Chapitre 12
Interruptions

Le fonctionnement d'un ordinateur est complexe et peut être représenté par un ensemble de tâches asynchrones interdépendantes; ce fonctionnement est rendu possible par l'existence d'un *système d'interruptions*. Le système d'interruptions peut être relativement simple, permettant au programmeur de demander que son programme soit interrompu, par exemple, à la fin d'une opération d'entrée-sortie, ou lorsqu'un dispositif périphérique est prêt, ou lorsqu'une erreur arithmétique s'est produite. Il peut être également plus complexe et traiter les diverses interruptions par ordre de priorité.

12.1 Systèmes d'interruptions simples

Il existe habituellement dans les processeurs un registre spécial, appelé *registre d'interruptions*, qui comprend autant de bits qu'il y a de conditions d'interruptions possibles. Chaque condition externe pouvant causer une interruption est reliée à un bit particulier du registre d'interruption; de même, chaque condition interne pouvant causer une interruption est associée à un bit particulier de ce registre.

Lorsqu'une interruption se produit, le bit correspondant est positionné dans le registre d'interruption et reste positionné tant qu'il n'est pas modifié par le programme utilisateur ou par le système.

Le contrôle des interruptions se fait par programme au moyen du *registre de masque d'interruptions* qui est rempli, soit par le programme utilisateur, soit par le système. On positionne un bit dans le masque pour chaque interruption que l'on veut reconnaître. Un registre auxiliaire contiendra alors le produit logique du registre d'interruption et du masque d'interruption. Après l'exécution de chaque instruction, on effectue une vérification pour déterminer si une condition d'interruption voulue a eu lieu; cette vérification a lieu au cours du cycle de lecture de la prochaine instruction.

On balaye le registre auxiliaire de droite à gauche, de gauche à droite ou en suivant un ordre de priorité donné; lorsqu'on trouve une interruption voulue, on transfère le contrôle à une adresse spéciale réservée par le système d'exploitation, indiquant l'adresse du programme de traitement de l'interruption.

Pour qu'une interruption soit reconnue il faut en général que:

- le registre de masque d'interruption indique bien l'interruption,
- le système d'interruption soit activé,
- il existe des sous-programmes pour traiter l'interruption.

12.2 Traitement d'une interruption

Dans tout système de programmation, certaines interruptions (dites "temps réel") doivent être traitées au plus tard quelques cycles d'horloge après leur arrivée; d'autres interruptions n'ont pas de contrainte de temps réel et peuvent donc attendre un certain temps avant d'être traitées.

Dans le cas d'un système d'interruption simple (non temps réel), lorsqu'une condition d'interruption se produit, l'ordinateur annihile automatiquement le système d'interruption. Il conserve alors l'adresse de retour (adresse de l'instruction à exécuter après le traitement de l'interruption) à un endroit réservé. Il sauvegarde également le contenu de tous les registres, en les rangeant à un endroit donné, et il passe le contrôle au sous-programme de traitement de l'interruption; ce sous-programme doit revenir à la bonne adresse pour que l'ordinateur puisse restaurer les registres et réactiver le système d'interruption. Le programme reprend alors où il a été interrompu, sans aucune perte. Lorsque le sous-programme de traitement de l'interruption était en cours d'exécution le système d'interruption n'était pas actif; une interruption se produisant alors devait attendre qu'il soit à nouveau actif pour être traitée.

Dans les systèmes plus complexes permettant des interruptions "temps réel", les sous-programmes de traitement des interruptions peuvent eux-mêmes être interrompus. Pour réaliser ceci, on associe un numéro de priorité à chaque type d'interruption et un sous-programme de traitement des interruptions en cours d'exécution ne peut être interrompu que par une interruption ayant une priorité supérieure. Ainsi une interruption donnée ne sera pas interrompue par une interruption de même type.

On peut traiter les interruptions multiples soit en associant une pile d'interruptions à chaque type d'interruption, soit en empilant les interruptions dans les organes qui les provoquent. Cette dernière méthode est très valable si l'organe provoquant une interruption ne peut continuer à fonctionner jusqu'à ce que l'interruption ait été traitée.

La priorité d'une interruption est déterminée en fonction de l'urgence du traitement à effectuer. Afin d'assurer que les interruptions de priorité élevée puissent être traitées de façon adéquate, les suites d'instructions à exécuter pour les traiter doivent être courtes. Il est parfois possible de traiter tout de suite la partie "temps réel" d'une interruption en quelques cycles d'horloge et de placer des interruptions de priorité inférieure dans les diverses piles du système, interruptions qui seront traitées par la suite.

Pour un ordinateur fonctionnant en temps réel, les sous-programmes de traitement des interruptions constituent le premier plan du système d'exploitation, qui permet de contrôler en temps réel le fonctionnement de dispositifs externes. Les programmes d'arrière plan du système (assemblage, compilation) ne jouent alors pas un rôle très important et ne sont exécutés que lorsque les programmes de premier plan ne sont pas actifs.

12.3 Interruptions sur un processeur réel (MC68000)

Le processeur Motorola MC68000 possède des instructions (TRAP, CHK, etc.) qui interrompent l'exécution normale des instructions d'un programme; l'unité centrale peut également interrompre

l'exécution du programme pour signaler des erreurs système. Des périphériques peuvent aussi interrompre l'unité centrale en activant des lignes de contrôle. Les instructions et les conditions provoquant ces interruptions sont appelées "*exceptions*" en jargon Motorola.

Le processeur MC68000 se trouve dans l'un de trois états:

- l'état normal correspondant à l'exécution séquentielle des instructions d'un programme,
- l'état d'exception causé par l'arrivée d'une interruption et pendant lequel le programme de traitement de l'interruption est exécuté,
- l'état arrêté causé par une interruption comme une erreur de bus ou une erreur d'adresse qui ne permettent pas de reprendre l'exécution de façon fiable et qui exigent une intervention externe.

Le registre d'état du MC68000 (figure 12.1) est un registre de 16 bits, dont l'octet droit comprend les codes de condition X, N, Z, V et C. L'octet de gauche utilise 5 bits pour définir le mode et le niveau d'interruption du système.

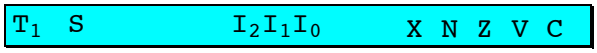

Figure 12.1 Registre d'état du MC68000

Le bit 15, ou T_1, du registre d'état indique le mode d'exécution des instructions. Si ce bit vaut un, les instructions sont exécutées une par une et après chacune, une interruption est engendrée qui permet en particulier l'examen des variables du système: c'est le mode "*trace*".

Le bit 13, ou S, indique le mode de fonctionnement du processeur MC68000. Si sa valeur est 1, on est en *mode superviseur* et on a accès à toutes les instructions, y compris celles qui modifient le registre d'état. Sinon, on est en *mode utilisateur* et on n'a accès qu'à un sous-ensemble d'instructions. Les sous-programmes de traitement des interruptions fonctionnent en mode superviseur.

