, Z, C)
[DE] [vv] [ma] [aa]	ADD #val, ADDR	(N, Z, C)

Exemple :

ADD H'0FF, B ; [B1] [00] [FF] Ajoute la valeur mémorisée à l'adresse H'0FF à la valeur du registre B et mémorise le résultat dans le registre B

Soustractions

Code	Instruction	Fanions
[90+r]	SUB A, *r*	(N, Z, C)
[94+r]	SUB B, *r*	(N, Z, C)
[98+r]	SUB X, *r*	(N, Z, C)
[9C+r]	SUB Y, *r*	(N, Z, C)
[A4+r] [vv]	SUB *#val*, *r*	(N, Z, C)
[B4+r] [ma] [aa]	SUB *ADDR*, *r*	(N, Z, C)
[BC+r] [ma] [aa]	SUB *r*, *ADDR*	(N, Z, C)
[DF] [vv] [ma] [aa]	ADD #val, ADDR	(N, Z, C)

Exemple :

SUB X, Y ; [9B] Soustrait la valeur du registre X à Y et mémorise le résultat dans le registre Y

ET logique

Code	Instruction	Fanions
[E0]	AND A, B	(N, Z)
[E1]	AND B, A	(N, Z)
[74+r] [vv]	AND *#val*, *r*	(N, Z)
[E8+r'] [ma] [aa]	AND *ADDR*, *r'*	(N, Z)
[F0+r'] [ma] [aa]	AND *r'*, *ADDR*	(N, Z)

Exemple :

AND A, B ; [E0] Compare bit par bit la valeur de A avec celle B en utilisant l'opérateur logique AND et mémorise le résultat dans le registre B

OU logique

Code	Instruction	Fanions
[E2]	OR A, B	(N, Z)
[E3]	OR B, A	(N, Z)
[78+r] [vv]	OR *#val*, *r*	(N, Z)
[EA+r'] [ma] [aa]	OR ADDR, r'	(N, Z)
[F2+r'] [ma] [aa]	OR r', ADDR	(N, Z)

Exemple :

OR B, A ; [E3] Compare bit par bit la valeur de B avec celle A en utilisant l'opérateur logique OR et mémorise le résultat dans le registre A

Test de bit

Code	Instruction	Fanions
[E4]	XOR A, B	(N, Z)
[E5]	XOR B, A	(N, Z)
[7C+r] [vv]	XOR *#val*, *r*	(N, Z)
[EC+r'] [ma] [aa]	XOR *ADDR*, *r'*	(N, Z)
[F4+r'] [ma] [aa]	XOR *r'*, *ADDR*	(N, Z)

Exemple :

XOR H'0B8, B ; [ED] [00] [B8] Compare bit par bit la valeur mémorisée à l'adresse H'0B8 avec celle B en utilisant l'opérateur logique XOR et mémorise le résultat dans le registre B

Test de bit

Code	Instruction	Fanions
[C0]	TEST B: A	(Z)
[C1]	TEST A: B	(Z)
[D0+r'] [vv]	TEST r': *#val*	(Z)
[C8+r'] [ma] [aa]	TEST ADDR: r'	(Z)
[D8] [vv] [ma] [aa]	TEST ADDR: #val	(Z)

Exemple :

TEST A: #6 ; [D0] [06] Test la valeur du 6ème bit du registre A et modifie le fanion Z en conséquence (bit = 1 → Z=false et si le bit = 0 → Z=true)

Test de bit et mise à un

Code	Instruction	Fanions
[C2]	TSET B: A	(Z)
[C3]	TSET A: B	(Z)
[D2+r'] [vv]	TSET r': *#val*	(Z)
[CA+r'] [ma] [aa]	TSET ADDR: r'	(Z)
[D9] [vv] [ma] [aa]	TSET ADDR: #val	(Z)

Exemple :

TSET B: #5 ; [D3] [05] Test la valeur du 5ème bit du registre B et modifie le fanion Z en conséquence (bit = 1 → Z=false et si le bit = 0 → Z=true) puis modifie la valeur du bit à 1.

