Dauphin 2e partie

Analyse du répertoire d'instruction du Psi30 et exemples

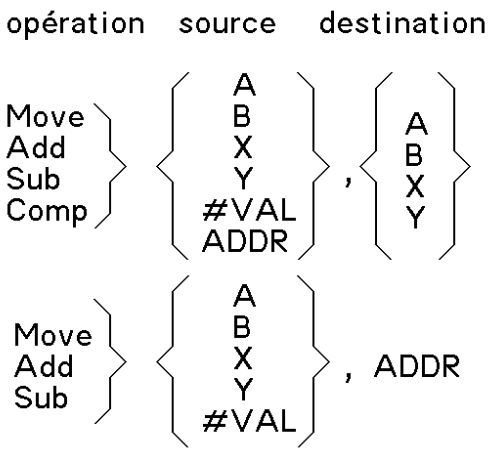
Les notations sont celles du langage CALM défini au LAMI dès 1975 pour éviter de se perdre dans la multiplicité des notations cabalistiques définies par chaque fabricant.

Rappelons que les instructions agissent sur des opérandes qui peuvent être
- une valeur, dite valeur immédiate, car elle est immédiatement disponible dans l'instruction : Move **#H'2C**,A ;
- un registre (**A,B,X,Y,F**), dont le contenu est lu ou mis à jour : Move #H'2C,**A** ;
- une position mémoire ou un périphérique (**ADDR**), analogue à un registre : Move A,**H'C00**;
- un pointeur, ou valeur indirecte, c'est-à-dire que dans le registre indiqué on trouve un nombre qui est l'adresse ou un déplacement à ajouter à l'adresse : Move **Table+{X},**B.

Le Psi30 a un joli répertoire d'instruction très « orthogonal », en général chaque instruction accepte tous les modes d'adressage.

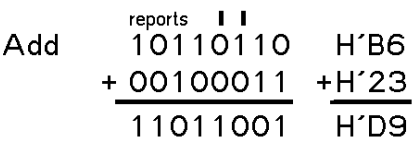
Déplacements et opérations arithmétiques

Chaque instruction a un nom qui exprime l'opération effectuée, suivi de l'opérande source et l'opérande destination. Cette notion de source-destination est évidente avec l'instruction Move. Le contenu de la source (un registre, une position mémoire ou périphérique, une valeur) et copié dans la destination (un registre, une position mémoire ou périphérique).

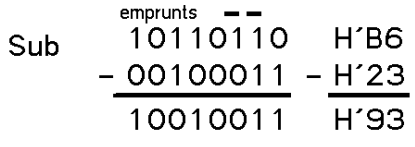


Par exemple Move #3,A
Move A,X
a pour effet de mettre la valeur 3 dans A et X. Le tableau ci-contre montre que ces deux instructions son permises. Par contre
Move #3,F n'est pas une instruction qui existe.
Move A,#3 n'a pas de sens.

L'instruction Add ajoute les deux opérandes et met le résultat dans la destination.
Add A,B ajoute les contenus de A et B et met le résultat dans B. (B)+(A) → B



Attention avec l'instruction Sub !
Sub A,B prend le contenu du 2e opérande (B) et soustrait le contenu de la source (A) . Le résultat est mis dans la destination B. (B)-(A) → B

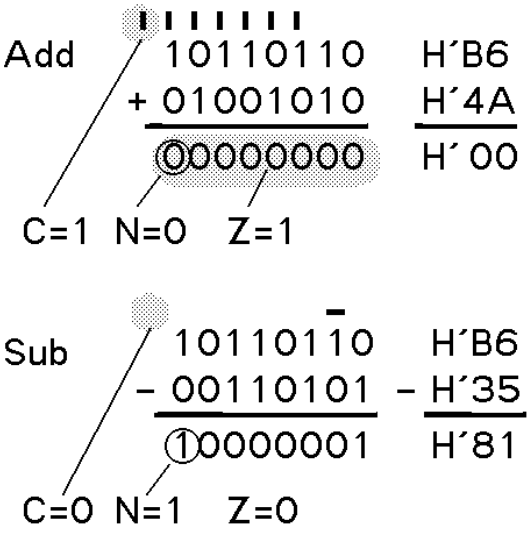


La comparaison n'est qu'une soustraction qui ne modifie pas la destination. Les deux opérandes sont inchangés, mais les fanions sont modifiés, ce qui permet d'utiliser un saut conditionnel pour tenir compte de la comparaison.