Les bits 8, 9, 10 ou I_0 , I_1 et I_2 indiquent un niveau d'interruption. Lorsqu'une interruption arrive du processeur, celui-ci peut la traiter immédiatement ou la remettre à plus tard. En particulier, si deux interruptions arrivent en même temps, le système doit décider laquelle traiter d'abord. Le processeur MC68000 utilise le niveau des interruptions pour décider quoi traiter: si la valeur du champ est inférieure à 7, le système ne traitera que les interruptions dont le niveau est supérieur à cette valeur. Au cours du traitement de l'interruption, ce niveau est placé à la valeur de l'interruption courante, et ne peut donc être interrompu que par des instructions de niveau supérieur. Les interruptions de niveau 7 sont traitées, quelle que soit la valeur du champ. Lorsque la valeur du champ est 7, les interruptions de niveau 1 à 6 ne sont pas traitées et le système fonctionne en mode aveugle aux interruptions ("*interrupts disabled*").

Le registre d'état du processeur peut être modifié par quatre instructions de rangement privilégiées (en mode superviseur). Il existe également des instructions pour lire le contenu du registre d'état. Dans ce processeur, il existe en réalité deux piles: l'une d'elles (SP) est utilisée lorsque le processeur fonctionne en mode superviseur, l'autre (USP) est utilisée en mode utilisateur. Il faut cependant noter que dans des systèmes mono-utilisateur comme l'ancien

Macintosh les deux modes de fonctionnement ne sont pas nécessaires, et par conséquent on fonctionne toujours en mode superviseur.

Exemple de cycle de traitement des interruptions du MC68000
Pour chaque interruption (ou exception) le même cycle de traitement est répété.

1. Identification de l'interruption;

2. Sauvegarde temporaire du registre d'état courant dans un registre général;

3. Initialisation du registre d'état. Le bit 13 est mis à 1 (mode superviseur) et le bit T est mis à zéro car on ne veut pas *tracer* le sous-programme de traitement de l'interruption. Pour une interruption externe on place le niveau de l'interruption dans le registre d'état.

4. Détermination du numéro du vecteur d'interruption. Les premières positions de mémoire (de 0 à 1K) constituent le vecteur d'interruptions: les 256 éléments de 4 octets sont chargés au démarrage et comprennent les adresses des sous-programmes de traitement correspondants. Chaque position dans le vecteur est fixée d'avance et correspond à une interruption particulière, comme par exemple :

```
numéro    adresse    interruption
1.   0    0000    Reset
2.   1    0004    Reset
3.   2    0008    Erreur bus
4.   3    000C    Erreur adresse
5.   4    0010    Instruction illégale
6.   5    0014    Division par zéro
7.   6    0018    Instruction CHK
8.   7    001C    Instruction TRAPV
9.   8    0020    Violation de privilège
10.  9    0024    Trace
11.  10   0028    A-line
12.  11   002C    F-line
etc.
```

5. Sauvegarde de l'adresse de retour et du registre d'état. Le compteur ordinal et le registre d'état sont empilés sur la pile du superviseur.

6. Chargement de l'adresse du sous-programme et traitement. L'adresse obtenue en 4 est placée dans le compteur ordinal et l'exécution du sous-programme de traitement démarre.

On utilise l'instruction privilégiée RTE pour le retour du traitement d'une interruption: le registre d'état et le compteur ordinal sont restaurés à partir de la pile du superviseur et l'exécution reprend là où elle avait été interrompue.

Pour le processeur MC68000, il existe un certain nombre d'instructions qui permettent d'engendrer des interruptions. On peut noter les suivantes:


```
RESET           ;ré-initialise les dispositifs externes
TRAP    #N      ;appel système par l'adresse rangée au numéro (32+N)
                ;du vecteur d'interruption
TRAPV           ;génération d'une interruption si le code de
                ;condition V = 1
TRAPcc  #N      ;génération d'une interruption si le code de
                ;condition = 1
CHK     op,Dn   ;génération d'une interruption si la valeur de Dn
                ;est inférieure à zéro ou supérieure à op
CHK2    op,Rn   ;génération d'une interruption si la valeur du
                ;registre n'est pas entre deux valeurs limites
                ;identifiées par op
ILLEGAL         ;cause une interruption d'instruction illégale
```

Exemple:
Le programmeur peut contrôler les cas de division par zéro et de débordement. D'après la table ci-dessus la division par zéro porte le numéro 5 et se trouve donc à l'adresse 5 x 4 = 20_{10} = 14_{16} . Le débordement se trouvant au numéro 7 a donc pour adresse 28_{10} ou $1C. Le programmeur place dans le vecteur d'interruptions les adresses des deux sous-programmes de traitement de ces interruptions. Lorsque ces deux interruptions se produiront, le système passera automatiquement le contrôle à ces sous-programmes.

Dans le cas de débordement, on pourrait se contenter d'afficher un message d'avertissement, avant de continuer l'exécution du programme interrompu. Dans le cas de la division par zéro, on pourrait également afficher un message suivi de l'adresse de retour (adresse qui suit l'instruction où la division par zéro s'est produite) et arrêter l'exécution du programme, après avoir nettoyé la pile en y enlevant les copies du registre d'état et du compteur ordinal empilées par le système de traitement des interruptions.

12.4 PEP 8: interruptions des instructions non implantées

Dans tous les processeurs existe une interruption particulière déclenchée par l'essai d'exécution d'instructions dont le code est non défini (A-line et F-line dans le cas du MC68000 vu plus haut). Dans PEP 8, toutes les instructions pour lesquelles les 5 ou 6 premiers bits du code opération sont soit 001001_2 , 00101_2, 00110_2, 00111_2 ou 01000_2 sont reconnues par le processeur comme des instructions qui n'existent pas ("*unimplemented instructions*"). De façon plus précise, il s'agit des codes opération 24 à 28, 30, 38, et 40). Ces instructions engendrent les seules interruptions du système PEP 8. Ces instructions permettent en particulier au système d'émuler des instructions non disponibles sur la machine. Les applications types pourraient être l'émulation de la multiplication et de la division entières ou l'arithmétique point flottant.

Le traitement de ces interruptions se fait automatiquement: le système empile sur la pile système (qui est différente de celle de l'utilisateur, voir figure 12.2) dans l'ordre : le registre d'instruction (1 octet), le pointeur de pile utilisateur (2 octets), le compteur ordinal (2 octets), le registre X (2 octets), le registre A (2 octets), les codes de condition NZVC (1 octet). La figure 12.3 illustre la pile système après rencontre d'une interruption et empilage de ces dix octets.

