Les machines IMAGSYS

Description et utilisation

Bernard Cassagne
Laboratoire CLIPS — Institut IMAG
Université Joseph Fourier & CNRS
Grenoble

sept 1997

Chapitre 1

Présentation des machines

1.0.1 Position du problème

Pour les besoins de l'enseignement du système, il faut disposer de "machines nues" c'est à dire de machines sans système, puisque le but est précisément de faire des expérience avec le système. Il n'est donc pas possible de faire ces essais sur des machines partagées telles que peuvent l'être des machines UNIX, sur lesquelles à un instant donné travaillent n personnes. Les $n-1$ personnes apprécieraient très peu de perdre leur travail en cours sous prétexte que la n^{ime} fait son TP système !

Un autre contrainte est l'obligation de connaître très précisément l'architecture de la machine. En effet, impossible d'écrire un driver de périphérique si on ne connaît pas l'adresse à laquelle est implanté le coupleur qui gère le périphérique.

Pour satisfaire cette contrainte de connaissance complète du matériel, il faut s'adresser à des fournisseurs de sous-ensembles qui vendent aux entreprises qui désirent construire elles mêmes leur machine pour satisfaire un besoin spécifique. En effet, chaque sous-ensemble est vendu avec une documentation complète de son architecture interne car sinon il serait inutilisable.

1.0.2 Les choix que nous avons faits

Nous avons choisi comme processeur un membre de la famille 68000 car l'enseignement de l'assembleur se fait sur ce processeur, et nous avons choisi le bus VME qui est un bus MOTOROLA mais non propriétaire, et qui est un choix naturel quand on a fait le choix de la famille 68000.

1.0.3 La construction des machines

Pour obtenir une machine minimale, il faut assembler :

- une carte possédant un processeur, de la mémoire et quelques coupleurs d'entrées/sorties.
- un rack pour mettre les cartes.
- un fond de panier pour réaliser la connexion entre les cartes (dans notre cas, il implémentera un bus VME).
- une alimentation.

En ce qui concerne la carte processeur, nous voulions une carte comportant un membre quelconque de la famille 68000, une taille de mémoire raisonnable, deux lignes synchrones, un timer et une connexion ethernet. De nombreux fabricants proposent des cartes satisfaisant nos contraintes, et notre choix s'est porté sur une carte fabriquée par MOTOROLA : la MVME 162.

En ce qui concerne le rack, nous avons choisi un rack à 3 positions, (il nous reste donc 2 places libres) offrant un compromis raisonnable entre le coût et les possibilités d'extension de la machine.

Nous avons acheté le matériel et fait faire l'assemblage par un prestataire de service extérieur à l'école.

Chapitre 2

Utilisation des machines

2.1 Généralités

Les machines IMAGSYS ne possèdent pas de mémoire de masse, elles ne possèdent donc aucun des outils habituels nécessaires à la création de programmes : compilateur, éditeur de liens etc. Développer des programmes pour de telles machines ne peut se réaliser que si l'on dispose d'une autre machine, dite machine de développement, qui elle, comporte ces outils.

2.1.1 Un peu de vocabulaire

Quand on développe sur une machine des programmes devant s'exécuter sur une autre machine, on dit que l'on fait du « développement croisé ». Les outils de développement que l'on utilise s'appelle « outils de développement croisés » (les outils habituels, ceux qui génèrent des programmes s'exécutant sur la même machine s'appellent « outils de développement natifs ».

La machine sur laquelle on fait le développement s'appelle « machine hôte » (sous-entendu : hôte des outils de développement). La machine sur laquelle s'exécutent les programmes s'appelle « machine cible».

Dans le cas présent, la machine hôte sera un serveur UNIX et la machine cible sera une IMAGSYS.

2.1.2 Le travail du programmeur

En développement natif, le cycle habituel du programmeur est le suivant :

- édition du programme source ;
- compilation - édition de liens ;
- exécution.

en développement croisé, ce cycle se complexifie :

1. Sur la machine hôte :
   - édition du programme source ;

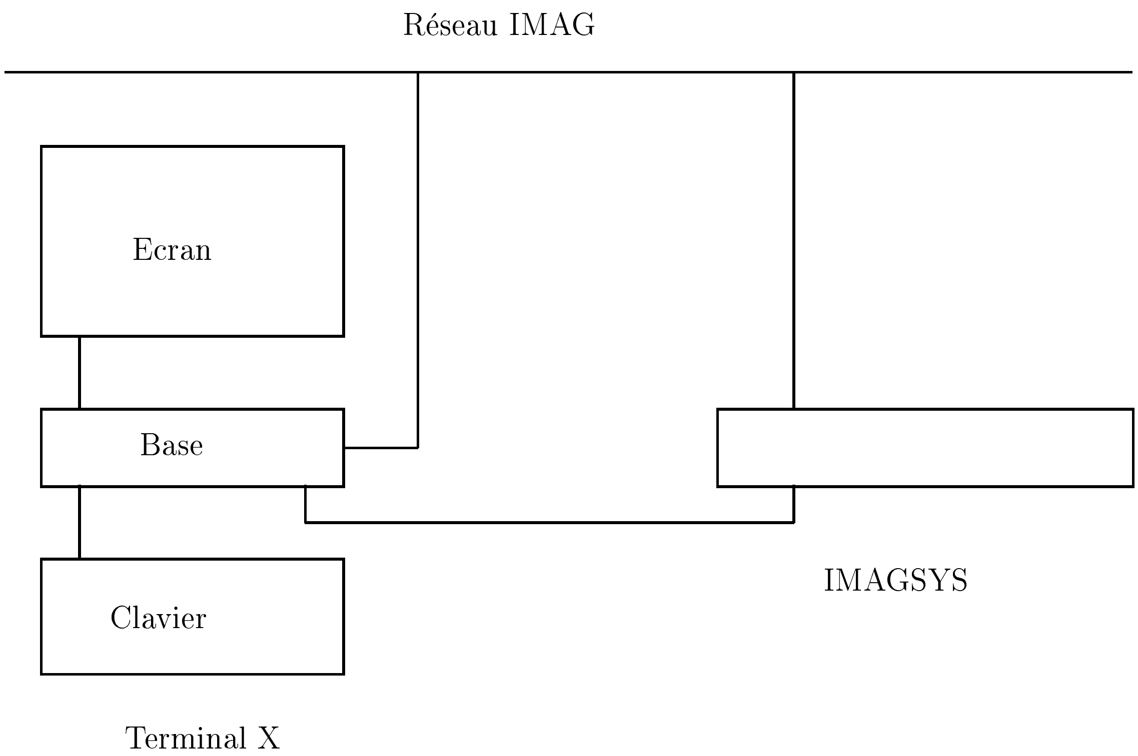
– compilation - édition de liens ;
– transformation du fichier exécutable en un format convenable pour la machine cible.

2. Sur la machine cible :
   – transfert du programme exécutable ;
   – exécution.

La machine hôte et la machine cible doivent donc être reliés d'une manière ou d'une autre pour permettre le transfert du programme exécutable.

2.2 Le poste de travail sur IMAGSYS

Le poste de travail sur une IMAGSYS se compose d'un terminal X et de l'IMAGSYS à laquelle il est connecté. Le terminal et l'IMAGSYS sont connectés ensemble, et tous deux sont connectés au réseau selon le schéma ci-desous.



Une IMAGSYS a donc deux connexions :
– une connexion à un terminal X permettant le travail interactif ;
– une connexion réseau (et par la même une connexion avec la machine hôte) pour le transfert de programme.

2.3 Prise en main d'une IMAGSYS

1. Choisir un terminal X connecté à une IMAGSYS et se connecter sur une machine UNIX de manière à disposer d'un window manager.

2. Mettre l'IMAGSYS sous tension.

3. Appuyer sur le bouton « Setup » du terminal de manière à faire apparaitre une fenêtre intitulée « NCD User Services: Console ».

4. Cliquez sur l'option « Terminals » de la barre de menus de cette fenêtre, ceci fait apparaitre un sous-menu.

5. Dans ce sous-menu choissez l'option « New Serial » qui fait apparaitre une fenêtre intitulée « NCD User Services: Terminal Emulator ». Un message s'affiche dans cette fenêtre : `Connecting to serial port....success.`

   Si ça ne marche pas, vérifiez qu'il y a bien un cable de connexion entre la base du terminal X et le port intitulé « console 1 » sur l'IMAGSYS.