Test de bit et mise à zéro

Code	Instruction	Fanions
[C4]	TCLR B: A	(Z)
[C5]	TCLR A: B	(Z)
[D4+r'] [vv]	TCLR r': *#val*	(Z)
[CC+r'] [ma] [aa]	TCLR ADDR: r'	(Z)
[DA] [vv] [ma] [aa]	TCLR ADDR: #val	(Z)

Exemple :

TCLR B: #5 ; [D5] [05] Test la valeur du 5ème bit du registre B et modifie le fanion Z en conséquence (bit = 1 → Z=false et si le bit = 0 → Z=true) puis modifie la valeur du bit à 0.

Test de bit et inversion

Code	Instruction	Fanions
[C6]	TNOT B: A	(Z)
[C7]	TNOT A: B	(Z)

[D6+r'] [vv]	TNOT r': *#val*	(Z)
[CE+r'] [ma] [aa]	TNOT ADDR: r'	(Z)
[DB] [vv] [ma] [aa]	TNOT ADDR: #val	(Z)

Exemple :

TNOT B: #5 ; [D7] [05] Test la valeur du 5ème bit du registre B et modifie le fanion Z en conséquence (bit = 1 → Z=false et si le bit = 0 → Z=true) puis inverse la valeur du bit (0→1, 1→0).

Comparaisons

Code	Instruction	Fanions
[60+r]	COMP A, *r*	(N, Z, C)
[64+r]	COMP B, *r*	(N, Z, C)
[66+r]	COMP X, *r*	(N, Z, C)
[6C+r]	COMP Y, *r*	(N, Z, C)
[70+r] [vv]	COMP *#val*, *r*	(N, Z, C)
[F8+r] [ma] [aa]	COMP *ADDR*, *r*	(N, Z, C)
[DD+r'] [vv] [ma] [aa]	COMP #val, ADDR	(N, Z, C)

Exemple :

COMP #B'01100001, Y ; [73] [61] Compare la valeur #B'01100001 avec celle mémorisée dans le registre Y. La comparaison est en fait une soustraction de #B'01100001 à Y, mais le résultat n'est pas mémorisé, seuls les fanions sont mis à jour.

Opérations unaires

Code	Instruction	Fanions
[20+r]	CLR *r*	(N=0, Z=0)
[24+r]	NOT *r*	(N, Z)
[28+r]	INC *r*	(N, Z)
[2C+r]	DEC *r*	(N, Z)
[A8] [ma] [aa]	CLR ADDR	(N=0, Z=0)
[A9] [ma] [aa]	NOT ADDR	(N, Z)
[AA] [ma] [aa]	INC ADDR	(N, Z)
[AB] [ma] [aa]	DEC ADDR	(N, Z)

Exemple :

INC A ; [28] Augmente de 1 la valeur mémorisée dans le registre A et modifie ce dernier

Rotations

Code	Instruction	Fanions
[30+r]	RL *r*	(N, Z, C)
[34+r]	RR *r*	(N, Z, C)
[38+r]	RLC *r*	(N, Z, C)
[3C+r]	RRC *r*	(N, Z, C)
[AC] [ma] [aa]	RL ADDR	(N, Z, C)
[AD] [ma] [aa]	RR ADDR	(N, Z, C)
[AE] [ma] [aa]	RLC ADDR	(N, Z, C)
[AF] [ma] [aa]	RRC ADDR	(N, Z, C)

Exemple :

RR X ; [36] Exécute un décalage vers la droite des bits de la valeur mémorisée dans X et modifie ce dernier. (il faut 8 décalages successifs pour retrouver la valeur de départ)

RLC H'0A5 ; [AE] [00] [A5] Exécute un décalage vers la gauche de la valeur mémorisée à l'adresse H'0A5 en utilisant le carry comme 8ème bit. (il faut 9 décalages successifs pour retrouver la valeur de départ)

Sauts

Code	Instruction	Condition
[10] [ma] [aa]	JUMP ADDR	-
[12] [ma] [aa]	JUMP,EQ ADDR	Z=1
[12] [ma] [aa]	JUMP,ZS ADDR	Z=1
[13] [ma] [aa]	JUMP,NE ADDR	Z=0
[13] [ma] [aa]	JUMP,ZC ADDR	Z=0
[13] [ma] [aa]	JUMP,NE ADDR	Z=0
[14] [ma] [aa]	JUMP,LO ADDR	C=1
[14] [ma] [aa]	JUMP,CS ADDR	C=1
[15] [ma] [aa]	JUMP,HS ADDR	C=0
[15] [ma] [aa]	JUMP,CC ADDR	C=0
[16] [ma] [aa]	JUMP,LS ADDR	C=1 OR Z=1
[17] [ma] [aa]	JUMP,HI ADDR	C=0 AND Z=0
[18] [ma] [aa]	JUMP,NS ADDR	N=1
[19] [ma] [aa]	JUMP,NC ADDR	N=0