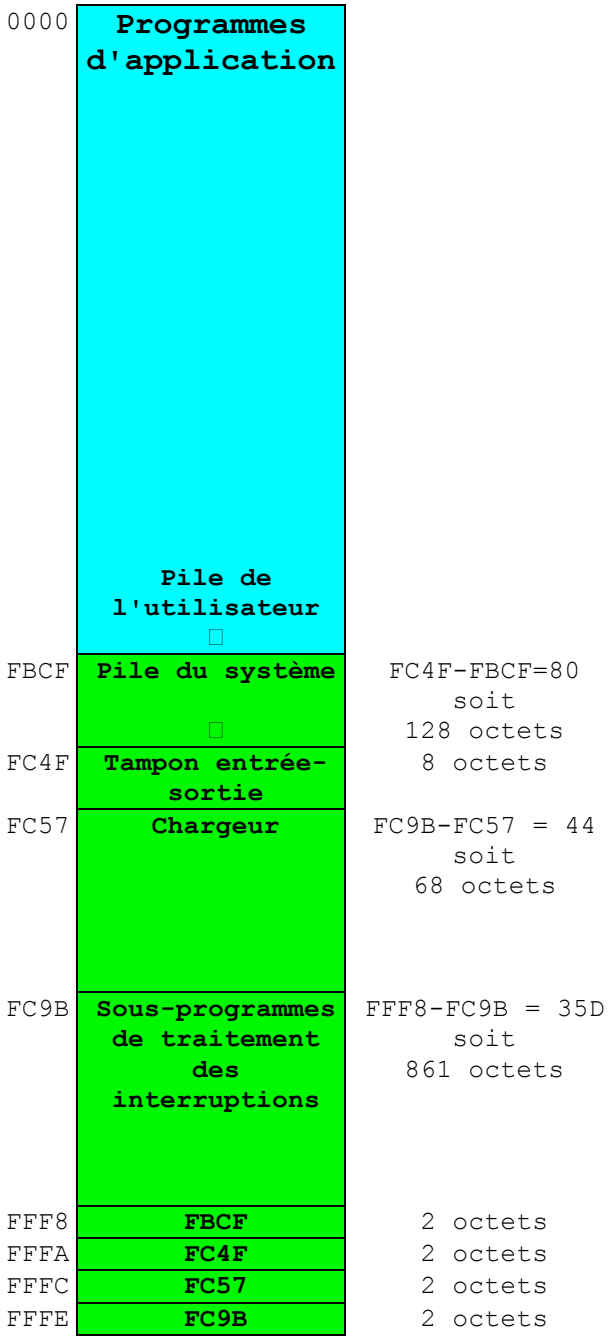

Figure 12.2 Occupation de la mémoire du système PEP 8

Le pointeur de pile (SP) est alors positionné sur ce dernier octet empilé et le compteur ordinal prend la valeur rangée dans les deux dernières positions de la mémoire (FFFE et FFFF). En fait le système Pep8 réserve en permanence 1 073 octets (valeur qui peut changer avec les versions du système) dans la partie haute de la mémoire, comme le montre la figure 12.2.

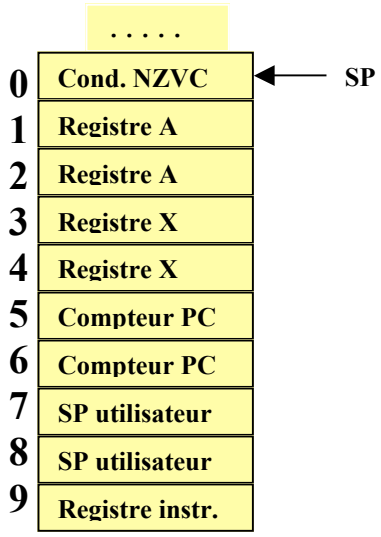

Figure 12.3 Octets de la pile système après interruption

Cette organisation est décrite de la façon suivante dans le code du système d'exploitation de PEP 8.

```
Addr  Code    Symbol    Mnemon  Operand    Comment
              ;******* Pep/8 Operating System, 2004/08/30
              ;
              TRUE:     .EQUATE 1
              FALSE:    .EQUATE 0
              ;
              ;******* Operating system RAM
FBCF          osRAM:    .BLOCK  128        ;System stack area
FC4F          wordBuff:.BLOCK   1          ;Input/output buffer
FC50          byteBuff:.BLOCK   1          ;Least significant byte of wordBuff
FC51          wordTemp:.BLOCK   1          ;Temporary word storage
FC52          byteTemp:.BLOCK   1          ;Least significant byte of tempWord
FC53          addrMask:.BLOCK   2          ;Addressing mode mask
FC55          opAddr:   .BLOCK  2          ;Trap instruction operand address
              ;
              ;******* Operating system ROM
FC57                    .BURN   0xFFFF
```

Figure 12.4 Variables du système d'exploitation

À cause de la directive .BURN, le code et les lignes suivantes sont assemblés de telle manière que la dernière instruction se termine à l'adresse FFFF, c'est à dire la plus haute adresse de mémoire, ce qui indique que Pep 8 possède 64 Koctets de mémoire, la plus haute adresse étant 65535. L'adresse FC57 est la première adresse correspondant au code du chargeur, tandis que l'adresse FC9B est l'adresse de début du code des sous-programmes de traitement des interruptions qui suit.

```
;******* Trap handler
            oldIR:   .EQUATE 9          ;Stack address of IR on trap
            ;
FC9B  C80000 trap:    LDX     0,i       ;Clear X for a byte compare
FC9E  DB0009          LDBYTEX oldIR,s   ;X := trapped IR
FCA1  B80028          CPX     0x0028,i  ;If X >= first nonunary trap
opcode
FCA4  0EFCB7          BRGE    nonUnary  ;trap opcode is nonunary
            ;
FCA7  980003 unary:   ANDX    0x0003,i  ;Mask out all but rightmost two
bits
FCAA  1D              ASLX              ;An address is two bytes
FCAB  17FCAF          CALL    unaryJT,x ;Call unary trap routine
FCAE  01              RETTR             ;Return from trap
            ;
FCAF  FDB6   unaryJT: .ADDRSS opcode24  ;Address of NOP0 subroutine
FCB1  FDB7            .ADDRSS opcode25  ;Address of NOP1 subroutine
FCB3  FDB8            .ADDRSS opcode26  ;Address of NOP2 subroutine
FCB5  FDB9            .ADDRSS opcode27  ;Address of NOP3 subroutine
            ;
FCB7  1F     nonUnary:ASRX              ;Trap opcode is nonunary
FCB8  1F              ASRX              ;Discard addressing mode bits
FCB9  1F              ASRX
FCBA  880005          SUBX    5,i       ;Adjust so that NOP opcode = 0
FCBD  1D              ASLX              ;An address is two bytes
FCBE  17FCC2          CALL    nonUnJT,x ;Call nonunary trap routine
FCC1  01     return:  RETTR             ;Return from trap
            ;
FCC2  FDBA   nonUnJT: .ADDRSS opcode28  ;Address of NOP subroutine
FCC4  FDC4            .ADDRSS opcode30  ;Address of DECI subroutine
FCC6  FF3B            .ADDRSS opcode38  ;Address of DECO subroutine
FCC8  FFC6            .ADDRSS opcode40  ;Address of STRO subroutine
```