6. Tapez un Retour Chariot dans la fenêtre, vous devez obtenir le prompt : `162-Bug>`.

Vous êtes alors alors en communication avec `16xBug` le moniteur de l'IMAGSYS. Ce moniteur joue vis à vis d'une IMAGSYS le même rôle que joue un *shell* vis à vis d'une machine UNIX. Les commandes du moniteur 16xBug sont décrites en détail au chapitre 3.

2.4 Les outils de développement croisés

Les outils utilisés pour travailler sur les IMAGSYS sont les outils associés au compilateur C de GNU. Il s'agit des commandes suivantes :

`ar`	`g++`	`ld`	`protoize`	`strip`
`as`	`gasp`	`nm`	`ranlib`	`unprotoize`
`c++`	`gcc`	`objcopy`	`size`	
`c++filt`	`gdb`	`objdump`	`strings`	

Ces outils existent sur les serveurs UNIX sous deux formes : la forme native et la forme croisée pour la famille de processeurs Motorola 68000. Pour permettre la coexistence des deux formes de chaque programme, les versions croisées ont pour suffixe `68`. Les noms des versions croisées sont donc les suivants :

`ar68`	`g++68`	`ld68`	`protoize68`	`strip68`
`as68`	`gasp68`	`nm68`	`ranlib68`	`unprotoize68`
`c++68`	`gcc68`	`objcopy68`	`size68`	
`c++filt68`	`gdb68`	`objdump68`	`strings68`	

Les programmes essentiels de cette famille sont :

- `gcc68` : le compilateur ;

- `as68` : l'assembleur ;

- `ld68` : l'éditeur de liens ;
- `ar68` : gestionnaire de librairie ;
- `nm68` : donne les adresses des variables globales et des fonctions d'un exécutable ;
- `objdump68` : donne en clair le contenu d'un exécutable ;
- `objcopy68` : traduit un exécutable en différents formats.

2.4.1 La Compilation

Pas de différence avec la compilation native, seul le compilateur à appeler change :

```
gcc68 -c essai.c
```

ceci produira le fichier `essai.o`.

2.4.2 L'édition de liens

Problème de `crt0`

Dans le système UNIX la convention de liaison entre le système et un programme utilisateur est la suivante : une fois chargé dans un espace d'adressage virtuel, l'exécution du programme débute à l'adresse (virtuelle) 0. Le modèle offert par le langage ANSI C est différent : l'exécution d'un programme est lancée par l'exécution de la fonction `main`.

L'implémentation du modèle ANSI C sur une machine UNIX est réalisé à l'aide d'un petit programme implanté à l'adresse 0, que l'on nomme traditionnellement `crt0`, qui implémente l'algorithme suivant :

- initialisation de l'environnement d'exécution ;
- `jsr main`
- terminaison propre du programme (appel à `exit()`).

Dans le cas d'un développement natif, le compilateur appelle l'éditeur de liens en lui faisant implanter un `crt0` convenable en début du programme. Dans le cas d'un développement croisé, c'est à la charge de l'utilisateur de fournir un `crt0` adapté à la machine cible. En ce qui concerne les IMAGSYS, en plus du `jsr main`, le strict minimum est d'initialiser le pointeur de pile. Voici un exemple de `crt0` adapté aux IMAGSYS :

```
/*
        Version de crt0.s de base
*/
        .globl  main

Lreturn =       0x63

        move.l  #0x400000,%a6
        move.l  #0x400000,%a7
        jsr     main
```

```
        trap    #15
        .word   Lreturn

        .globl __main
__main:
        rts
```

La présence de la procédure vide `__main` permet de ne pas avoir d'erreur à l'édition de liens.

Description de la mémoire

Dans le cas d'un développement natif, la description de la mémoire est fournie à l'éditeur de liens. Dans le cas d'un développement croisé, l'utilisateur doit fournir cette description.

Vous touverez ci-dessous un exemple d'une telle description convenable pour les IMAG-SYS (la RAM va de 0 à $400000_{[16]}$, mais 16xBug utilise pour son propre compte la région allant de 0 à $10000_{[16]}$).

```
/*****************************************************************************/
/*                                                                           */
/*   directives.ld simplissime :                                             */
/*     - on definit la RAM                                                   */
/*     - on met les sections                                                 */
/*         +  .text (les instructions)                                       */
/*         +  .data (les donnees initialisees)                               */
/*         +  .bss  (les donnees non initialisees)                           */
/*       de maniere contigue dans la RAM                                     */
/*                                                                           */
/*****************************************************************************/
MEMORY {
  ram : org = 0x10000, len = 0x3F0000
}

SECTIONS {
    .text  : {
        *(.text)
     } > ram
     .data : {
        *(.data)
     } > ram
     .bss : {
        *(.bss) [ COMMON ]
     } > ram
}
```

On supposera pour la suite, que ces directives ont été mises dans un fichier de nom `directives.ld`.

La commande d'édition de liens

Une fois que l'on a crée un `crt0.s` convenable, qu'on l'a assemblé pour produire un `crt0.o` et que l'on a crée un `directives.ld`, la commande d'édition de liens est :

```
ld68 -T directives.ld crt0.o essai.o -o essai.exe
```

Attention à bien mettre `crt0.o` en premier dans la liste des fichiers de manière à ce qu'il soit placé en tête du programme exécutable.

2.4.3 Transformation du format de l'exécutable

Il existe un très grand nombre de formats de fichiers exécutables. Rien que dans le monde UNIX, il existe les formats suivants : `a.out`, COFF, ELF. Les outils de développements croisés que vous utilisez génèrent du format COFF que le moniteur 16xBug ne sait pas interpréter. Après la phase d'édition de liens, il faut donc transformer le format de l'exécutable en un format utilisable par 16xBug, ce que l'on fait par un appel à `objcopy68` :

```
objcopy68 --output-target=binary essai.exe essai.bin
```

Le format `binary` est une image de la mémoire en binaire.

2.5 Chargement d'un programme

Le chargement d'un programme se fait depuis une machine de développement par l'intermédiaire du réseau. On utilise la commande `nbh` de 16xBug qui utilise le protocole tftp pour charger en mémoire le contenu d'un fichier se trouvant sur la machine de développement.

2.5.1 Organisation de la machine de développement

Les programmes à charger sur les IMAGSYS résident dans le directory `/tftpboot/IMAGSYS`. A l'intérieur de ce directory, il y a un directory par IMAGSYS. Chaque directory a pour nom le numéro d'une IMAGSYS. A l'intérieur de chacun de ces directories se trouve un fichier de nom `bootfile` : c'est ce fichier qui sera chargé en mémoire de l'IMAGSYS. Donc, l'IMAGSYS qui a pour nom IMAGSYS3 doit charger le programme `/tftpboot/IMAGSYS/3/bootfile`. Si vous travaillez sur IMAGSYS3, pour transférer un fichier en mémoire, vous devez donc d'abord copier ce fichier dans `/tftpboot/IMAGSYS/3/bootfile`.