Exemple :

JUMP H'120 ; [10] [01] [20] Saute à l'adresse H'120

JUMP,LO H'120 ; [14] [01] [20] Saute é l'adresse H'120 seulement si le carry est à 1

Appels de routines

Code	Instruction	Fanion
[01] [ma] [aa]	CALL ADDR	
[FF] [ma] [aa]	CALL Routines	
[02]	RET	

Exemple :

CALL _DisplayBinaryDigit ; [FF] [00] [0A] Appelle la routine interne DisplayBinaryDigit

CALL LOOP ; [01] [ma] [aa] Appelle la routine définie par le label LOOP:

Utilisation de la pile

Code	Instruction	Fanion
[08+r]	PUSH r	
[0C+R]	POP r	
[07] [vv]	SUB #val, SP	
[08] [vv]	ADD #val, SP	
[54+R] [40] [dd]	MOVE {SP}±depl, r	(N, Z)
[58+r] [40] [dd]	MOVE r, {SP}±depl	(N,Z)

Exemple :

PUSH B ; [09] Ajoute la valeur du registre B sur la pile

SUB #3, SP ; [07] [03] Soustrait 3 au registre SP

MOVE A, {SP}-8 ; [58] [40] [F8] Mémorise la valeur du registre A à l'adresse SP-8

Gestion des fanions

Code	Instruction	Fanion
[04]	SETC	C=1
[05]	CLRC	C=0
[FE]	NOTC	C

Divers

Code	Instruction	Fanion

[00]	NOP
[03]	HALT
[5C]	EX A,B
[5D]	EX X, Y
[5E]	SWAP A
[5F]	SWAP B

Exemple :

EX A, B ; [5C] Échange la valeur du registre A avec celle de B

SWAP B; [5F] Permute les 4 premiers bits avec les 4 derniers de la valeur du registre B

2.2. Routines

_WaitSec

Cette routine n'est pas implémentée dans *Dauphin Web* pour le moment. Néanmoins, pour des raisons de rétrocompatibilité, l'instruction demeure valide, mais n'a aucun effet.

[FF] [00] [00] CALL _WaitSec

_DisplayBinaryDigit

Affiche des segments à choix.

[FF] [00] [0A] CALL _DisplayBinaryDigit

in A bits des segments à allumer

B digit 0...3 (de gauche à droite)

out -
mod F

_DisplayHexaDigit

Affiche un digit hexadécimal.

[FF] [00] [10] CALL _DisplayHexaDigit

in A valeur 0...15

B digit 0...3 (de gauche à droite)

out -
mod F

_DisplayHexaByte

Affiche un octet hexadécimal sur deux digits.

[FF] [00] [0E] CALL _DisplayHexaByte

in A valeur 0...255

B premier digit 0...2 (de gauche à droite)

out -
mod F

_DsiplayCharDigit

Dessine un caractère dans l'écran digital.

[FF] [00] [10] CALL _DisplayCharDigit

in A valeur 0...127 (code ASCII du caractère)

  B premier digit 0...3 (de gauche à droite)

out -
mod F

_ClearScreen

Efface tout l'écran bitmap.

[FF] [00] [14] CALL _ClearScreen

in -
out -
mod F

_SetPixel

Allume un pixel dans l'écran bitmap.

[FF] [00] [16] CALL _SetPixel

in X coordonnée colonne 0...31

  Y coordonnée ligne 0...23

out -
mod F

_ClrPixel

Eteint un pixel dans l'écran bitmap.

[FF] [00] [18] CALL _ClrPixel

in X coordonnée colonne 0...31

  Y coordonnée ligne 0...23

out -
mod F

_NotPixel

Inverse un pixel dans l'écran bitmap.