La suite des nombres de Fibonacci s'obtient en additionnant les 2 dernier nombres pour construire le suivant : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, … Mettons ces deux derniers nombres dans A et B. La suite apparaît dans B en exécutant le programme suivant :

```
        Move    #0,A
        Move    #1,B    ; on a initialisé
FiboSuivant :
        Move    B,X     ; on sauve B  (on pourrait aussi écrire Push B)
        Add     A,B     ; on a la valeur suivante dans B
        Move    X,A     ; on récupère l'ancienne 2e valeur qui devient la première (Pop A)
        Jump    FiboSuivant
```

Les Fanions



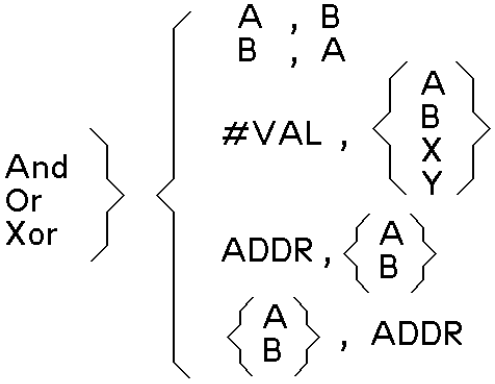
Les fanions **Z N C** sont mis à jour à la fin de l'opération.
Z (zero) est activé à 1 si le résultat de l'opération est nul.
Par exemple, **Move #0,B** va activer Z
N (negatif) est activé si le bit 7, dit «bit de signe» est à 1. Ceci concerne les nombres négatifs qui seront vus plus loin, mais cela peut être pratique de savoir que ce bit est à un (c'est le cas dans l'exemple d'addition ci-contre)
C (carry/borrow) est activé si l'addition déborde, ou que la soustraction s'est terminée par un emprunt.

Les instructions Move n'ont pas de raison de modifier le Carry. Mais les fanions N et Z sont mis à jour à la fin de chaque instruction Move.

Reprenons notre exemple du calcul des nombres de Fibonacci. Avec notre processeur 8 bits, il va y avoir rapidement débordement (résultat supérieur à 255), signalé par le carry. On va donc tester le Carry et s'arrêter quand il est à 1.

```
            Move        #0,A
            Move        #1,B        ; on a initialisé
FiboSuivant :
            Move        B,X         ; on sauve B
            Add         A,B         ; on a la valeur suivante dans B
            Move        X,A         ; on récupère l'ancienne 2e valeur qui devient la première
            Jump,CC     FiboSuivant ; on continue tant que le Carry vaut zéro
            Halt                    ; A affiche la dernière valeur (H'E9)
```

Opérations logiques



L'opération ET (**AND**) donne un bit à 1 dans la destination si les deux bits correspondant sont à 1.
On l'a utilisée en page 28 et 38 pour mettre à zéro des bits non utiles (on dit masquer ces bits).

L'opération OU (**OR**) donne un bit à 1 dans la destination si l'un des deux bits correspondant est à 1.
Par exemple pour savoir si A et B sont tous deux à zéro, on peut écrire

```
  Or   A,B ; détruit B
  Jump,EQ  TousLesDeuxAZero
```

L'opération OU exclusif (**XOR**) donne un bit à 1 dans la destination si l'un des deux bits correspondant est à 1, mais pas les deux.
Cette instruction est très utile sur un écran, pour faire passer un pointeur de souris par dessus l'information, sans la perdre.

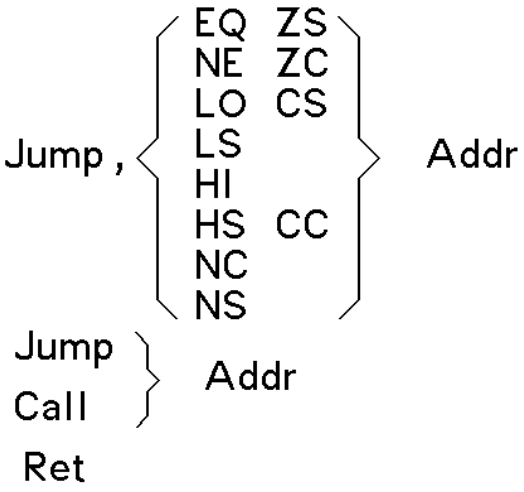
```
And   10110110   H´B6
      00100011   H´23
      --------   ----
      00100010   H´22

Or    10110110   H´B6
      00100011   H´23
      --------   ----
      10110111   H´B7

Xor   10110110   H´B6
      00100011   H´23
      --------   ----
      10010101   H´95
```

Sauts et appels

La notion de saut (**JUMP**) est simple. Dans d'autres langages cela s'appelle GoTo ou Branch. Le saut conditionnel ne se fait que si une condition, c'est-à-dire l'état 0 ou 1 d'un fanion, est vraie.
Jump,EQ Addr saute à Addr si le fanion Z=1
On peut aussi écrire Jump,ZS si c'est plus clair de penser au fanion Z plutôt qu'au résultat de l'opération précédente qui a modifié Z.
Jump,LO s'utilise après une comparaison ou soustraction pour savoir si le résultat est « lower », c'est-à-dire strictement plus petit. La soustraction active le Carry dans ce cas.
Pour savoir si le résultat est plus petit ou égal, il faut faire un deuxième saut conditionnel avec **Jump,EQ** ou **Jump,NE**.
Par contre **Jump,LS** saute si le résultat est strictement inférieur « LowerSame ».