2.5.2 Utilisation de la commande `nbh`

Une fois que le fichier que l'on désire charger a été mis en place, on passe à la fenêtre de communication avec 16xBug et on tape la commande `nbh`. On obtient alors une interaction identique à celle présentée ci-dessous (réalisée avec IMAGSYS15) :

```
162-Bug>nbh
Network Booting from: VME162, Controller 0, Device 0
Loading: IMAGSYS/15/bootfile

Client IP Address      = 195.221.227.144
```

```
Server IP Address      = 195.221.228.3
Gateway IP Address     = 195.221.227.129
Subnet IP Address Mask = 255.255.255.224
Boot File Name         = IMAGSYS/15/bootfile
Argument File Name     =

Network Boot File load in progress... To abort hit <BREAK>
```

Puis au bout de quelques secondes :

```
Bytes Received =&20, Bytes Loaded =&20
Bytes/Second   =&20, Elapsed Time =1 Second(s)
PC   =00010000 SR   =2700=TR:OFF_S._7_....
VBR  =00000000 USP  =00010000 MSP  =00010000 ISP* =00010000
SFC  =1=UD     DFC  =1=UD     CACR =00000000=D:._I:.
D0   =00000000 D1   =00000000 D2   =00000000 D3   =00000000
D4   =49504C2C D5   =00000000 D6   =00000000 D7   =00000000
A0   =FFF46000 A1   =00010000 A2   =000060B0 A3   =00005FB0
A4   =00005FB0 A5   =00005EB0 A6   =00005EC3 A7   =00010000
00010000 494D              DC.W        $494D
162-Bug>
```

Si ça ne marche pas

Il faut vérifier que l'IMAGSYS est bien configurée. Supposons que vous travailliez avec IMAGSYSx (x est son numéro) :

1. vérifiez que le fichier chargé est correct. C'est ce qui est indiqué à la fois dans `Loading:` et `Boot File Name`. La bonne valeur est `IMAGSYS/x/bootfile`.
2. Vérifiez l'adresse du client : c'est la valeur de `Client IP Address`. Pour cela, sur la machine de développement, utilisez la commande `nslookup`. Exemple :

   ```
   ensisun(1) nslookup imagsys15
   Server:  localhost
   Address:  127.0.0.1

   Name:    imagsys15.imag.fr
   Address:  195.221.227.144
   ```

3. Vérifiez l'adresse du serveur : c'est la valeur de `Server IP Address`. Ce doit être l'adresse IP de la machine de développement. Là aussi, pour la connaître, utilisez `nslookup`.
4. Vérifiez l'adresse de la passerelle : c'est la valeur de `Gateway IP Address`. Ce doit être `195.221.227.129`.
5. Vérifiez le masque de sous-réseau : c'est la valeur de `Subnet IP Address Mask`. Ce doit être `255.255.255.224`.

Si la configuration de l'IMAGSYS est incorrecte prévenez par mail `Catherine.Cassagne@imag.fr`.

2.6 Résumé des étapes de développement

On suppose que l'on développe un programme écrit en C se trouvant dans le fichier `essai.c`

Sur la machine hôte

1. Compilation :

   ```
   gcc68 -c essai.c
   ```

2. Édition de liens :

   ```
   ld68 -T directives.ld crt0.o essai.o -o essai.exe
   ```

3. Transformation en image mémoire :

   ```
   objcopy68 --output-target=binary essai.exe essai.bin
   ```

4. Mise en place pour le transfert :

   ```
   cp essai.bin /tftpboot/IMAGSYS/13/bootfile
   ```

Sur la machine cible

1. Transfert :

   ```
   nbh
   ```

2. Exécution :

   ```
   go
   ```

Chapitre 3

Les commandes de 16xBug

3.1 Généralités sur les commandes

Le programme 16xBug se trouve en ROM et constitue le seul programme disponible après avoir mis le courant. Un tel programme permet essentiellement de charger un programme et de le déboguer.

3.1.1 Les caractères de contrôle du dialogue

Il y a 6 caractères de contrôle du dialogue entre l'utilisateur et le programme 16xBug :

Control-H Efface le dernier caractère tapé.

Delete Même fonction.

Control-X Efface le début de ligne déja tapé.

Control-D Réaffiche le début de ligne déja tapé.

Control-S Stoppe l'affichage sur l'écran.

Control-Q Reprend l'affichage sur l'écran.

3.1.2 Les unités syntaxiques

Nous définissons dans cette section, différents éléments syntaxiques qui seront utilisés par les définitions des commandes de 16xBug.

Les unités syntaxiques de base

On définit les éléments suivants :

delim délimiteur : un espace ou le caractère `,`.

chaîne suite de caractères entourés de `'`.

nombre suite de “chiffres”. Les nombres peuvent être exprimés selon l'une quelconque des bases suivantes : 2, 8, 10, 16. L'indication de la base est un signe conventionnel placé devant le nombre.

base	indicateur	exemple
16	`$`	`$1f3c`
10	`&`	`&192`
8	`@`	`@132`
2	`%`	`%101011`

La base par défaut est la base 16. Pour 16xBug, `10` signifie donc 16.

expression ensemble de nombres combinés par des opérateurs. Attention : **pas de priorités d'opérateurs** et les expressions sont évaluées de la gauche vers la droite. Par exemple `1+2*3` vaut 9 !!

Par contre, on peut utiliser des parenthèses, pour obtenir le $1 + 2 * 3$ des mathématiciens, on peut donc écrire : `1+(2*3)`.

Les opérateurs possibles sont les suivants :

opérateur	sémantique
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`>>`	décalage droit
`<<`	décalage gauche
`&`	et logique

Les opérandes des *expressions* sont des nombres ou des *chaînes*. Quand un opérande est une *chaîne*, sa valeur est le nombre obtenu par concaténation des valeurs ASCII des caractères de la chaîne. Exemples :

chaîne utilisée en opérande d'expression	valeur
`'A'`	`41`
`'AB'`	`4142`

L'unité syntaxique *adresse*

Pour définir une adresse, nous introduirons les notations supplémentaires suivantes :

An Une notation assembleur pour un registre d'adresse du MC68000. (ex : `A5` ou `A6`).

Xn Une notation assembleur pour un registre d'index du MC68000. (ex : `A2` ou `D3`).

Une adresse peut avoir l'une des formes suivantes :

adresse	sémantique
expression	la valeur de *expression*
expression `+ Rn`	contenu de `Rn` + valeur de *expression*
`(An)`	contenu de `An`
expression`(An)`	contenu de An + valeur de *expression*
expression`(An,Xn)`	contenu de An + contenu de Xn + valeur de *expression*

L'unité syntaxique *intervalle*

Un *intervalle* a la forme syntaxique suivante :

adresse$_1$ *delim* *adresse*$_2$

ou :

adresse : *expression*

Un *intervalle* permet de définir une zone de mémoire dont les adresses vont de *adresse*$_1$ à *adresse*$_2$ dans la première forme, et de *adresse* à *adresse* + *expression* dans la seconde.

L'unité syntaxique *taille*

Certaines commandes peuvent nécessiter de préciser que la taille d'un paramètre est soit un byte, un mot ou un long mot. Elles utilisent pour cela une *taille* qui a l'une des 3 formes suivantes :

`;  B`
`;  W`
`;  L`

Lorsque le paramètre *taille* est optionnel, c'est la valeur `W` qui est prise par défaut.

3.2 Commande `bf` : remplissage de zone mémoire

Syntaxe :

`bf` *intervalle* *delim* *expression*$_1$ *expression*$_2$ *taille*$_{option}$

Sémantique :

- *intervalle* définit la zone de mémoire que l'on désire remplir.
- *expression*$_1$ est la valeur avec laquelle on remplit la zone de mémoire.
- *expression*$_2$ est un incrément dont est incrémenté *expression*$_1$ après chaque écriture.

Exemples :

`bf 10000,11000 2ef4`
remplit des mots consécutifs avec la valeur `2ef4`.
`bf 10000,11000 2ef4;l`
remplit des longs mots consécutifs avec la valeur `00002ef4`.
`bf 10000,1003f 'a' 1;b` remplit avec les codes des caractères A, B, C, etc...

3.3 Commande `bm` : recopie de zone mémoire

Syntaxe :

`bm` *intervalle* *delim* *adresse* $taille_{option}$

Sémantique :

La commande `bm` recopie la zone de mémoire définie par *intervalle* dans une autre zone commançant à l'adresse *adresse*.

Exemples :

`bm 10000,11000 20000`
recopie la zone mémoire allant de 10000 à 11000 dans la zone de même taille débutant en 20000.

3.4 Commande `br` : mise d'un point d'arrêt

Syntaxe :

`br` $adresse_{option}$ $nombre\text{-}de\text{-}passages_{option}$

dans lequel *nombre-de-passages* a la structure : : *expression*

Sémantique :

La commande `br` permet de mettre un point d'arrêt à une certaine *adresse*. Quand en exécutant le programme, le processeur accède à *adresse* pour y lire une instruction, le programme est stoppé, son état est sauvegardé et le contrôle est redonné à 16xBug. L'utilisateur peut alors inspecter l'état du programme à l'aide des commandes (par exemple `rd` pour afficher la valeur des registres)

Exemples :

On met un point d'arrêt en `20000` :

```
162-Bug>br 20000
BREAKPOINTS
00020000
162-Bug>
```

On voit que 16xBug après avoir exécuté la commande, affiche l'ensemble des points d'arrêts valides à cet instant.