[FF] [00] [1A] CALL _NotPixel

in X coordonnée colonne 0...31

  Y coordonnée ligne 0...23

out -
mod F

_TestPixel

Teste l'état d'un pixel dans l'écran bitmap.

[FF] [00] [1C] CALL _TestPixel

in X coordonnée colonne 0...31

  Y coordonnée ligne 0...23

out EQ si le pixel est éteint

  NE si le pixel est allumé

mod F

_DrawChar

Dessine un caractère dans l'écran bitmap. Les chiffres sont codés de 30 à 39, et les lettres de 41 à 5A.

[FF] [00] [28] CALL _DrawChar

in A caractère ascii

  X colonne 0...7

  Y ligne 0...4

out -
mod F

_DrawHexaDigit

Dessine un digit hexadécimal dans l'écran bitmap.

[FF] [00] [2A] CALL _DrawHexaDigit

in A valeur 0...15

  X colonne 0...7

  Y ligne 0...3

out -
mod F

_DrawHexaByte

Dessine un octet hexadécimal sur deux digits dans l'écran bitmap.

[FF] [00] [2C] CALL _DrawHexaByte

in A valeur 0...255

  X colonne 0...6

  Y ligne 0...4

out -
mod

2.3. Exemples de programme

Suite de Fibonacci

```
      MOVE   #0, A
      MOVE   #1, B  ; initialisation
FiboSuivant:
      MOVE   B, X   ; On sauve B (on pourrait aussi écrire PUSH B)
      ADD    A, B   ; On a la valeur suivante de B (et de la suite)
      MOVE   X, A   ; On récupère l'ancienne 2ème valeur (n-1) (POP A)
      JUMP, CC FiboSuivant      ; On boucle si le carry est à 0
      HALT            ; sinon on s'arrête. La suite a dépassé 255 (limite 8bits)
```

Hello ASCII

```
;HELLO ASCII
START:
      MOVE   #0, X              ; rang de la lettre
DISPLAY:
      MOVE   #0, Y              ; rang de l'afficheur
LOOP:
      MOVE   TABLE+{X}+{Y}, A   ; cherche la lettre
      MOVE   A, _ASCII+{Y}      ; affiche la lettre
      INC    Y                  ; afficheur suivant
      COMP   #6, Y              ; 4ème afficheur ?
      JUMP,LO      LOOP         ; non -> LOOP
                                ; oui ->
      MOVE   #3, A              ; durée à attendre
      INC    X                  ; lettre suivante
      COMP   #17-6, X           ; dernière lettre ?
      JUMP,LS      DISPLAY            ; non -> DISPLAY
      JUMP   START              ; oui -> START
      TABLE  #17                ; table de 12 octets
TABLE:
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"H"   ; lettre H
      BYTE   #"E"   ; lettre E
      BYTE   #"L"   ; lettre L
      BYTE   #"L"   ; lettre L
      BYTE   #"O"   ; lettre O
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
      BYTE   #"_"   ; espace
```

Hello DIGIT

```
;Hello_digit
START:
      MOVE   #0, X        ; rang de la lettre
DISPLAY:
      MOVE   #0, Y        ; rang de l'afficheur
LOOP:
      MOVE   TABLE+{X}+{Y}, A    ; cherche la lettre
      MOVE   A, _DIGIT0+{Y}      ; affiche la lettre
```

```
        INC    Y            ; afficheur suivant
        COMP   #4, Y        ; 4ème afficheur ?
        JUMP,LO      LOOP         ; non -> LOOP
                             ; oui ->
        MOVE   #3, A        ; durée à attendre
        INC    X            ; lettre suivante
        COMP   #12-4, X           ; dernière lettre ?
        JUMP,LS      DISPLAY            ; non -> DISPLAY
        JUMP   START        ; oui -> START
        TABLE  #12          ; table de 12 octets
TABLE:
        BYTE   #B'00000000  ; espace
        BYTE   #B'00000000  ; espace
        BYTE   #B'00000000  ; espace
        BYTE   #B'01011011  ; lettre H
        BYTE   #B'01111100  ; lettre E
        BYTE   #B'00011100  ; lettre L
        BYTE   #B'00011100  ; lettre L
        BYTE   #B'00111111  ; lettre O
        BYTE   #B'00000000  ; espace
        BYTE   #B'00000000  ; espace
        BYTE   #B'00000000  ; espace
        BYTE   #B'00000000  ; espace
```