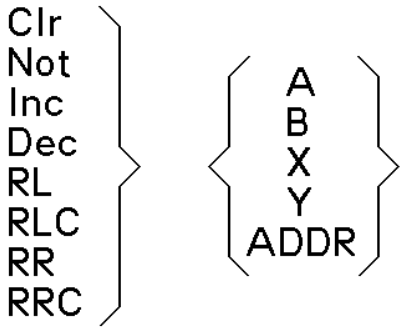


On a de même **Jump,HI** « Higher » (plus grand), **Jump,HS** « Higher or Same », **Jump,CC** « Carry Clear », **Jump,NC** « Negative Clear » (le bit de signe, le bit 7 du résultat, est à 0 » et **Jump,NS** « Negative Set ».

L'instruction Call Addr est exécutée comme un saut, mais en plus l'adresse de l'instruction suivante est mémorisée sur la pile. On saute dans un bout de programme qui doit se terminer par l'instruction RET, qui est un saut à l'adresse mémorisée, donc un retour à l'instruction suivant le Call. Ceci permet d'écrire un module de programme, appelé routine ou procédure, qui peut être appelé depuis plusieurs endroits dans le programme principal. Il n'y a pas besoin de réécrire plusieurs fois les mêmes instructions. Une routine peut appeler une autre routine.
Par exemple pour un écran, on écrira une routine qui fait aller un objet à droite, à gauche, en bas, en haut, et on appellera ces routines en fonction du jeu. Chacune de ces routines risque bien d'appeler une routine qui dessine l'objet.

Opérations à un opérande

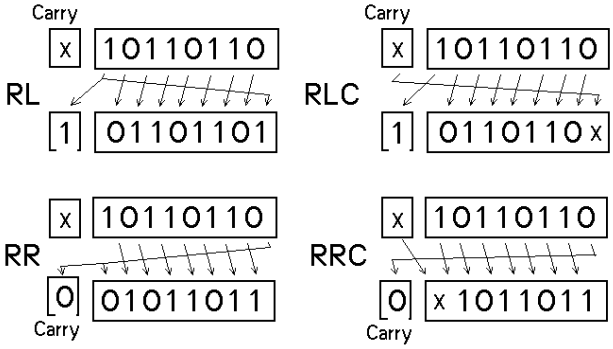
L'instruction **Clr dest** est équivalente à Move #0,dest, mais elle est plus lisible.
L'instruction **Not dest** inverse tous les bits
L'instruction **Inc dest** ajoute 1. Si le résultat est nul (le compteur a fait un tour), le fanion Z est mis à 1.
L'instruction **Dec dest** soustrait 1. Si la valeur initiale est 0, après décrémentation, le résultat est H'FF (des 1 partout).
Comment faire pour qu'un compteur se bloque en arrivant au maximum H'FF ?
On teste si le compteur est arrivé à zéro et on décompte pour annuler le comptage :




```
      Inc          A
      Jump,NE  Next   ; saute à Next si fanion Z=0
      Dec         A
Next :  suite
```

Comment faire pour qu'un décompteur se bloque à zéro ? De même pour qu'un compteur se bloque à la valeur H'7F ou à la valeur B'31 (par exemple) ? C'est ce qu'on appelle saturation.

Décalages

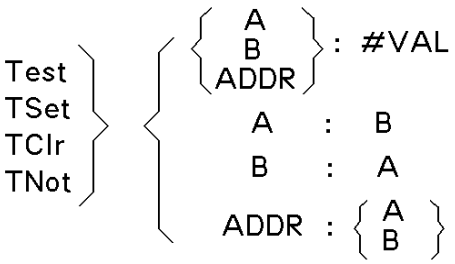


L'instruction **RL dest** « Rotate Left » fait tourner le mot de 8 bits sélectionné sur lui-même. Le bit qui fait le tour est copié dans le Carry,
RR dest « Rotate Right » fait la même chose dans l'autre sens.
Ceci permet par exemple de compter le nombre de bits à un dans le registre A.
On initialise un compteur par 8 et on décale dans la boucle. Chaque fois que le carry est à un, on incrémente un compteur.
Les instructions **RLC** « Rotate Left through Carry » traversent le Carry il faut 9 décalages pour retrouver la même valeur. **RRC** « Rotate Right through Carry » fait la même chose dans l'autre sens. Ceci est utile pour décaler des mots plus longs que 8 bits. Prenons par exemple une ligne d'écran de 32 bits. Si on écrit

```
        Move    #B'1000000,_BitMap
        Clr     _BitMap+1
        Clr     _BitMap+2
        Clr     _BitMap+3
        ClrC
Bcle :  RRC     _BitMap
        RRC     _BitMap+1
        RRC     _BitMap+2
        RRC     _BitMap+3
        Jump    Bcle
```

On voit que le bit mis à un tourne sur la première ligne d'écran. Modifiez l'initialisation et observez.