Si la commande est utilisée avec un *nombre-de-passages*, la moniteur met un point d'arrêt à l'*adresse* indiquée, mais l'arrêt ne se fera qu'après être passé *nombre-de-passages* fois sur le point d'arrêt.

On met un point d'arrêt en `20000`, mais on ne veut s'arrêter qu'au deuxième passage :

```
162-Bug>br 20000:2
BREAKPOINTS
00020000:2
```

Si la commande est utilisée sans aucune option, elle a pour effet de faire lister l'ensemble des points d'arrêts qui sont actuellement mis dans le programme.

3.5 Commande `nobr` : suppression d'un point d'arrêt

Syntaxe :

`nobr` $adresse_{option}$

Sémantique :

cette commande supprime le point d'arrêt qui se trouve à *adresse*. Si la commande est utilisée sans indiquer d'*adresse*, elle supprime tous les points d'arrêt.

3.6 Commande `bs` : Recherche dans une zone de mémoire

Syntaxe :

La commande `bs` peut avoir l'une des deux formes suivantes :

`bs` *intervalle* *delim* *chaîne* $taille_{option}$

ou :

`bs` *intervalle* *delim* $expression_1$ $expression_{2option}$ $taille_{option}$

Sémantique :

Dans la première forme on demande au moniteur de rechercher la *chaîne* dans la zone mémoire définie par *intervalle*.

Dans la deuxième forme, on demande au moniteur de rechercher la valeur de $expression_1$ dans la zone mémoire définie par *intervalle*. La taille de la valeur à rechercher peut être précisée par le paramètre *taille*, sinon elle est prise par défaut comme étant un mot. Le paramètre optionnel $expression_2$ est un masque indiquant la manière dont se fait la comparaison de la valeur courante avec $expression_1$. Pour que la valeur courante soit considérée comme correspondant à $expression_1$, il suffit qu'elle lui corresponde pour les seuls bit à 1 dans le masque.

Exemples :

Recherchons à quelle adresse 16xBug possède la chaîne qui lui sert de prompt :

```
162-Bug>bs ff800000 ff880000 '162-Bug'
Effective address: FF800000
Effective address: FF880000
FF8633AD
162-Bug>
```

On voit que cette chaîne se trouve à l'adresse `FF8633AD`. Attention, une telle recherche prend environ 4 minutes.

Recherchons dans le moniteur des mots contenant la valeur `1234`

```
62-Bug>bs ff800000 ff880000  1234
```

```
Effective address: FF800000
Effective address: FF87FFFF
FF81D816|1234  FF822CD6|1234  FF83BAB2|1234  FF83BAD2|1234
FF83BAFA|1234  FF83C8A8|1234  FF83C980|1234  FF83C9A2|1234
FF83CFE8|1234  FF83D00A|1234  FF83D01E|1234  FF83D382|1234
FF83D426|1234  FF83D430|1234
162-Bug>
```

On voit que le moniteur trouve cette valeur en 14 endroits et cette fois-ci la recherche prend environ 15 secondes, car les comparaisons se font mot par mot au lieu d'octet par octet.

Cherchons maintenant toujours dans le code de 16xBug un mot contenant la valeur `x001x010x011x100` où x représente un bit dont la valeur peut être 0 ou 1.

```
162-Bug>bs ff800000 ff880000 1234 7777
Effective address: FF800000
Effective address: FF87FFFF
FF802814|12BC  FF802914|12BC  FF802A50|12BC  FF802AE4|123C
FF802B12|123C  FF8090F0|12BC  FF81B88A|12BC  FF81B88E|12BC
....
162-Bug>
```

3.7 Commande gd : exécution sans point d'arrêt

Syntaxe :

`gd` $adresse_{option}$

Sémantique :

Cette commande demande à 16xBug de lancer l'exécution à l'adresse *adresse* si celle-ci est précisée, ou à partir du contenu du registre `PC`, l'exécution se faisant en ignorant les points d'arrêts éventuels.

3.8 Commande gn : exécution d'une instruction

Syntaxe :

`gn`

Sémantique :

Cette commande demande à 16xBug de mettre un point d'arrêt temporaire, à l'adresse de l'intruction suivante, et de lancer l'exécution. Une seule instruction va donc s'exécuter, sauf si l'instruction courante est un `jsr` ou `bsr` auquel cas c'est toute la routine appelée qui va s'exécuter.

3.9 Commande `go` : exécution avec point d'arrêt

Syntaxe :

`go` $adresse_{option}$

Sémantique :

Cette commande demande à 16xBug de lancer l'exécution à l'adresse *adresse* si celle-ci est précisée, ou à partir du contenu du registre `PC`. L'exécution s'arrêtera sur la rencontre du premier point d'arrêt.

3.10 Commande `gt` : exécution avec point d'arrêt temporaire

Syntaxe :

`gt` *adresse*

Sémantique :

Cette commande demande à 16xBug de lancer l'exécution à partir du contenu du registre `PC` après avoir mis un point d'arrêt temporaire à l'adresse *adresse*.

3.11 Commande `he` : aide en ligne

Syntaxe :

`he` $nom\text{-}de\text{-}commande_{option}$

Sémantique :

La commande `he` sans paramètre provoque l'affichage de l'ensemble des commandes acceptées par 16xBug accompagnées d'un texte très court résumant leur utilisation.

Utilisée avec en paramètre *nom-de-commande*, `he` affiche la syntaxe de la commande.

Exemples :

```
162-Bug>he bm
Block of Memory Move:
BM <RANGE><DEL><ADDR>[;B|W|L]
162-Bug>
```

3.12 Commande `ma` : définition de macro

Syntaxe :

`ma` nom_{option}

Sémantique :

Cette commande a pour but de définir une macro de nom *nom*. Après avoir émis la commande, l'utilisateur tape une suite de lignes terminées par une ligne vide. Cette suite de lignes forme le corps de la macro *nom*. Ensuite, *nom* est un raccourci d'écriture pour l'ensemble des lignes formant le corps de la macro.

Exemples :

Par exemple, faisons une macro pour afficher le contenu de la pile sur une longueur de 16 mots :

```
162-Bug>ma pile
M=md 0(a7):10
M=
162-Bug>
```

La commande `md` est une commande de dump de mémoire, elle sera étudiée en 3.14. Maintenant, utilisons la macro :

```
162-Bug>pile
162-Bug>md 0(a7):10
00004000 0000  FFFF  0000  FFFF   0000  FFFF  0000  FFFF      ................
00004010 0000  FFFF  0000  FFFF   0000  FFFF  0000  FFFF      ................
```

On voit que 16xBug avant d'exécuter chaque commande du corps de la macro, imprime cette commande.

Les macros admettent des paramètres grâce au caractère d'échappement `\`. Dans le corps d'une macro, les chaînes `\0`, `\1`, `\2` etc... seront remplacées respectivement par le premier, second troisième etc... paramètre.

Paramétrons la macro précédente :

```
62-Bug>ma pile
M=md 0(A7):\0
M=
162-Bug>
```

puis utilisons la :

```
162-Bug>pile 10
162-Bug>md 0(A7):10
00300000  0000 0000 0000 0000  0000 0400 0000 0000  ................
00300010  1000 0000 0008 0000  0000 0000 0000 0080  ................
162-Bug>
```

L'exécution de `ma` sans paramètre provoque l'affichage de toutes les macros définies.

3.13 Commande `noma` : effacement de macro

Syntaxe :

`noma` nom_{option}

Sémantique :

La commande `noma` permet d'effacer une macro dont on donne le nom, ou d'effacer toutes les macros si le paramètre *nom* est absent.

3.14 Commande `md` : Impression du contenu d'une zone de mémoire

Syntaxe :

`md` *intervalle* $taille_{option}$

ou

`md` *intervalle* `; di`

Sémantique :

La première forme demande à 16xBug de réaliser un dump en hexadécimal de la zone de mémoire définie par *intervalle*. La seconde forme demande à 16xBug de désassembler une zone de mémoire. On peut, à la place d'un *intervalle*, donner une *adresse*. Dans ce cas, la zone mémoire est prise comme débutant à *adresse* et ayant une longueur de 16 octets.