Hello Bitmap

```
;Hello_pixel
NBLETTERS = 5
TOP:
        CLR    Y            ; en haut
START:
        CLR    B            ; rang 1ère lettre
NEXT:
        CLR    X            ; à gauche
LETTER:
        PUSH   Y
        MOVE   B, Y
        MOVE   TABLE+{X}+{Y}, A    ; lettre à afficher
        POP    Y
        CALL   _DrawChar    ; affiche une lettre
        INC    X            ; à droite
        COMP   #8, X        ; dernière position x ?
        JUMP,LO      LETTER       ; non -> LETTER
                             ; oui ->
        MOVE   #20, A
        CALL   _WAITSEC           ; attend 1 seconde
        INC    B            ; 1ère lettre suivante
        COMP   #7+NBLETTERS, B    ; dernière lettre ?
        JUMP,LS      NEXT         ; non -> NEXT
                             ; oui ->
        INC    Y            ; plus bas
        COMP   #5, Y        ; bas atteint ?
        JUMP,LO      START        ; non -> START
        JUMP   TOP          ; oui -> TOP
        TABLE  #7+NBLETTERS+8     ; 7+8 espace et lettres
TABLE:
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
```

```
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'48        ; H (selon NBLETTERS)
        BYTE   #H'45        ; E
        BYTE   #H'4C        ; L
        BYTE   #H'4C        ; L
        BYTE   #H'4F        ; O
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
        BYTE   #H'20        ; espace
```

Segment cyclique

```
        MOVE   #1, A        ; met le bit initial
LOOP:
        MOVE   A, _DIGIT0           ; allume un segment
        RR     A            ; décale le bit à gauche
        JUMP   LOOP         ; recommence à l'infini
```

Rebond1

```
        CLR    X            ; X à gauche
        CLR    Y            ; Y en haut
        MOVE   #1, B        ; de gauche à droite
LOOP:
        COMP   #_BITMAPWIDTH-1, X ; touche le bord droit ?
        JUMP,LO      RIGHT        ; non -> RIGHT
        MOVE   #-1, B       ; de droite à gauche
RIGHT:
        COMP   #0, X        ; touche le bord gauche ?
        JUMP,HI      LEFT         ; non -> LEFT
        MOVE   #1, B        ; de gauche à droite
LEFT:
        ADD    B, X         ; avance ou recule X
        CALL   _NotPixel           ; allume le nouveau point

        CALL   _NotPixel           ; éteint l'ancien point
        JUMP   LOOP
```

Rebond2

```
        CLR    X            ; X à gauche
        MOVE   #_BITMAPHEIGHT/2-1, Y ; Y au milieu
        MOVE   #1, B        ; de gauche à droite
LOOP:
        COMP   #_BITMAPWIDTH-3, X ; touche le bord droit ?
        JUMP,LO      RIGHT        ; non -> RIGHT
        MOVE   #-1, B       ; de droite à gauche
RIGHT:
        COMP   #1, X        ; touche le bord gauche ?
        JUMP,HI      LEFT         ; non -> LEFT
        MOVE   #1, B        ; de gauche à droite
LEFT:
        ADD    B, X         ; avance ou recule X
        CALL   NOTBALL             ; allume nouvelle balle

        CALL   NOTBALL             ; éteint l'ancienne balle
```

```
        JUMP   LOOP

; Inverse la balle.
; in   X position horizontale
;      Y position verticale
; out  -
; mod  F
NOTBALL:
        PUSH   X
        PUSH   Y

        DEC    Y
        CALL   _NotPixel
        INC    X
        CALL   _NotPixel
        INC    X
        INC    Y
        CALL   _NotPixel
        INC    Y
        CALL   _NotPixel
        DEC    X
        INC    Y
        CALL   _NotPixel
        DEC    X
        CALL   _NotPixel
        DEC    X
        DEC    Y
        CALL   _NotPixel
        DEC    Y
        CALL   _NotPixel

        POP    Y
        POP    X
        RET
```