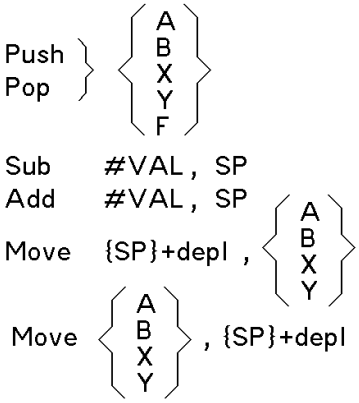
Test et modification de bits



Les instructions de test et modification de bits doivent spécifier un registre ou une position mémoire, et le numéro du bit. Cette adresse de bit (un parmi 8) est une valeur immédiate de 0 à 7, ou est le nombre dans les 3 bits de poids faible du registre A ou B (sous-adresse).
Le « deux-point » sépare les deux adresses.
L'instruction **Test** copie le bit dans le fanion Z. On peut ensuite appliquer l'instruction **Jump,EQ** ou **Jump NE**
L'instruction **TSet** teste le bit, puis le met à un. **TClr** met à zéro. **TNot** inverse.

Par exemple, comment savoir si le bit au centre de l'écran est à un et changer sa valeur ? Le centre est à la 8e ligne, bit 0 du 2e mot. L'adresse mémoire est _BitMap+8*4+2, l'adresse du bit est zéro. Il faud donc écrire **TNot (_BitMap+8*4+2):#0**. La parenthèse n'est pas nécessaire, et il ne faut pas remplacer le (8*4+2) par 34, l'assembleur sait mieux calculer que nous, et le nombre 34 n'évoque pas une coordonnée !

Pile et instructions spéciale



La pile est une zone mémoire pointée par le registre SP, qui permet d'empiler des valeurs, donc des contenus. **Push A** va mettre le contenu de A sur la pile. **Pop B** va charger ce contenu. Avec ces deux instructions, on fait presque exactement un **Move A,B**. Pourquoi presque ? Parce que Move A,B modifie le fanion Z, mais pas un Push/Pop.
La pile est initialisée en H'800 au démarrage, et descend vers le programme. Le premier Push écrit en H'7FF. Si on se trompe et qu'il y a plus de Push que de Pop, la pile va grandir et finir par écraser le programme. Pour observer la pile, cliquer au bon endroit.
Les autres instructions de ce tableau nécessitent une bonne expérience de programmation et seront vues lorsque cela sera nécessaire.

Dernières instructions

Quelques instructions faciles pour terminer.
SetC « Set Carry » **ClrC** « Clear Carry » et **NotC** « Not Carry » agissent sur le fanion C. Ces instructions sont souvent utiles quand on décale un registre et que l'on veut agir sur le carry. Essayez

```
        Clr     A
        ClrC
Bcle :  RRC     A
        NotC
        Jump  Bcle
```

Vous avez fait un compteur par 16, dit compteur en anneau.
Les instructions **Ex** « Exchange » permutent les valeurs des deux opérandes. On pourrait s'en passer, mais il faudrait 3 instructions (lesquelles ?).
L'instruction **Swap** permute les 4 premiers bits avec les 4 derniers. Très pratique quand on travaille avec des chiffres hexadécimaux ou décimaux.

```
SetC
ClrC
NotC
Ex      A,B
Ex      X,Y
Swap    A
Swap    B
Nop
Halt
```

L'instruction **Nop** aide à perdre du temps sans rien influencer sauf le pointeur d'instructions **PC** « program counter » qui est incrémenté.
L'instruction **Halt** bloque ce même pointeur d'instructions.

C'est joli tout cela, et drôlement flexible. Il manque souvent une instruction pour faire simplement ce que l'on voudrait, mais il y a toujours un moyen de la remplacer par plusieurs instructions.