Figure 12.5 Traitement des interruptions

Le code qui suit comprend alors les huit sous-programmes de traitement des instructions dont le code instruction correspond à ce qu'on a vu plus haut. Viennent ensuite quelques sous-programmes internes au système et le code se termine par :

```
;******* Vectors for System Memory Format
FFF8  FBCF            .ADDRSS osRAM     ;User stack pointer
FFFA  FC4F            .ADDRSS wordBuff  ;System stack pointer
FFFC  FC57            .ADDRSS loader    ;Loader program counter
FFFE  FC9B            .ADDRSS trap      ;Trap program counter
            ;
                      .END
```

Figure 12.6 Adresses critiques du système d'exploitation

ce qui correspond au bas de la figure 12.2 (dernières adresses de la mémoire).

Le code de traitement des interruptions présenté ci-dessus ne fait qu'appeler le sous-programme de traitement voulu, dans le cas d'une instruction unaire ou non. Vous noterez qu'une fois un de ces sous-programme exécuté, on revient à l'adresse FCAE ou FCC1, où l'on exécute une instruction `RETTR` (retour d'une interruption). Cette dernière provoque la restauration des codes de condition à partir de la pile du système, la restauration du registre A, du registre X, du compteur ordinal, et du pointeur de pile (utilisateur). Si l'on regarde ce qu'on avait empilé, on remarque que la partie de l'instruction empilée sur la pile du système n'a pas été restaurée. Ceci est normal; en fait on a consommé cette partie de l'instruction pour déterminer de quelle instruction il s'agissait, afin de pouvoir appeler le sous-programme qui lui correspond (jetez un coup d'œil aux instructions commençant à l'adresse FC9B). Si l'instruction ayant provoqué l'interruption occupait un seul octet (instruction dite unaire), le compteur ordinal pointait déjà à la prochaine instruction, un octet plus loin que l'instruction ayant provoqué l'interruption. Si, par contre, l'instruction ayant provoqué l'interruption occupait trois octets (instruction en mode d'adressage `i`, `d`, ou `s`, dite non unaire), l'ancienne valeur du compteur ordinal avait déjà été augmentée de 2 pour pointer effectivement à l'instruction suivant celle qui a déclenché l'interruption. La valeur du compteur ordinal empilée pointe dans les deux cas à l'instruction suivante et est utilisée telle quelle par l'instruction `RETTR`.

12.5 PEP 8: traitement des interruptions des instructions non implantées

12.5.1 Vérification du mode d'adressage

```
            ;******* Assert valid trap addressing mode
            oldIR4:  .EQUATE 13         ;oldIR + 4 with two return addresses
FCCA  C00001 assertAd:LDA     1,i       ;A := 1
FCCD  DB000D          LDBYTEX oldIR4,s  ;X := OldIR
FCD0  980007          ANDX    0x0007,i  ;Keep only the addressing mode bits
FCD3  0AFCDD          BREQ    testAd    ;000 = immediate addressing
FCD6  1C      loop:   ASLA              ;Shift the 1 bit left
FCD7  880001          SUBX    1,i       ;Subtract from addressing mode count
FCDA  0CFCD6          BRNE    loop      ;Try next addressing mode
FCDD  91FC53 testAd:  ANDA    addrMask,d ;AND the 1 bit with legal modes
FCE0  0AFCE4          BREQ    addrErr
FCE3  58              RET0              ;Legal addressing mode, return
FCE4  50000A addrErr: CHARO   '\n',i
FCE7  C0FCF4          LDA     trapMsg,i ;Push address of error message
FCEA  E3FFFE          STA     -2,s
FCED  680002          SUBSP   2,i       ;Call print subroutine
FCF0  16FFE2          CALL    prntMsg
FCF3  00              STOP              ;Halt: Fatal runtime error
FCF4  455252 trapMsg: .ASCII  "ERROR: Invalid trap addressing mode.\x00"
      4F523A
      20496E
      76616C
      696420
      747261
      702061
      646472
      657373
      696E67
      206D6F
      64652E
      00
```

Figure 12.7 Sous-programme assertAd

Comme différentes instructions ont des modes d'adressage différents, le système PEP 8 peut automatiquement détecter un mode d'adressage interdit et une erreur d'adressage. Cependant, pour les instructions non implantées, le système de traitement des interruptions ne fait qu'exécuter

le code de traitement et il est nécessaire que ce code vérifie les modes d'adressage qui sont utilisés dans les nouvelles instructions. Le sous-programme `assertAd` vérifie les modes d'adressage pour les instructions ainsi simulées.

Ce sous-programme doit accéder au registre d'instruction empilé par l'interruption ; on le repère par `oldIR`. On a empilé deux adresses de retour depuis l'empilage des données de l'interruption, d'abord à cause d'un appel au sous-programme de traitement de l'interruption, comme `CALL nonUnJT,x`, issu du code de traitement de la figure 12.5, et ensuite à cause d'un appel à `assertAd` venu du sous-programme spécifique de traitement appelé, `opcodenn`. Ceci explique la valeur 13 trouvée dans le .EQUATE au lieu de la valeur 9 trouvée dans la figure 12.5 et dans la figure 12.3. Le sous-programme suppose que la variable du système d'exploitation `addrMask` comprend un masque indiquant les modes d'adressage permis (chose faite par le programme spécifique de traitement d'interruption, `opcodenn`). Si le mode d'adressage de l'instruction ayant provoqué l'interruption fait partie de l'ensemble décrit par le masque, le traitement se poursuit, sinon un message d'erreur est produit et le traitement est arrêté. Les huit bits du masque représentent dans l'ordre de gauche à droite : sxf, sx, x, sf, s, n, d, i ; ainsi, le masque 00100000 représente l'adressage indexé. Le test est fait simplement, premièrement par un positionnement par décalage à gauche du bit de mode d'adressage, puis par une opération ET logique qui devrait produire un bit 1 (le bit correspondant au mode choisi).