Rebond3

```
        CLR    X            ; X à gauche
        MOVE   #_BITMAPHEIGHT/2-1, Y ; Y au milieu
        MOVE   #1, A        ; de gauche à droite
        MOVE   #1, B        ; de haut en bas
LOOP:
        COMP   #_BITMAPWIDTH-3, X ; touche le bord droite ?
        JUMP,LO      RIGHT        ; non -> RIGHT
        MOVE   #-1, A       ; de droite à gauche
RIGHT:
        COMP   #1, X        ; touche le bord gauche ?
        JUMP,HI      LEFT         ; non -> LEFT
        MOVE   #1, A        ; de gauche à droite
LEFT:
        COMP   #_BITMAPHEIGHT-3, Y ; touche le bord inf ?
        JUMP,LO      BOTTOM       ; non -> BOTTOM
        MOVE   #-1, B       ; de bas en haut
BOTTOM:
        COMP   #1, Y        ; touche le bord sup ?
        JUMP,HI      TOP          ; non -> TOP
        MOVE   #1, B        ; de haut en bas
TOP:
        ADD    A, X         ; avance ou recule X
        ADD    B, Y         ; descend ou monte Y
```

```
        CALL    NOTBALL             ; allume nouvelle balle

        PUSH    A
        POP     A

        CALL    NOTBALL             ; éteint l'ancienne balle
        JUMP    LOOP

; Inverse la balle.
; in    X position horizontale
;       Y position verticale
; out   -
; mod   F
NOTBALL:
        PUSH    X
        PUSH    Y

        DEC     Y
        CALL    _NotPixel
        INC     X
        CALL    _NotPixel
        INC     X
        INC     Y
        CALL    _NotPixel
        INC     Y
        CALL    _NotPixel
        DEC     X
        INC     Y
        CALL    _NotPixel
        DEC     X
        CALL    _NotPixel
        DEC     X
        DEC     Y
        CALL    _NotPixel
        DEC     Y
        CALL    _NotPixel

        POP     Y
        POP     X
        RET
```