Adressage indexé

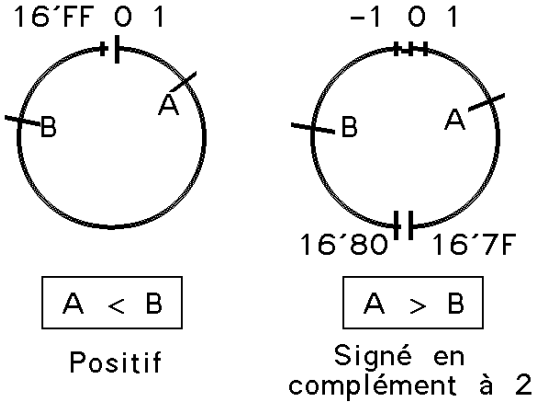
Les exemples vus jusqu'à présent utilisent l'adressage direct : on donne l'adresse en mémoire de la variable ou du saut.
Avec l'adressage indexé, l'adresse est calculée et dépend des registres X et/ou Y.
Addr + {X} est un nombre formé par la somme de l'adresse Addr, dite adresse de base, et du contenu d'un registre, appelé déplacement. Par exemple, pour se promener dans l'écran, on écrira _BitMap+{X}. Si X contient la valeur 3, _BitMap+{X} pointe l'octet à droite en haut de l'écran. On peut lire, écrire, tester cette position. Pour activer le bit tout à droite en haut de l'écran, l'instruction est **TSet _BitMap+{X} :#0**, ou si B vaut zero, **TSet _BitMap+{X} : B**
L'adressage est très utile pour parcourir une table (c'est un tableau de nombres), copier une zone mémoire dans une autre.
Par exemple pour écrire un motif sur l'écran, on peut le définir dans le programme, et le copier quand c'est nécessaire sur l'écran avec les instructions suivantes dans une boucle initialisée correctement (par exemple X ← 0 et B ← nombre de bytes à copier):

```
Move    TaProg+{X},A
Move    A,_BitMap+{X}
Inc     X
```

Nombres négatifs

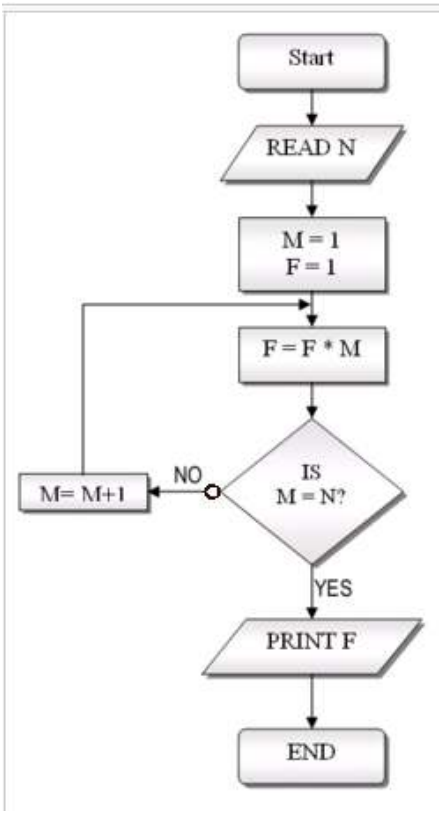
Si vous prenez une voiture neuve et que vous faites 1 km en marche arrière, le compteur va indiquer 99999 km. De même, si un registre contenant zéro est décrémenté (ou on soustrait 1), il va afficher FF. C'est la représentation naturelle de nombres négatifs, dite en complément à 2.
On peut aussi travailler avec une représentation signée, en mettant la valeur absolue dans un registre et le signe dans un autre registre.
Comment savoir si H'FF représente le nombre –1 ou le nombre 255 ? Le processeur ne peut pas vous aider. Pour lui, c'est 8 bits à un, rien de plus. Le programmeur décide et doit gérer les dépassements de capacité différemment selon qu'il a mis dans les registres des nombres positifs ou signés (positifs ou négatifs).

La meilleure représentation est de placer les nombres 8 bits sur un cercle. Avec les nombres positifs, si on passe la frontière ente FF et 00, il y a débordement que le Carry va signaler.
Avec les nombres en complément à 2, la frontière est entre 7F et 80, quand le bit 7, dit bit de signe, change.
Si on veut comparer deux nombres, il faut savoir s'ils sont positifs ou signés.
Les processeurs de plus de 8 bits ont des instructions qui facilitent la gestion des nombres négatifs, voire des nombres en virgule flottante.
Avec notre processeur simplifié, il faut tester le bit de signe N et le Carry pour décider si le résultat d'une addition ou soustraction est valide.