Exemples :

Regardons le début de 16xBug :

```
162-Bug>md ff800000
FF800000  FFE0 E000 FF80 0A36  FF80 0DBA FF80 0DC4  .......6........
162-Bug>
```

On voit que sur reset, on charge la valeur FFE0E000 dans le pointeur de pile système, et la valeur FF800A36 dans PC. Désassemblons alors la routine reset de 16xBug :

```
62-Bug>md FF800A36 ;di
FF800A36 46FC2700          MOVE.W      #$2700,SR
FF800A3A 4EF9FF80 0A40     JMP         ($FF800A40).L
FF800A40 203C00FF C040     MOVE.L      #$FFC040,D0
FF800A46 4E7B0007          MOVEC.L     D0,DTT1
FF800A4A 207CFFF4 2000     MOVEA.L     #$FFF42000,A0
FF800A50 317C0001 0026     MOVE.W      #$1,$26(A0)
FF800A56 243CFFE0 0000     MOVE.L      #$FFE00000,D2
FF800A5C 4842              SWAP.W      D2
...
```

3.15 Commande `mm` : modification du contenu d'une zone mémoire

Syntaxe :

`mm` *adresse* $taille_{option}$

ou

`md` *adresse* `DI`

Sémantique :

Cette commande est une commande interactive qui permet d'inspecter une suite de mémoire contigues débutant à *adresse* et d'en modifier certaines.

Exemples :

Montrons sur un exemple : nous allons inspecter (et éventuellement modifier) à partir de `3e00`.

```
162-Bug>mm 3e00
00003E00 0000?
00003E02 FFFF? 1
00003E04 0000?
00003E06 FFFF? 2
00003E08 0000? .
162-Bug>
```

On voit que les mémoires d'adresse `3e00` et `3e04` contenaient la valeur 0 et que l'utilisateur ne les a pas modifiées, et que les mémoires d'adresse `3e02` et `3e06` contenaient `ffff` et que leur précédent contenu a été remplacé par `1` et `2`. L'utilisateur signifie la fin de la commande `mm` en tapant une ligne ne contenant que le caractère `.` (point).

On peut également utiliser la commande `mm` avec l'option `di`. Dans ce cas, `mm` désassemble instruction par instruction, et à chaque fois autorise l'utilisateur a taper le source d'une autre instruction à la place. Il n'est raisonnable d'utiliser cette option que pour faire des modifications minimes au programme.

3.16 Commande `nbh` : chargement de programme

L'utilisation de la commande `nbh` a été vue en 2.5.2.

3.17 Commande `rd` : visualisation des registres

Syntaxe :

`rd`

Sémantique :

Cette commande provoque l'affichage des registres ainsi que le désassemblage de l'instruction courante

3.18 Commande `rm` : visualisation/ modification de registres

Syntaxe :

`rm` *nom*

Sémantique :

Cette commande est une commande interactive fonctionnant à la manière de `mm` permettant de visualiser et éventuellement de modifier le contenu des registres. Le *nom* indiqué doit être un nom de registre.

Exemples :

Exemple : on inspecte D3,D4,D5 et on ne modifie que D5 :

```
162-Bug>rm d3
D3   =00000000 ?              <-- Retour chariot invisible
D4   =00000001 ? 2
D5   =00000000 ? .
162-Bug>
```

Autre exemple, la modification du regitre d'état. Dans l'interaction ci-dessous, on remet dans le registre d'état la valeur convenable pour 16xBug (mode maître, masqué).

```
162-Bug>rm sr
SR   =0000=TR:OFF_..._0_....? 2700
VBR  =00000000? .
162-Bug>
```

3.19 Commande `t` : exécution d'une instruction

Syntaxe :

`t` $expression_{option}$

Sémantique :

Cette commande demande d'exécuter un nombre d'instruction égal à *expression*. Si le *expression* est omis, elle est prise égale à 1. Après chaque instruction 16xBug imprime tous les registres et désassemble l'instruction courante (la prochaine à exécuter). Cette commande permet de tracer l'exécution d'un programme en pas à pas.

3.20 Commande `tt` : trace avec point d'arrêt temporaire

Syntaxe :

`tt` *adresse*

Sémantique :

Cette commande demande à 16xBug de lancer l'exécution en mode trace (exécution pas à pas avec affichage du contenu des registres après chaque instruction), avec un point d'arrêt temporaire

3.21 L'ensemble des commandes en un coup d'oeil

Visualiser/modifier la mémoire

bf	Block Fill	Remplissage d'une zone de mémoire
bm	Block Move	Recopie de zone de mémoire
bs	Block Search	Recherche dans une zone de mémoire
md	Memory Display	Affichage du contenu d'une zone mémoire
mm	Memory Modify	Modification d'une zone de mémoire

Visualiser/modifier les registres

rd	Register Display	Affichage du contenu des registres
rm	Register Modify	Modification du contenu des registres

Gestion des points d'arrêt

br	Breakpoint	Pose d'un point d'arrêt
nobr	No Breakpoint	Suppression de points d'arrêt

Exécution des programmes

gd	Go Direct	Exécution sans point d'arrêt
gn	Go Next	Exécution d'une instruction
go	Go	Exécution avec points d'arrêt
gt	Go to Temporary	Exécution jusqu'à point d'arrêt temporaire
t	Trace	Exécution d'une instruction
tt	Trace to Temporary	Exécution pas à pas jusqu'a point d'arrêt temporaire

Connexion à la machine de développement

nbh	Network Boot and Halt	Chargement de programme

Divers

he	Help	Aide en ligne
noma	No Macro	Effacement de macro
ma	MAcro	Definition de macro

Chapitre 4

Les appels noyau de 16xBug

4.1 Généralités sur les appels noyau

En plus des services accessibles par le dialogue utilisateur (les commandes), 16xBug offre un certain nombre de services accessibles par la programmation : ce sont les appels noyau. Les appels noyau se font tous à l'aide de l'instruction

```
        trap    #15
```

Pour pouvoir indiquer à la routine qui traite le `trap    #15` quel appel noyau le programmeur désire activer, il faut mettre derrière cette instruction un mot contenant un code caractérisant l'appel noyau. Par exemple:

```
        trap    #15
        .word   0x21
```

est un appel à la routine OUTSTR qui imprime un buffer sur la voie de sortie par défaut ($21_{[16]}$ est le code de l'appel noyau OUTSTR). Pour rendre le code plus clair, on recommande de programmer de la manière suivante :

```
outstr  =       0x21

        trap    #15
        .word   outstr
```

La routine qui traite le `trap    #15` fait progresser le compteur ordinal de l'appelant de manière à lui faire sauter le mot contenant le code.

Les paramètres de chaque appel noyau sont mis dans la pile, et généralement chaque appel noyau dépile ses paramètres (l'appelant n'a pas à s'en charger). Exemple d'un appel à OUTSTR:

```
outstr  =       0x21

        move.l  last,-(%sp)     /*   adr du dernier carac du buffer + 1   */
        move.l  first,-(%sp)    /*   adr du premier carac du buffer       */
        trap    #15
        .word   outstr
retour:                         /*   ici first et last ont étés dépilés   */
```

Les appels noyau concernent essentiellement les entrées sorties. Pour ces entrées sorties, le noyau gère les notions de “voie d'entrée par défaut” et de “voie de sortie par défaut”. Notons la différence avec les commandes qui elles, gèrent des voies physiques: 0 est la voie terminal (entrée et sortie) et la voie 1 est la voie connexion avec la machine de développement (entrée et sortie)...

La “voie d'entrée par défaut” et la “voie de sortie par défaut” correspondent grosso-modo aux concept de “standard input” et “standard output” dans UNIX.

4.2 L'appel noyau `INCHR`

Code : $0_{[16]}$

Description :
Lit un caractère sur la voie d'entrée par défaut

La pile en entrée :
en 0(sp) doit se trouver un byte non initialisé dans lequel `INCHR` déposera le résultat.
en 1(sp) doit se trouver un byte non initialisé dont le seul but est de maintenir la pile alignée sur une frontière de mot.

La pile en sortie :
L'appel `INCHR` ne modifie pas la valeur de sp.