12.5.2 Calcul de l'adresse de l'opérande

Les sous-programmes de traitement spécifiques des interruptions qui ne sont pas unaires doivent calculer l'adresse de leur opérande en fonction du mode d'adressage de l'instruction interrompue.

```
            ;******* Set address of trap operand
            oldX4:   .EQUATE 7          ;oldX + 4 with two return addresses
            oldPC4:  .EQUATE 9          ;oldPC + 4 with two return addresses
            oldSP4:  .EQUATE 11         ;oldSP + 4 with two return addresses
FD19  DB000D setAddr: LDBYTEX oldIR4,s   ;X := old instruction register
FD1C  980007          ANDX    0x0007,i   ;Keep only the addressing mode bits
FD1F  1D              ASLX               ;An address is two bytes
FD20  05FD23          BR      addrJT,x
FD23  FD33   addrJT:  .ADDRSS addrI      ;Immediate addressing
FD25  FD3D            .ADDRSS addrD      ;Direct addressing
FD27  FD4A            .ADDRSS addrN      ;Indirect addressing
FD29  FD5A            .ADDRSS addrS      ;Stack relative addressing
FD2B  FD6A            .ADDRSS addrSF     ;Stack relative deferred addressing
FD2D  FD7D            .ADDRSS addrX      ;Indexed addressing
FD2F  FD8D            .ADDRSS addrSX     ;Stack indexed addressing
FD31  FDA0            .ADDRSS addrSXF    ;Stack indexed deferred addressing
            ;
FD33  CB0009 addrI:   LDX     oldPC4,s   ;Immediate addressing
FD36  880002          SUBX    2,i        ;Oprnd = OprndsSpec
FD39  E9FC55          STX     opAddr,d
FD3C  58              RET0
            ;
FD3D  CB0009 addrD:   LDX     oldPC4,s   ;Direct addressing
FD40  880002          SUBX    2,i        ;Oprnd = Mem[OprndSpec]
FD43  CD0000          LDX     0,x
FD46  E9FC55          STX     opAddr,d
FD49  58              RET0
            ;
FD4A  CB0009 addrN:   LDX     oldPC4,s   ;Indirect addressing
FD4D  880002          SUBX    2,i        ;Oprnd = Mem[Mem[OprndSpec]]
FD50  CD0000          LDX     0,x
FD53  CD0000          LDX     0,x
FD56  E9FC55          STX     opAddr,d
FD59  58              RET0
```



```
              ;
FD5A  CB0009 addrS:   LDX     oldPC4,s   ;Stack relative addressing
FD5D  880002          SUBX    2,i        ;Oprnd = Mem[SP + OprndSpec]
FD60  CD0000          LDX     0,x
FD63  7B000B          ADDX    oldSP4,s
FD66  E9FC55          STX     opAddr,d
FD69  58              RET0
              ;
FD6A  CB0009 addrSF:  LDX     oldPC4,s   ;Stack relative deferred addressing
FD6D  880002          SUBX    2,i        ;Oprnd = Mem[Mem[SP + OprndSpec]]
FD70  CD0000          LDX     0,x
FD73  7B000B          ADDX    oldSP4,s
FD76  CD0000          LDX     0,x
FD79  E9FC55          STX     opAddr,d
FD7C  58              RET0
              ;
FD7D  CB0009 addrX:   LDX     oldPC4,s   ;Indexed addressing
FD80  880002          SUBX    2,i        ;Oprnd = Mem[OprndSpec + X]
FD83  CD0000          LDX     0,x
FD86  7B0007          ADDX    oldX4,s
FD89  E9FC55          STX     opAddr,d
FD8C  58              RET0
              ;
FD8D  CB0009 addrSX:  LDX     oldPC4,s   ;Stack indexed addressing
FD90  880002          SUBX    2,i        ;Oprnd = Mem[SP + OprndSpec + X]
FD93  CD0000          LDX     0,x
FD96  7B0007          ADDX    oldX4,s
FD99  7B000B          ADDX    oldSP4,s
FD9C  E9FC55          STX     opAddr,d
FD9F  58              RET0
              ;
FDA0  CB0009 addrSXF: LDX     oldPC4,s   ;Stack indexed deferred addressing
FDA3  880002          SUBX    2,i        ;Oprnd = Mem[Mem[SP + OprndSpec] + X]
FDA6  CD0000          LDX     0,x
FDA9  7B000B          ADDX    oldSP4,s
FDAC  CD0000          LDX     0,x
FDAF  7B0007          ADDX    oldX4,s
FDB2  E9FC55          STX     opAddr,d
FDB5  58              RET0
```

Figure 12.8 Sous-programme setAddr

Le sous-programme `setAddr` effectue ce calcul. Il suppose que les informations relatives à l'interruption sont sur la pile du système et retourne l'adresse recherchée dans la variable `opAddr` du système d'exploitation. Comme pour le sous-programme `assertAd`, et pour les mêmes raisons, le sous-programme `setAddr` doit ajouter 4 aux positions des informations d'interruption sur la pile. À partir du registre d'instruction empilé, le sous-programme détermine le mode d'adressage et effectue un saut à l'un des huit calculs de l'adresse de l'opérande en fonction des huit modes d'adressage possibles. Le compteur ordinal d'une instruction non unaire pointe à la prochaine instruction ; pour avoir le code de l'instruction il faut lui soustraire 2, ce que fait le code de chacun des huit cas traités. Le calcul de l'adresse effectue par programme ce que le processeur fait automatiquement pendant l'exécution d'une instruction.

12.5.3 Traitement des instructions NOP

Les sous-programmes de traitement spécifiques des interruptions unaires `NOPn` qui sont définis dans le système PEP 8 ne font rien ; ils peuvent cependant permettre à qui en a besoin de définir de nouvelles opérations. Cependant, comme ces quatre instructions n'ont pas d'opérandes, le traitement réel qu'elles peuvent faire est limité ; elles vous donnent cependant la possibilité de définir quatre nouvelles instructions *unaires*.

L'instruction `NOP`, quant à elle, n'est pas unaire et a un opérande en mode d'adressage immédiat. Le traitement du système d'exploitation s'assure que le mode d'adressage est conforme à la définition de l'instruction, mais en dehors de cela ne fait rien de spécifique. Là encore, le code peut être modifié pour un traitement plus complet, correspondant aux besoins spécifiques de l'utilisateur : nouvelles instructions traitant un ensemble de possibilités selon la valeur de l'opérande immédiat, par exemple.

```
             ;******* Opcode 0x24
             ;The NOP0 instruction.
FDB6  58     opcode24:RET0
             ;
             ;******* Opcode 0x25
             ;The NOP1 instruction.
FDB7  58     opcode25:RET0
             ;
             ;******* Opcode 0x26
             ;The NOP2 instruction.
FDB8  58     opcode26:RET0
             ;
             ;******* Opcode 0x27
             ;The NOP3 instruction.
FDB9  58     opcode27:RET0
             ;
             ;******* Opcode 0x28
             ;The NOP instruction.
FDBA  C00001 opcode28:LDA     0x0001,i   ;Assert i
FDBD  E1FC53          STA     addrMask,d
FDC0  16FCCA          CALL    assertAd
FDC3  58              RET0
```