Quelques conseils

Avant de se lancer dans l'écriture d'un programme un peu complexe, il faut réfléchir à sa structure, à sa décomposition en modules et en routines. Un organigramme, comme l'organigramme suivant (calcul des factorielles) pris dans le document Wikipedia expliquant les organigrammes, est essentiel pour tenir compte des différents cas et avoir une vision plus globale des variables utilisées.
Dans l'organigramme ci-contre, chaque rectangle ou losange correspond à une ou deux instructions.
Si le programme est complexe, les rectangles correspondent à des routines ou des modules de programmes documentés par un autre organigramme.
Une fois la structure bien préparée, il faut nommer les variables avec des noms explicites (l'organigramme ci-contre n'est pas un bon exemple).
L'organigramme peut être remplacé par des phrases avec des Si – Alors – Faire comme dans les langages dit évolués.
Les commentaires dans les programmes sont essentiels pour pouvoir se relire et se remettre dans le bain après quelques jours ou mois, et naturellement pour permettre à un ami de s'y retrouver. Bien commenter n'est pas mettre un commentaire à chaque ligne, mais un commentaire expliquant le rôle d'un groupe d'instructions. Une ligne vide est un bon commentaire puisqu'elle peut montrer comment regrouper la lecture des instructions.



Les routines doivent pouvoir être utilisées sans que cela soit nécessaire de lire leurs instructions. Le commentaire initial doit dire la fonction réalisée, quels sont les registres et variables qui apportent l'information à traiter, quels sont les registres et variables qui rendent de l'information, quels sont les registre utilisés et modifiés. Par exemple une routine qui compte en décimal et pas en binaire se documente et s'écrit :

```
; IncBCD incrémente en décimal codé binaire
; On compte en binaire, mais si le comptage passe de 9 à A sur les 4 bits de poids faible,
; on corrige en mettant à zéro et en incrémentant les 4 bits de poids fort.
; Si la valeur sur les poids forts passe de 9 à A, on met à zéro et active le Carry
; in : A registre avec une valeur BCD de 1 à 99 :  00 01 .. 09 10 … 99
; out : A registre augmenté de 1. C=1 si passage de 99 à 00
; mod : A, B, F
  IncBCD :    Inc       A
              Move      A,B
              And       #H'0F,B      ; On filtre les 4 bits de poids faible
```

```
                Comp    #H'0A,B     ; Est-ce supérieur à 9 ?
                Jump,LO NoCorLow
                And     #H'f0,A     ; On met à zéro les 4 bits de poids faible
                Add     #H'10,A     ; On incrémente les poids fort
NoCorLow :      Move    A,B
                And     #H'F0,B     ; On filtre les 4 bits de poids fort
                Comp    #H'A0,A     ; Est-ce supérieur à 90 ?
                Jump,LO NoCorHigh   ; Saute si Carry à 1
                And     #H'0F,A     ; Clr A est plus simple puisque l'on passe
                                    ;  nécessairement de 99 à 00
NoCorHigh :     NotC                ; Carry à 1 seulement si passage de 99 à 00
                Ret
```

Routines en ROM

Aux adresses H'800 à H0BFF, le Dauphin a l'équivalent d'une ROM. Cette zone qui ne peut pas être modifiée contient des routines qui facilitent l'écriture des programmes. En 803, 806, etc, on trouve une instruction de saut à ces routines. Pourquoi ? Pour éviter de modifier l'adresse de l'appel si la routine est modifiée. C'est ce que l'on appelle une indirection.

_WaitKey

Attend la pression d'une touche du clavier.

[01] [08] [03]	CALL H'803
in	-
out	A code de la touche pressée
mod	A, F

Par exemple, pour incrémenter le registre A à chaque pression sur la touche 3, et seulement la touche 3, on écrit :

```
Debut :     Clr     A
Bcle :      Push    A       ; A sera modifié
Bcle2 :     Call    _WaitKey
            Comp    #3,A
            Jump,NE Bcle2 ; Attention à la pile
            Pop     A
            Inc     A
            Move    A,B
            Jump    Bcle
```

_DisplayBinaryDigit

Affiche des segments à choix.

[01] [08] [09]	CALL H'809
in	A bits des segments à allumer B digit 0..3 (de gauche à droite)
out	-
mod	F

Cette routine ne modifie que F. Elle masque B pour ignorer les bits de poids fort, et sauve B et X sur la pile.
Si X est libre, et que l'on est sûr que le contenu de B ne dépasse pas 3, on peut écrire plus simplement:

```
DisBinDigit :   Move    B , X
                Move    A , _Digit0+{X}
                Ret
```

_DisplayHexaByte

Affiche un octet hexadécimal sur deux digits.

[01] [08] [0F]	CALL H'80F
in	A valeur 0..255 B premier digit 0..2 (de gauche à droite)
out	-
mod	F

Affichons un compteur qui incrémente à chaque action sur une touche 0..7

```
Debut:  Clr     A
        Move    #2,B
        Call    _DisplayHexaByte
Bcle:   Push    A
        Call    _WaitKey ; modifie A
        Pop     A
        Inc     A
        Call    _DisplayHexaByte
        Jump    Bcle
```

_ClearScreen

Efface tout l'écran bitmap.