Exemple d'appel :

```
inchr   =       0x0

        subq.l  #2,%sp          /*   de la place pour le résultat   */
        trap    #15
        .word   inchr
        move.b  (%sp)+,%d0      /*   résultat dans d0               */
```

4.3 L'appel noyau `INSTAT`

Code : $1_{[16]}$

Description :
Teste si il y a un caractère à lire sur la voie d'entrée par défaut

La pile en entrée :
Il n'y a aucun paramètre à cet appel noyau.

La pile en sortie :
L'appel `INSTAT` ne modifie pas la valeur de sp.

La valeur rendue :
Le bit Z du code condition vaut 0 si il y a un caractère à lire, 1 sinon.

Exemple d'appel :

```
instat  =       0x1

        trap    #15
        .word   instat
        beq     vide          /*  on se branche a vide si il n'y a pas de   */
                              /*  caractère a lire                          */
```

4.4 L'appel noyau INLN

Code : $2_{[16]}$

Description :

Lit une ligne sur la voie d'entrée par défaut.

La pile en entrée :

en 0(sp) doit se trouver un long contenant l'adresse du buffer où l'on désire que `INLN` mette les caractères lus. Ce buffer doit avoir une longueur de 256 octets. Le caractère indiquant la fin de la ligne est le caractère `\r`, et ce caractère n'est pas mis dans le buffer. La lecture de caractères est faite en acceptant les caractères de contrôle définis en 3.1.1.

en 4(sp) doit se trouver un long non initialisé dans lequel `INLN` déposera l'adresse +1 du dernier caractère mis dans le buffer.

La pile en sortie :

L'appel `INLN` dépile le premier paramètre.

Exemple d'appel :

```
inln    =       0x2

        subq.l  #4,%sp          /*  de la place pour le résultat        */
        pea     buffer          /*  empilement de l'adresse de buffer   */
        trap    #15
        .word   inln
        move.l  (%sp)+,%d0      /*  adr dernier carac + 1 dans d0       */

        .data
buffer: .space  256
```

4.5 L'appel noyau OUTCHR

Code : $20_{[16]}$

Description :
Ecrit un caractère sur la voie de sortie par défaut.

La pile en entrée :
en 0(sp) doit se trouver un byte contenant le caractère à écrire. en 1(sp) doit se trouver un byte non initialisé dont le seul but est de maintenir la pile alignée sur une frontière de mot.

La pile en sortie :
L'appel `OUTCHR` dépile les deux bytes.

Exemple d'appel :

```
outchr  =       0x20

        move.b  %d0,-(%sp)      /*  d0 contient le caractère a écrire   */
        trap    #15
        .word   outchr
```

4.6 L'appel noyau `OUTSTR`

Code : $21_{[16]}$

Description :
Ecrit une chaîne de caractères sur la sortie par défaut.

La pile en entrée :
en 0(sp) doit se trouver un long contenant l'adresse du premier caractère de la chaîne. en 4(sp) doit se trouver un long contenant 1 + l'adresse du dernier caractère de la chaîne.

La pile en sortie :
L'appel `OUTSTR` dépile les deux paramètres.

Exemple d'appel :

```
outstr  =       0x21
RC      =       13
LF      =       10

        pea     endstr          /*  adr du dernier carac de la chaîne + 1   */
        pea     str             /*  adr du premier carac de la chaîne       */
        trap    #15
        .word   outstr

        .data
str:    .ascii  "Bonjour!"
```

```
        .byte    RC
        .byte    LF
endstr:
```

4.7 L'appel noyau `OUTLN`

Code : $22_{[16]}$

Description :
Ecrit une chaîne de caractères puis Carriage-Return Line-Feed sur la sortie par défaut.

La pile en entrée :
en 0(sp) doit se trouver un long contenant l'adresse du premier caractère de la chaîne.
en 4(sp) doit se trouver un long contenant 1 + l'adresse du dernier caractère de la chaîne.

La pile en sortie :
L'appel `OUTLN` dépile les deux paramètres.

Exemple d'appel :

```
outln   =       0x22

        pea     endstr          /*   adr du dernier carac de la chaine + 1   */
        pea     str             /*   adr du premier carac de la chaine       */
        trap    #15
        .word   outln

        .data
str:    .ascii  "Bonjour!"
endstr:
```

4.8 L'appel noyau `WRITD`

Code : $28_{[16]}$

Description :
Réalise sur la sortie par défaut une impression formattée. `WRITD` a deux paramètres

1. un format contenant des caractères à imprimer tels quels, et des séquences d'échappement demandant l'impression d'une valeur numérique. Une telle séquence d'échappement a la forme suivante: `|` *base largeur* `|` . Dans ces séquences,
   - *base* est la base dans laquelle on demande l'impression d'une valeur. Cette base est elle même exprimée en héxadécimal!

– *longueur* est le nombre de chiffres à imprimer. Si la valeur est trop grande pour pouvoir être écrite avec *longueur* chiffres, les chiffres les plus à gauches sont enlevés. De plus un `Z` optionnel peut être mis derrière *longueur* pour demander la suppression des zéros initiaux.

Voyons sur un exemple. La valeur $32_{[10]}$ selon plusieurs formats

le format	imprimera
`|10,8|`	00000020
`|10,8Z|`	20
`|A,8|`	00000032
`|A,8Z|`	32

Attention: le premier octet du format doit contenir la longueur de la chaîne qui suit.

2. un tableau contenant les valeurs a imprimer.

La pile en entrée :

en 0(sp) doit se trouver un long contenant l'adresse d'une chaîne de caractères qui est un format.
en 4(sp) doit se trouver un long contenant l'adresse d'un tableau contenant les valeurs à imprimer.

La pile en sortie :

L'appel `WRITD` dépile ses deux paramètres.

Exemple d'appel :

```
writd   =       0x28
RC      =       13
LF      =       10

        pea     tabval      /*   adresse du tableau contenant les valeurs   */
        pea     format      /*   adresse de la chaine contenant le format   */
        trap    #15
        .word   writd

        .data
format: .byte   endstr-str   /*   octet contenant la longueur de la chaine   */
str:    .ascii  "i = |A,8| j = |A,8|"
        .byte   RC
        .byte   LF
endstr:

tabval:
i:      .long   123
j:      .long   78
```

4.9 L'appel noyau `WRITDLN`

Code : $25_{[16]}$

Description :
Même chose que `WRITD` mais imprime un Carriage-Return et un Line-Feed après avoir terminé l'impression du format.

La pile en entrée :
en 0(sp) doit se trouver un long contenant l'adresse d'une chaîne de caractères qui est un format.
en 4(sp) doit se trouver un long contenant l'adresse d'un tableau contenant les valeurs à imprimer.

La pile en sortie :
L'appel `WRITDLN` dépile ses deux paramètres.

Exemple d'appel :

```
writdln =       0x25

        pea     tabval      /*  adresse du tableau contenant les valeurs   */
        pea     format      /*  adresse de la chaine contenant le format   */
        trap    #15
        .word   writdln

        .data
format: .byte   endstr-str  /*  octet contenant la longueur de la chaine   */
str:    .ascii  "i = |A,8| j = |A,8|"
endstr:

tabval:
i:      .long   123
j:      .long   78
```

4.10 L'appel noyau `REDIR_I`

Code : $61_{[16]}$

Description :
Redirige l'entrée par défaut. Cet appel a un effet de bord sur tous les appels noyaux réalisant des entrées sur la "voie d'entrée par défaut".

La pile en entrée :
en 0(sp) doit se trouver un mot contenant la voie physique vers laquelle on veut rediriger l'entrée par défaut. On rappelle que la voie 0 est le terminal de dialogue et que la voie 1 est la connexion avec la machine de développement.

La pile en sortie :
L'appel REDIR_I dépile son paramètre.

Exemple d'appel :

```
redir_i =       0x61

        move.w  #1,-(%sp)       /*   redirection vers la voie 1   */
        trap    #15
        .word   redir_i
```

4.11 L'appel noyau REDIR_O

Code : $62_{[16]}$

Description :
Redirige la sortie par défaut. Cet appel fait un effet de bord sur tous les appels noyaux réalisant des sorties sur la “voie de sortie par défaut”.

La pile en entrée :
en 0(sp) doit se trouver un mot contenant la voie physique vers laquelle on veut rediriger la sortie par défaut. On rappelle que la voie 0 est le terminal de dialogue et que la voie 1 est la connexion avec la machine de développement.