Figure 12.9 Traitement des instructions NOP

12.5.4 Traitement de l'instruction `DECI`

Le sous-programme de traitement spécifique à l'instruction `DECI` est le plus long, car il doit traiter un certain nombre de possibilités. En effet la valeur entière donnée peut commencer par un nombre quelconque d'espaces ou de fins de ligne, un signe plus ou moins ou un chiffre décimal; elle peut aussi comprendre trop de chiffres et créer une valeur trop grande (débordement). La valeur est lue caractère par caractère et, en fonction des caractères rencontrés, est composée numériquement par additions et multiplications. Le sous-programme positionne également les codes de condition, à l'exception de la retenue (*C*).

```
             ;******* Opcode 0x30
             ;The DECI instruction.
             ;Input format: Any number of leading spaces or line feeds are
             ;allowed, followed by '+', '-' or a digit as the first character,
             ;after which digits are input until the first nondigit is
             ;encountered. The status flags N,Z and V are set appropriately
             ;by this DECI routine. The C status flag is not affected.
             oldNZVC: .EQUATE 14         ;Stack address of NZVC on interrupt
             total:   .EQUATE 10         ;Cumulative total of DECI number
             valAscii:.EQUATE 8          ;Value(asciiCH)
             isOvfl:  .EQUATE 6          ;Overflow boolean
             isNeg:   .EQUATE 4          ;Negative boolean
             state:   .EQUATE 2          ;State variable
             temp:    .EQUATE 0
             ;
             init:    .EQUATE 0          ;Enumerated values for state
             sign:    .EQUATE 1
             digit:   .EQUATE 2
FDC4  C000FE opcode30:LDA     0x00FE,i   ;Assert d, n, s, sf, x, sx, sxf
FDC7  E1FC53          STA     addrMask,d
FDCA  16FCCA          CALL    assertAd
```

```
FDCD  16FD19          CALL    setAddr    ;Set address of trap operand
FDD0  68000C          SUBSP   12,i       ;Allocate storage for locals
FDD3  C00000          LDA     FALSE,i    ;isOvfl := FALSE
FDD6  E30006          STA     isOvfl,s
FDD9  C00000          LDA     init,i     ;state := init
FDDC  E30002          STA     state,s
FDDF  C00000          LDA     0,i        ;wordBuff := 0 for input
FDE2  E1FC4F          STA     wordBuff,d
FDE5  49FC50 do:      CHARI   byteBuff,d ;Get asciiCh
FDE8  C1FC4F          LDA     wordBuff,d ;Set value(asciiCH)
FDEB  90000F          ANDA    0x000F,i
FDEE  E30008          STA     valAscii,s
FDF1  C1FC4F          LDA     wordBuff,d ;A = asciiCh throughout the loop
FDF4  CB0002          LDX     state,s    ;switch (state)
FDF7  1D              ASLX               ;An address is two bytes
FDF8  05FDFB          BR      stateJT,x
FDFB  FE01   stateJT: .ADDRSS sInit
FDFD  FE5B            .ADDRSS sSign
FDFF  FE76            .ADDRSS sDigit
FE01  B0002B sInit:   CPA     '+',i      ;if (asciiCh == '+')
FE04  0CFE16          BRNE    ifMinus
FE07  C80000          LDX     FALSE,i    ;isNeg := FALSE
FE0A  EB0004          STX     isNeg,s
FE0D  C80001          LDX     sign,i     ;state := sign
FE10  EB0002          STX     state,s
FE13  04FDE5          BR      do
FE16  B0002D ifMinus: CPA     '-',i      ;else if (asciiCh == '-')
FE19  0CFE2B          BRNE    ifDigit
FE1C  C80001          LDX     TRUE,i     ;isNeg := TRUE
FE1F  EB0004          STX     isNeg,s
FE22  C80001          LDX     sign,i     ;state := sign
FE25  EB0002          STX     state,s
FE28  04FDE5          BR      do
FE2B  B00030 ifDigit: CPA     '0',i      ;else if (asciiCh is a digit)
FE2E  08FE4C          BRLT    ifWhite
FE31  B00039          CPA     '9',i
FE34  10FE4C          BRGT    ifWhite
FE37  C80000          LDX     FALSE,i    ;isNeg := FALSE
FE3A  EB0004          STX     isNeg,s
FE3D  CB0008          LDX     valAscii,s ;total := Value(asciiCh)
FE40  EB000A          STX     total,s
FE43  C80002          LDX     digit,i    ;state := digit
FE46  EB0002          STX     state,s
FE49  04FDE5          BR      do
FE4C  B00020 ifWhite: CPA     ' ',i      ;else if (asciiCh is not a space
FE4F  0AFDE5          BREQ    do
FE52  B0000A          CPA     '\n',i     ;or line feed)
FE55  0CFF11          BRNE    deciErr    ;exit with DECI error
FE58  04FDE5          BR      do
FE5B  B00030 sSign:   CPA     '0',i      ;if asciiCh (is not a digit)
FE5E  08FF11          BRLT    deciErr
FE61  B00039          CPA     '9',i
FE64  10FF11          BRGT    deciErr    ;exit with DECI error
FE67  CB0008          LDX     valAscii,s ;else total := Value(asciiCh)
FE6A  EB000A          STX     total,s
FE6D  C80002          LDX     digit,i    ;state := digit
FE70  EB0002          STX     state,s
FE73  04FDE5          BR      do
FE76  B00030 sDigit:  CPA     '0',i      ;if (asciiCh is not a digit)
FE79  08FEC7          BRLT    deciNorm
FE7C  B00039          CPA     '9',i
FE7F  10FEC7          BRGT    deciNorm   ;exit normaly
FE82  C80001          LDX     TRUE,i     ;else X := TRUE for later assignments
FE85  C3000A          LDA     total,s    ;Multiply total by 10 as follows:
FE88  1C              ASLA               ;First, times 2
FE89  12FE8F          BRV     ovfl1      ;If overflow then
FE8C  04FE92          BR      L1
FE8F  EB0006 ovfl1:   STX     isOvfl,s   ;isOvfl := TRUE
FE92  E30000 L1:      STA     temp,s     ;Save 2 * total in temp
FE95  1C              ASLA               ;Now, 4 * total
FE96  12FE9C          BRV     ovfl2      ;If overflow then
```

```
FE99  04FE9F          BR      L2
FE9C  EB0006 ovfl2:   STX     isOvfl,s   ;isOvfl := TRUE
FE9F  1C     L2:      ASLA               ;Now, 8 * total
FEA0  12FEA6          BRV     ovfl3      ;If overflow then
FEA3  04FEA9          BR      L3
FEA6  EB0006 ovfl3:   STX     isOvfl,s   ;isOvfl := TRUE
FEA9  730000 L3:      ADDA    temp,s     ;Finally, 8 * total + 2 * total
FEAC  12FEB2          BRV     ovfl4      ;If overflow then
FEAF  04FEB5          BR      L4
FEB2  EB0006 ovfl4:   STX     isOvfl,s   ;isOvfl := TRUE
FEB5  730008 L4:      ADDA    valAscii,s ;A := 10 * total + valAscii
FEB8  12FEBE          BRV     ovfl5      ;If overflow then
FEBB  04FEC1          BR      L5
FEBE  EB0006 ovfl5:   STX     isOvfl,s   ;isOvfl := TRUE
FEC1  E3000A L5:      STA     total,s    ;Update total
FEC4  04FDE5          BR      do
FEC7  C30004 deciNorm:LDA     isNeg,s    ;If isNeg then
FECA  0AFEE3          BREQ    setNZ
FECD  C3000A          LDA     total,s    ;If total != 0x8000 then
FED0  B08000          CPA     0x8000,i
FED3  0AFEDD          BREQ    L6
FED6  1A              NEGA               ;Negate total
FED7  E3000A          STA     total,s
FEDA  04FEE3          BR      setNZ
FEDD  C00000 L6:      LDA     FALSE,i    ;else -32768 is a special case
FEE0  E30006          STA     isOvfl,s   ;isOvfl := FALSE
FEE3  DB000E setNZ:   LDBYTEX oldNZVC,s  ;Set NZ according to total result:
FEE6  980001          ANDX    0x0001,i   ;First initialize NZV to 000
FEE9  C3000A          LDA     total,s    ;If total is negative then
FEEC  0EFEF2          BRGE    checkZ
FEEF  A80008          ORX     0x0008,i   ;set N to 1
FEF2  B00000 checkZ:  CPA     0,i        ;If total is not zero then
FEF5  0CFEFB          BRNE    setV
FEF8  A80004          ORX     0x0004,i   ;set Z to 1
FEFB  C30006 setV:    LDA     isOvfl,s   ;If not isOvfl then
FEFE  0AFF04          BREQ    storeFl
FF01  A80002          ORX     0x0002,i   ;set V to 1
FF04  FB000E storeFl: STBYTEX oldNZVC,s  ;Store the NZVC flags
FF07  C3000A exitDeci:LDA     total,s    ;Put total in memory
FF0A  E2FC55          STA     opAddr,n
FF0D  60000C          ADDSP   12,i       ;Deallocate locals
FF10  58              RET0               ;Return to trap handler
FF11  50000A deciErr: CHARO   '\n',i
FF14  C0FF21          LDA     deciMsg,i  ;Push address of message onto stack
FF17  E3FFFE          STA     -2,s
FF1A  680002          SUBSP   2,i
FF1D  16FFE2          CALL    prntMsg    ;and print
FF20  00              STOP               ;Fatal error: program terminates
FF21  455252 deciMsg: .ASCII  "ERROR: Invalid DECI input\x00"
      4F523A
      20496E
      76616C
      696420
      444543
      492069
      6E7075
      7400
```