[01] [08] [1E]	CALL H'81E
in	-
out	-
mod	F

Le programme Dauphin efface l'écran et les variables au démarrage du programme, ce qui n'est pas le cas des systèmes microprocesseurs. _ClearScreen n'est donc pas nécessaire dans l'initialisation, on l'utilise par exemple pour recommencer une partie après avoir dessiné dans l'écran.

_SetPixel

Allume un pixel dans l'écran bitmap.

[01] [08] [15]	CALL H'815
in	X coordonnée colonne 0..31 Y coordonnée ligne 0..23
out	-
mod	F

Pour tracer une verticale au centre de l'écran, on écrira :

```
SegVert:    Move    #31/2,X ; arrondi en dessous
            Move    #0,Y
            Move    #23,A
Des:        Call    _SetPixel
            Inc     Y
            Dec     A
            Jump,NE Des
            Halt
```

_NotPixel

Inverse un pixel dans l'écran bitmap.

[01] [08] [1B]	CALL H'81B
in	X coordonnée colonne 0..31
	Y coordonnée ligne 0..23
out	-
mod	F

Pour clignoter un pixel dans l'écran on appelle alternativement _NotPixel et une routine Delai.

_TestPixel

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

[01] [08] [2A]	CALL H'82A
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

S'il faut compter les voisins d'un pixel, comme dans le jeu de la vie, il est difficile de faire des boucles. Les pointeurs X et Y doivent être incrémentés - décrémentés en testant chaque fois le pixel sélectionné. Si la condition est NE, on compte dans A par exemple.

_DrawChar

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

[01] [08] [21]	CALL H'821
in	A caractère ascii
	X colonne 0..7
	Y ligne 0..3
out	-
mod	F

Cette routine appelle un générateur de caractère de 3x5 pixels et place les caractères sur un grille de 4 lignes de 8 caractères.
Le code des caractères et le code Ascii, mais tous les caractères et signes ne sont pas disponibles.

La routine suivante est facile à générer si elle n'existait pas. Mais il faut séparer l'affichage des chiffres hexa de A à F :

```
DrawHexaDigit:
        Add     #H'30,A
        Comp    #H'3A,A
        Jump,Lo Ok
        Add     #H'41-H'3A,A
 Ok:    Call    _DrawChar
        Ret
```

_DrawHexaDigit

Dessine un digit hexadécimal dans l'écran bitmap.

[01] [08] [24]	CALL H'824
in	A valeur 0..15
	X colonne 0..7
	Y ligne 0..3
out	-
mod	F

Affichons les chiffres de 0 à F sur 2 lignes pour vérifier cette routine, ou ce que nous avons écrit ci-dessus.

```
DrawAllDigits:
        Move    #0,A
        Move    #0,X
        Move    #1,Y
        Move    #8,B
 Bcle1: Call    _DrawHexaDigit
        Inc     A
        Inc     X
        Dec     B
        Jump,NE Bcle1
        Move    #2,Y   ; ou   Inc Y
        Move    #8,B
 Bcle2: Call    _DrawHexaDigit
        Inc     A
        Inc     X
        Dec     B
        Jump,NE Bcle2
        Halt
```

_DrawHexaByte

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

[01] [08] [27]	CALL H'827
in	A valeur 0..255
	X colonne 0..6
	Y ligne 0..3
out	-
mod	F

Pour afficher un score au bas de l'écran, on écrira :

```
\Routine score en bas à gauche
\in:  A score en BCD ou hexa
\mod: F
Score:  Push    X
        Push    Y
        Move    #0,X
        Move    #3,Y
        Call    _DrawHexaByte
        Pop     Y
        Pop     X
        Ret
```

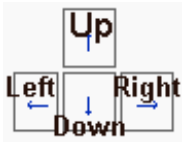
Noms réservés

En plus des noms des routines en ROM, l'assembleur connaît les noms suivant (c'est-à-dire leur assigne une valeur). Ils peuvent être écrits en minuscules.

```
_DIGIT0          = H'C00
_DIGIT1          = H'C01
_DIGIT2          = H'C02
_DIGIT3          = H'C03
_DIGITCOUNT      = D'4

_BITMAP          = H'C80
_BITMAPWIDTH     = D'32
_BITMAPHEIGHT    = D'24

_KEYBOARD        = H'C07    ;
Adresse
_KEYBOARDSHIFT   = D'3
_KEYBOARDCTRL    = D'4
_KEYBOARDDOWN    = D'3
_KEYBOARDUP      = D'4
_KEYBOARDLEFT    = D'5
_KEYBOARDRIGHT   = D'6
_KEYBOARDFULL    = D'7
```



Clavier PC

A noter que le clavier du Dauphin a été complété par les touches flèches du PC pour permettre des jeux interactifs. A l'adresse _Keyboard, les bits 3 et 4 sont à la fois sur les touches Shift et Ctrl du clavier du Dauphin (il faut les cliquer avec la souris, ce qui n'est pas rapide) et sur les touches flèches Down/Up du clavier du PC. Les touches Left/Right sont sur les deux bits qui restent.