La pile en sortie :
L'appel REDIR_O dépile son paramètre.

Exemple d'appel :

```
redir_o =       0x62

        move.w  #1,-(%sp)       /*   redirection vers la voie 1   */
        trap    #15
        .word   redir_o
```

4.12 L'appel noyau RETURN

Code : $63_{[16]}$

Description :
Retour à 16xBug.

La pile en entrée :
Pas de paramètre.

La pile en sortie :
Comme en entrée.

Exemple d'appel :

```
return  =       0x63

        trap    #15
        .word   return
```

4.13 L'appel noyau MULU32

Code : $69_{[16]}$

Description :
Réalise la multiplication entre deux valeurs non signées de 32 bits avec résultat en non signé sur 32 bits. (On rappelle que le MC68000 et le MC68010 ne peuvent réaliser que la multiplication de deux valeurs non signées de 16 bits avec résultat en non signé sur 32 bits). Malheureusement, l'overflow n'est pas détecte!

La pile en entrée :
en 0(sp) doit se trouver un long contenant le multiplicateur.
en 4(sp) doit se trouver un long contenant le multiplicande.
en 8(sp) doit se trouver un long non initialisé dans lequel `MULU32` déposera le résultat.

La pile en sortie :
L'appel `MULU32` dépile ses deux premiers paramètres.

Exemple d'appel :

```
mulu32  =       0x69

        subq.l  #4,%sp          /*  de la place pour le résultat   */
        move.l  cande,-(%sp)    /*  le multiplicande               */
        move.l  cateur,-(%sp)   /*  le multiplicateur              */
        trap    #15
        .word   mulu32
        move.l  (%sp)+,%d0      /*  résultat dans d0               */
```

4.14 L'appel noyau DIVU32

Code : $6A_{[16]}$

Description :
Réalise la division entre deux valeurs non signées de 32 bits avec résultat en non signé sur 32 bits. (On rappelle que le MC68000 et le MC68010 ne peuvent réaliser que la division d'une valeur non signée sur 32 bits par une valeur non signée sur 16 bits, avec résultat non signé sur 16 bits.)

La pile en entrée :

en 0(sp) doit se trouver un long contenant le diviseur.
en 4(sp) doit se trouver un long contenant le dividende.
en 8(sp) doit se trouver un long non initialisé dans lequel `DIVU32` déposera le résultat.

La pile en sortie :

L'appel `DIVU32` dépile ses deux premiers paramètres.

Exemple d'appel :

```
divu32  =       0x6A

        subq.l  #4,%sp          /*   de la place pour le résultat   */
        move.l  dende,-(%sp)    /*   le dividende                   */
        move.l  seur,-(%sp)     /*   le diviseur                    */
        trap    #15
        .word   divu32
        move.l  (%sp)+,%d0      /*   résultat dans d0               */
```

4.15 Les appels noyau en un coup d'oeil

`INCHR`	Lit un caractère sur la voie d'entrée par défaut
`INLN`	Lit une ligne sur la voie d'entrée par défaut
`OUTCHR`	Ecrit un caractère sur la voie de sortie par défaut
`OUTSTR`	Ecrit une chaîne de caractères sur la voie de sortie par défaut
`OUTLN`	Ecrit une chaîne de caractères puis Carriage Return sur la voie de sortie par défaut
`WRITD`	Fait une impression formattée sur la voie de sortie par défaut
`WRITDLN`	Fait une impression formattée puis écrit Carriage-Return Line-Feed sur la voie de sortie par défaut
`REDIR_I`	Redirige la voie d'entrée par défaut
`REDIR_O`	Redirige la voie de sortie par défaut
`RETURN`	Retour à 16xBug
`MULU32`	Multiplication entre nombres non signés 32 bits
`DIVU32`	Division entre nombres non signés 32 bits

Chapitre 5

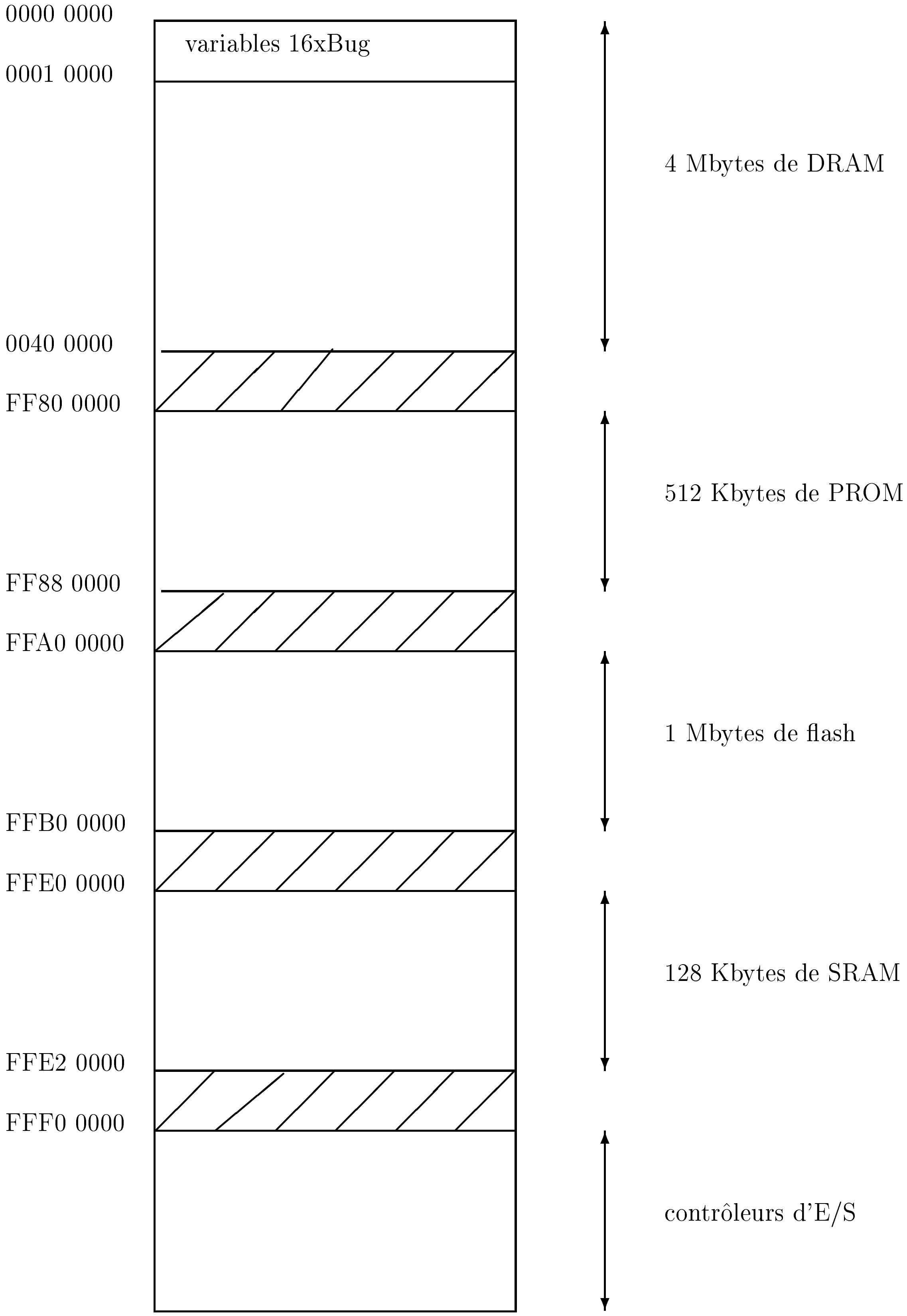
La carte mvme 162

5.1 Généralités

Cette carte comporte:

- un MC68LC040 cadencé à 25 MHz (le MC68LC040 est une version « low cost » du MC68040 : il n'y a pas d'unité de calculs flottants) ;
- 4 Mbytes de mémoire dynamique ;
- 128 Kbytes de mémoire statique ;
- 512 Kbytes de PROM contenant un chargeur/debugger : 16xBug.
- deux contrôleurs de lignes synchrone/asynchrone : le Zilog Z85230 ;
- un contrôleur ethernet : l'intel 82596CA ;
- un contrôleur SCSI : le NCR 53C710 ;
- un contrôleur mémoire (MCChip), qui entre autres fonctions, offre :
  + 4 timers programmables ;
  + la gestion des interruptions des contrôleurs d'entrées/sorties.