Figure 12.10 Traitement de l'instruction `DECI`

Une fois le mode d'adressage vérifié, le sous-programme réserve l'espace local sur la pile pour ses variables locales (12 octets). Le sous-programme comporte une boucle `do` dans laquelle on détermine l'action à prendre pour chaque caractère lu : signe plus (étiquette `sInit`), signe moins (`ifMinus`), premier chiffre décimal (`ifDigit`), espace ou fin de ligne (`ifWhite`), chiffre décimal (`sDigit`) ou fin du nombre sur caractère non chiffre décimal (`deciNorm`). La multiplication par 10 est faite par décalage successifs, à raison de 3 décalages pour « fois 8 » et d'une addition du double (étiquettes `L1`, `L2`, `L3` et `L4`). Il reste les instructions pour la définition des codes de

condition (`setNZ`, `checkZ`, `setV`) et le code de fin du sous-programme (rangement de la valeur lue à l'adresse calculée `opAddr`), sans oublier le code correspondant à l'erreur.

12.5.5 Traitement de l'instruction DECO

Le sous-programme de traitement spécifique à l'instruction `DECO` est plus simple que le précédent, car il ne doit sortir que cinq caractères numériques au maximum, précédés d'un signe moins, si nécessaire.

```
            ;******* Opcode 0x38
            ;The DECO instruction.
            ;Output format: If the operand is negative, the algorithm prints
            ;a single '-' followed by the magnitude. Otherwise it prints the
            ;magnitude without a leading '+'. It suppresses leading zeros.
            ;
            remain:  .EQUATE 0          ;Remainder of value to output
            chOut:   .EQUATE 2          ;Has a character been output yet?
            place:   .EQUATE 4          ;Place value for division
            ;
FF3B  C000FF opcode38:LDA     0x00FF,i   ;Assert i, d, n, s, sf, x, sx, sxf
FF3E  E1FC53          STA     addrMask,d
FF41  16FCCA          CALL    assertAd
FF44  16FD19          CALL    setAddr    ;Set address of trap operand
FF47  680006          SUBSP   6,i        ;Allocate storage for locals
FF4A  C2FC55          LDA     opAddr,n   ;A := oprnd
FF4D  B00000          CPA     0,i        ;If oprnd is negative then
FF50  0EFF57          BRGE    printMag
FF53  50002D          CHARO   '-',i      ;Print leading '-' and
FF56  1A              NEGA               ;make magnitude positive
FF57  E30000 printMag:STA     remain,s   ;remain := abs(oprnd)
FF5A  C00000          LDA     FALSE,i    ;Initialize chOut := FALSE
FF5D  E30002          STA     chOut,s
FF60  C02710          LDA     10000,i    ;place := 10,000
FF63  E30004          STA     place,s
FF66  16FF91          CALL    divide     ;Write 10,000's place
FF69  C003E8          LDA     1000,i     ;place := 1,000
FF6C  E30004          STA     place,s
FF6F  16FF91          CALL    divide     ;Write 1000's place
FF72  C00064          LDA     100,i      ;place := 100
FF75  E30004          STA     place,s
FF78  16FF91          CALL    divide     ;Write 100's place
FF7B  C0000A          LDA     10,i       ;place := 10
FF7E  E30004          STA     place,s
FF81  16FF91          CALL    divide     ;Write 10's place
FF84  C30000          LDA     remain,s   ;Always write 1's place
FF87  A00030          ORA     0x0030,i   ;Convert decimal to ASCII
FF8A  F1FC50          STBYTEA byteBuff,d
FF8D  51FC50          CHARO   byteBuff,d
FF90  5E              RET6
            ;
            ;Subroutine to print the most significant decimal digit of the
            ;remainder. It assumes that place (place2 here) contains the
            ;decimal place value. It updates the remainder.
            ;
            remain2: .EQUATE 2          ;Stack addresses while executing a
            chOut2:  .EQUATE 4          ;subroutine are greater by two becaus
            place2:  .EQUATE 6          ;the retAddr is on the stack
            ;
FF91  C30002 divide:  LDA     remain2,s  ;A := remainder
FF94  C80000          LDX     0,i        ;X := 0
FF97  830006 divLoop: SUBA    place2,s   ;Division by repeated subtraction
FF9A  08FFA6          BRLT    writeNum   ;If remainder is negative then done
FF9D  780001          ADDX    1,i        ;X := X + 1
FFA0  E30002          STA     remain2,s  ;Store the new remainder
FFA3  04FF97          BR      divLoop
FFA6  B80000 writeNum:CPX     0,i        ;If X != 0 then
FFA9  0AFFB5          BREQ    checkOut
FFAC  C00001          LDA     TRUE,i     ;chOut := TRUE
```

```
FFAF  E30004          STA     chOut2,s
FFB2  04FFBC          BR      printDgt   ;and branch to print this digit
FFB5  C30004 checkOut:LDA     chOut2,s   ;else if a previous char was output
FFB8  0CFFBC          BRNE    printDgt   ;then branch to print this zero
FFBB  58              RET0               ;else return to calling routine
                ;
FFBC  A80030 printDgt:ORX     0x0030,i   ;Convert decimal to ASCII
FFBF  E9FC4F          STX     wordBuff,d ;for output
FFC2  51FC50          CHARO   byteBuff,d
FFC5  58              RET0               ;return to calling routine
```