Exemples de programmes

Chenillard

Un chenillard de bits sur l'écran utilise les instructions de rotation. Un chenillard sur l'affichage doit décaler des adresses mémoire. Relisez le programme de la page 35 de la brochure. Il doit être facile à comprendre maintenant.

Nombres aléatoires

Pour lancer un dé, c'est à dire générer un nombre au hasard entre 1 et 6, on ne peut pas compter sur un programme : il démarrera toujours de la même façon et générera la même séquence de nombres, même avec le programme le plus compliqué que l'on peut imaginer. Il faut un signal extérieur dont la durée ne peut pas être prévue, comme presser sur une touche. Programmons un compteur par 6 qui compte à toute vitesse en surveillant le clavier. Si la touche Shift est pressée, le compteur s'arrête et affiche une valeur difficile à influencer si le compteur va 100 fois plus vite que nos réflexes.

```
\prog;Dés Affiche un nombre de 1 à 6 sur le premier digit quand on pèse sur la touche Shift
; On compte de 1 à 6 si la touche Shift est relâchée
Des :    Move      #1,A          ; initialisation
Bcle :   Test      _Keyboard:#3  ; Touche shift
         Jump,NE   Cnt
         Jump      Bcle          ; touche relâchée Z=0
Cnt:     Inc       A             ; compte par 6
         Comp      #7,A
         Jump,NE   Bcle          ; c'est plus petit que 7
         Jump      Des           ; on remet 1
```

Calculatrice hexa + / -

Un joli programme pour se faire la main. Que faire si le résultat d'une soustraction est négatif ? Signaler une erreur ? Donner le résultat en complément à 2 ? Afficher un signe et une valeur absolue ?

Multiplication

C'est facile de multiplier par un petit nombre : on fait des additions successives. Une multiplication de 8 bits par 8 bits donne un résultat de 16 bits. Une succession de décalages et addition conduit au résultat. Si on sait faire l'opération à la main (c'est comme à l'école, mais avec seulement de 0 et de 1), on sait la programmer .

Division

La division est très délicate. Souvent on peut la remplacer par une multiplication, avec un résultat en général approximatif, Par exemple, pour diviser par 31, on peut multiplier par 256/31 = 8,26. On va multiplier par 8 et faire l'opération en 16 bits. L'octet de poids fort donne un résultat approché.

Bin-BCD

La conversion de binaire en BCD se fait en passant par une table qui contient les puissances de 2. On décale le nombre binaire en pointant sur cette table et en additionnant quand le bit est à 1.

Attention, il faut additionner en BCD, ce qui est un peu plus compliqué que l'incrémentation BCD vue auparavant.

BCD-Bin
La conversion de BCD en binaire se fait le plus facilement et le plus rapidement avec une table qui donne les valeurs binaires de 10, 20, .. 90. L'instruction Swap est utile pour préparer l'adresse dans cette table, et ensuite additionner les unités.

Copie d'une zone
Trivial avec deux pointeurs. Si les tables se recouvrent, il faut partir depuis la bonne extrémité.

Compter les occurrences
Comment compter le nombre de pixels à 1 dans l'écran ? On peut tester un bit après l'autre, et incrémenter un compteur si le bit est à 1. Pour aller plus vite, on lit une adresse, mais on regarde les 4 bits de poids faible puis les 4 bits de poids fort. En passant par une table qui contient le nombre de bits à un dans les chiffres de 0 à F, on obtient le résultat avec des additions binaires ou BCD, à vous de choisir.

Trier
Il y a plusieurs algorithmes, c'est passionnant. Cherchez « Algorithme de tri » sous Google.

Idées de programme

Horloges
Compteur de réflexe
Dé électronique affiché sur l'écran
PacMan
MasterMind
Puissance 4
Nim (jeu des allumettes)

Concours
EPSITEC récompensera tous les programmes intéressants. Annoncez-vous à epsitec@epsitec.ch

Comment continuer
Passer sur un microprocesseur réel se fait sans difficulté. Un document est en préparation pour montrer les différences et apprendre à programmer le PIC 16F877. Un kit permet alors de programmer en assembleur Calm pour des moteurs et des capteurs, et de construire toutes sortes d'applications, pas seulement des robots.
Une autre possibilité est d'apprendre à programmer en C/Java avec le simulateur CeeBot d'EPSITEC (www.ceebot.com). Les exercices CeeBot pilotent des robots simulés.

Jdn 070929