5.2 Topographie de la mémoire



5.3 Les adresses des registres des contrôleurs

5.3.1 Le Z85230

Le Z85230 comporte deux canaux: le canal A et le canal B. Chaque canal comporte deux registres qui sont “mappés” dans l’espace d’adressage: il s’agit des registres ‘Command Register 0” et “Data Register”. Il y a sur la carte deux contrôleurs Z85230 de façon à disposer d’un total de 4 lignes asynchrones. Les adresses sont les suivantes :

Premier contrôleur

	Command Register 0	Data Register
canal A	FFF45005	FFF45007
canal B	FFF45001	FFF45003

Second contrôleur

	Command Register 0	Data Register
canal A	FFF45805	FFF45807
canal B	FFF45801	FFF45803

5.3.2 Le MCchip

Le timer 1

registre	adresse	taille
Registre de comparaison	FFF42004	32 bits
Registre de comptage	FFF42008	32 bits
Registre de contrôle	FFF42017	8 bits
Registre de contrôle des IT	FFF4201B	8 bits

Le timer 2

registre	adresse	taille
Registre de comparaison	FFF4200C	32 bits
Registre de comptage	FFF42010	32 bits
Registre de contrôle	FFF42016	8 bits
Registre de contrôle des IT	FFF4201A	8 bits

Le timer 3

registre	adresse	taille
Registre de comparaison	FFF42030	32 bits
Registre de comptage	FFF42034	32 bits
Registre de contrôle	FFF4201F	8 bits
Registre de contrôle des IT	FFF42019	8 bits

Le timer 4

registre	adresse	taille
Registre de comparaison	FFF42038	32 bits
Registre de comptage	FFF4203C	32 bits
Registre de contrôle	FFF4201E	8 bits
Registre de contrôle des IT	FFF42018	8 bits

Les Z85230

registre	adresse	taille
Registre de contrôle des IT	FFF4201D	8 bits

5.4 Description des registres du MCchip

5.4.1 Registre de contrôle général

7	6	5	4	3	2	1	0
						MIEN	FAST

MIEN Master Interrupt Enable. Quand ce bit est à 1, si il y a des demandes d'interruption de la part de l'un quelconque des contrôleurs gérés par le MCchip, le MCchip activera une demande d'interruption du processeur. Si ce bit est à 0, le MCchip n'activera pas de demande d'interruption du processeur.

FAST Bit de contrôle de la mémoire. **Ne jamais modifier ce bit**.

5.4.2 Registre de contrôle des timers

7	6	5	4	3	2	1	0
OVFL					COVF	COC	CEN

OVFL il s'agit d'un registre de 4 bits qui compte le nomnbre de fois où le timer a généré une interruption.

COVF le registre **OVFL** est remis à zéro chaque fois que l'on écrit un 1 dans ce bit.

COC quand ce bit vaut 1, le registre de comptage est remis à zéro quand il atteint la valeur du registre de comparaison ; si ce bit vaut 0, le registre de comptage continue à être incrémenté après avoir atteint la valeur du registre de comparaison.

CEN quand ce bit vaut 1, le registre de comptage est incrémenté (le timer fonctionne) ; quand ce bit vaut 0, le registre de comptage n'est pas incrémenté (le timer est arrété).

5.4.3 Registre de contrôle des interruptions des timers

7	6	5	4	3	2	1	0
		INT	IEN	ICLR	LEVEL		

LEVEL Le niveau de l'interruption qui sera générée ;

ICLR quand on écrit un 1 dans ce bit, ça remet à zéro le bit INT de ce même registre.

IEN bit « enable » : autorise l'interruption quand IEN = 1, l'interdit sinon.

INT quand ce bit vaut 1, le MCchip active une demande d'interruption du processeur ; quand ce bit vaut 0, il n'y a pas de demande d'interruption du processeur.

Chapitre 6

Les logiciels imag

6.1 La librairie lib16xbug

J'ai développé une librairie permettant d'utiliser les routines de 16xBug à partir du langage C, ce qui permet d'écrire rapidement des programmes de test. Le source de cette librairie se trouve dans le directory `~cassagne/enseignement/mvme162/lib16xbug/src/`.

Voici les services apportés par cette librairie.

6.1.1 La fonction `inchr`

Synopsis:

```
char inchr()
```

Description:
Lit un caractère sur la voie d'entrée par défaut.

Valeur rendue:
Le caractère lu.

Exemple:

```
char buffer[80];
char *p = buffer;
char c;

while ((c = inchr()) != '\04')
   *p++ = c;
```

6.1.2 La fonction `instat`

Synopsis:

```
int instat()
```

Description:
Teste si il y a un caractère à lire sur la voie d'entrée par défaut.

Valeur rendue:
- 1 si il y a un caractère à lire.
- 0 sinon.

Exemple:

```
if (instat())
   c = inchr();
```

6.1.3 La fonction `inln`

Synopsis:

```
char * inln(buffer)
char *buffer;
```

Description:
Lit une ligne sur la voie d'entrée par défaut et range les caractères lus dans `buffer`.

Valeur rendue:
L'adresse dans `buffer` du caractère suivant le dernier caractère lu.

Exemple:

```
extern char *inln();
char buffer[256];
char *p;

p = inln(buffer);
*p++ = '\n';
```

6.1.4 La fonction `outchr`

Synopsis:

```
void outchr(c)
char c;
```

Description:
Ecrit le caractère `c` sur la voie de sortie par défaut.

Valeur rendue:
Pas de valeur rendue.

Exemple:

```
static char message[]="Hello world!!\n";
char *p = message;

while (*p)
   outchr(*p++);
```

6.1.5 La fonction `outstr`

Synopsis:

```
void outstr(deb,fin)
char *deb;
char *fin;
```

Description:
Ecrit une chaîne de caractères sur la voie de sortie par défaut.

Paramètres:
- `deb`: adresse du début de la chaîne de caractères.
- `fin`: 1 + adresse du dernier caractère de la chaîne.

Valeur rendue:
Pas de valeur rendue.

Exemple:

```
static char mess[] = "Hello world\r\n";
char *p = mess;
char *q = mess + 13;

outstr(p,q);
```

6.1.6 La fonction `outln`

Synopsis:

```
void outln(deb,fin)
char *deb;
char *fin;
```

Description:
Ecrit une chaîne de caractères suivie de *carriage return* et *line feed* sur la voie de sortie par défaut.

Paramètres:
- `deb`: adresse du début de la chaîne de caractères.
- `fin`: 1 + adresse du dernier caractère de la chaîne.

Valeur rendue:
Pas de valeur rendue.

Exemple:

```
static char mess[] = "Hello world";
char *p = mess;
char *q = mess + 11;

outln(p,q);
```

6.1.7 La fonction `writd`

Synopsis:

```
void writd(format,tabval)
char *format;
int tabval[];
```

Description:
Réalise une impression formattée sur la voie de sortie par défaut.

Paramètres:
- `format`: adresse du format à imprimer.
- `tabval`: tableau contenant les valeurs à imprimer.

Valeur rendue:
Le caractère lu.

Exemple:

```
int t[2];
static char format[] = "\031i = |A,10Z| j = |A,10Z|\n\r";

t[0] = i;
t[1] = j;
writd(format,t);
```

6.1.8 La fonction `writdln`

Synopsis:

```
void writdln(format,tabval)
char *format;
int tabval[];
```

Description:
Même fonction que `writd` mais imprime *carriage return* et *line feed* après `format`.

6.1.9 La fonction `redir_i`

Synopsis:

```
void redir_i(port)
int port;
```

Description:
Redirige l'entrée par défaut à partir de `port`.

6.1.10 La fonction redir_o

Synopsis:

```
void redir_o(port)
int port;
```

Description:
Redirige la sortie par défaut vers `port`.

6.1.11 La fonction ret16xbug

Synopsis:

```
void ret16xbug()
```

Description:
Retour à 16xBug.

Remarque:
Il n'était pas possible de choisir le nom de cet appel noyau dans 16xbug (RETURN) car il s'agit d'un mot réservé du langage C.

Table des matières