Jeu de la vie

```
;Jeux de la vie.
;L’espace mémoire doit être réglé à 4096 octets pour ce jeu
;pour obtenir une génération après 30min, l’initialisation du motif et de la
taille de l’écran a été modifiée :
; SCREENSIZE = 96 (ligne 5)
; routine PUTMOTIF (ligne 243, fin du programme)
;       MOVE    #H'40, _BITMAP+41
;       MOVE    #H'10, _BITMAP+45
;       MOVE    #H'CE, _BITMAP+49
SCREENCOPY    = H'20F             ; adresse de la copie d'écran
SCREENSIZE    = 16          ; taille de l'écran en octets (original = 96 ; max 128)


        MOVE    #0, A
        CALL    PUTMOTIF
```

```
        CALL   SCREENTOCOPY

        MOVE   #1, X
        MOVE   #2, Y
        CALL   NEIGHBOURS

LOOP:
        MOVE   #0, X
        MOVE   #0, Y

LOOPX:
        CALL   TESTPIXEL
        COMP   #1, A
        JUMP,EQ      ITSALIVE
        CALL   NEIGHBOURS
        COMP   #3, B
        JUMP,EQ      BIRTH
        JUMP   ENDLOOPX

ITSALIVE:
        CALL   NEIGHBOURS
        COMP   #2, B
        JUMP,LO      DIE
        COMP   #3, B
        JUMP,LS      ENDLOOPX

DIE:
        CALL   CLRPIXEL
        JUMP   ENDLOOPX

BIRTH:
        CALL   SETPIXEL

ENDLOOPX:
        INC    X
        COMP   #32, X
        JUMP,LO      LOOPX
        CLR    X

        INC    Y
        COMP   #24, Y
        JUMP,LO      LOOPX

        CALL   COPYTOSCREEN
        JUMP   LOOP

; Copie l'écran vers la copie d'écran.
SCREENTOCOPY:
        MOVE   #0, X
        MOVE   #0, Y
        MOVE   #SCREENSIZE, B
LOOP2:
        MOVE   _BITMAP+{X}+{Y}, A
        MOVE   A, SCREENCOPY+{X}+{Y}
        INC    X
        DEC    B
        JUMP,NE      LOOP2
        RET
```

```
; Copie la copie d'écran vers l'écran.
COPYTOSCREEN:
      MOVE   #0, X
      MOVE   #0, Y
      MOVE   #SCREENSIZE, B
LOOP1:
      MOVE   SCREENCOPY+{X}+{Y}, A
      MOVE   A, _BITMAP+{X}+{Y}
      INC    X
      DEC    B
      JUMP,NE      LOOP1
      RET

; Compte le nombre de voisins d'un pixel à un.
; in   X;Y    coordonnées du pixel
; out  B      nombre de voisins
NEIGHBOURS:
      PUSH   X
      PUSH   Y

      MOVE   #0, B

      DEC    X
      DEC    Y

      CALL   TESTPIXEL
      ADD    A, B
      INC    X
      CALL   TESTPIXEL
      ADD    A, B
      INC    X
      CALL   TESTPIXEL
      ADD    A, B

      SUB    #2, X
      INC    Y

      CALL   TESTPIXEL
      ADD    A, B
      ADD    #2, X
      CALL   TESTPIXEL
      ADD    A, B

      SUB    #2, X
      INC    Y

      CALL   TESTPIXEL
      ADD    A, B
      INC    X
      CALL   TESTPIXEL
      ADD    A, B
      INC    X
      CALL   TESTPIXEL
      ADD    A, B

      POP    y
      POP    X
      RET
```

```
; Teste un pixel dans l'écran.
; in   X,Y    coordonnées
; out  A      1 => pixel allumé
;             0 => pixel éteint
TESTPIXEL:
       PUSH   B
       PUSH   X
       PUSH   Y

       MOVE   #0, A

       CALL   WRAPXY

       AND    #H'1F, Y
       RL     Y
       RL     Y
       MOVE   X, B
       XOR    #H'07, B
       RR     X
       RR     X
       RR     X
       AND    #H'3, X

       TEST   _BITMAP+{X}+{Y} :B
       JUMP,EQ      TESTPIXEL1
       INC    A

TESTPIXEL1:
       POP    Y
       POP    X
       POP    B
       RET

; Efface un pixel dans la copie d'écran.
; in   X,Y    coordonnées
; out  A      1 => pixel allumé
;             0 => pixel éteint
CLRPIXEL:
       PUSH   B
       PUSH   X
       PUSH   Y
       AND    #H'1F, Y
       RL     Y
       RL     Y
       MOVE   X, B
       XOR    #H'07, B
       RR     X
       RR     X
       RR     X
       AND    #H'3, X
       MOVE   #0, A
       TCLR   SCREENCOPY+{X}+{Y} :B
       POP    Y
       POP    X
       POP    B
       RET

; Allume un pixel dans la copie d'écran.
; in   X,Y    coordonnées
```

```
; out  A      1 => pixel allumé
;            0 => pixel éteint
SETPIXEL:
       PUSH   B
       PUSH   X
       PUSH   Y
       AND    #H'1F, Y
       RL     Y
       RL     Y
       MOVE   X, B
       XOR    #H'07, B
       RR     X
       RR     X
       RR     X
       AND    #H'3, X
       MOVE   #0, A
       TSET   SCREENCOPY+{X}+{Y} :B

       POP    Y
       POP    X
       POP    B
       RET

; Wrap des coordonnées.
; in   X,Y    coordonnées
; out  X,Y    coordonnées wrappées
WRAPXY:
       COMP   #31, X
       JUMP,LS      XOK

       COMP   #128, X
       JUMP,LS      XOK1
       MOVE   #31, X
       JUMP   XOK

XOK1:
       MOVE   #0, X
       JUMP   XOK

XOK:
       COMP   #23, Y
       JUMP,LS      YOK

       COMP   #128, Y
       JUMP,LS      YOK1
       MOVE   #23, Y
       JUMP   YOK

YOK1:
       MOVE   #0, Y
YOK:
       RET

; Met un motif sympa dans l'écran.
PUTMOTIF:
       CALL   _CLEARSCREEN

       MOVE   #H'40, _BITMAP+1    ; original : _BITMAP+41
       MOVE   #H'10, _BITMAP+5    ; original : _BITMAP+45
```

```
MOVE   #H'CE, _BITMAP+9    ; original : _BITMAP+49

CALL   SCREENTOCOPY
RET
```