Figure 12.11 Traitement de l'instruction `DECO`

Le code commence par vérifier le mode d'adressage, obtenir l'adresse de l'opérande dans `opAddr` et réserver 6 octets pour trois variables locales. Il affiche ensuite le signe moins si la valeur est négative, puis affiche la valeur. Pour ce faire, il appelle un sous-programme interne de division, `divide`, qui soustrait la valeur rangée dans `place2` tant que cela est possible et place le quotient dans X et le reste dans `remain2`. Si la valeur de X est nulle, on vérifie pour voir si un chiffre a déjà été affiché ; si c'est le cas, on affiche le zéro, sinon, on ne l'affiche pas. Lorsqu'on trouve un chiffre non nul, on met l'indicateur `chOut2` à vrai pour s'assurer de sortir les zéros du reste du nombre. Notez la façon dont les variables locales de `opcode38` sont partagées par le sous-programme interne `divide` qui redéfinit leur déplacement dans la pile en ajoutant 2 à cause de l'adresse de retour qui s'est ajoutée sur la pile lors de son appel.

12.5.6 Traitement de l'instruction `STRO`

Le sous-programme de traitement spécifique à l'instruction `STRO` est très court : il vérifie le mode d'adressage, puis obtient l'adresse de l'opérande qu'il empile avant d'appeler le sous-programme `prntMsg`.

```
                ;******* Opcode 0x40
                ;The STRO instruction.
                ;Outputs a null-terminated string from memory.
                ;
FFC6  C00016 opcode40:LDA     0x0016,i   ;Assert d, n, sf
FFC9  E1FC53          STA     addrMask,d
FFCC  16FCCA          CALL    assertAd
FFCF  16FD19          CALL    setAddr    ;Set address of trap operand
FFD2  C1FC55          LDA     opAddr,d   ;Push address of string to print
FFD5  E3FFFE          STA     -2,s
FFD8  680002          SUBSP   2,i
FFDB  16FFE2          CALL    prntMsg    ;and print
FFDE  600002          ADDSP   2,i
FFE1  58              RET0
                ;
                ;******* Print subroutine
                ;Prints a string of ASCII bytes until it encounters a null
                ;byte (eight zero bits). Assumes one parameter, which
                ;contains the address of the message.
                ;
                msgAddr: .EQUATE 2          ;Address of message to print
                ;
FFE2  C80000 prntMsg: LDX     0,i        ;X := 0
FFE5  C00000          LDA     0,i        ;A := 0
FFE8  D70002 prntMore:LDBYTEA msgAddr,sxf;Test next char
FFEB  0AFFF7          BREQ    exitPrnt   ;If null then exit
FFEE  570002          CHARO   msgAddr,sxf;else print
FFF1  780001          ADDX    1,i        ;X := X + 1 for next character
FFF4  04FFE8          BR      prntMore
                ;
FFF7  58      exitPrnt:RET0
```

Ce dernier traite le paramètre comme l'adresse d'un vecteur de caractères, équivalent à une chaîne, et accède aux caractères individuels au moyen du mode d'adressage sur la pile indirect indexé. On accède à chaque caractère pour vérifier la fin de la chaîne (caractère de code zéro) ou pour le sortir au moyen d'une instruction `CHARO`.

12.5.7 Exemple de traitement d'une instruction causant une interruption

Pour mieux fixer les idées, nous allons suivre pas à pas l'exécution d'une instruction DECI extraite d'un programme du chapitre 7.

```
.......
000C  1D              ASLX                ;    //entier = 2 octets
000D  350048          DECI    vecteur,x   ;    cin >> vector[i];
0010  C90060          LDX     index,d     ;
.......
```

Lors du décodage de l'instruction, le processeur a placé dans le registre d'instruction le premier octet de l'instruction : 0x35, soit en binaire 00110101. L'inspection de l'Annexe C nous montre que l'instruction ayant pour code 00110 est l'instruction DECI ; son décodage provoque donc une interruption. Le système, utilisant le pointeur de pile système dont la valeur se trouve à l'adresse 0xFFFA, empile 10 octets sur la pile système, tel qu'illustré par la figure 12.3 ; il place ensuite le contenu de l'adresse 0xFFFE dans le compteur ordinal et poursuit l'exécution. La prochaine instruction exécutée est donc l'instruction d'étiquette `trap`, située à l'adresse FC9B illustrée dans la figure 12.5.

Le registre X est mis à zéro et on y place le premier octet empilé sur la pile système, soit le registre instruction comprenant l'instruction ayant provoqué l'interruption. Dans le cas de notre exemple, il vaut 0x35, qui est supérieur à 0x28 et provoque un branchement à `nonUnary`. Trois décalages à droite éliminent les trois derniers bits (mode d'adressage) de l'instruction, pour ne conserver que le code opération, auquel on enlève 5, obtenant une valeur égale à 1, laquelle est multipliée par 2 et utilisée comme indice dans une table d'adresse ; dans ce cas, on effectue un appel à la seconde adresse de la table : `opcode30`.

Cette étiquette est le début du sous-programme de traitement de l'instruction DECI, que l'on retrouve à la figure 12.10. On définit alors un masque qui permet tous les modes d'adressage sauf le mode immédiat, de code zéro (d, n, s, sf, x, sx, sxf). On appelle ensuite le sous-programme `assertAd`, qui se trouve à la figure 12.7, et qui vérifie que le mode d'adressage est bien valide pour l'instruction. Le sous-programme commence par placer 1 dans le registre A et par récupérer le contenu du registre d'instruction dans la pile système ; cette dernière est telle que décrite par la figure 12.3, avec en plus à son sommet deux adresses de retour, car il y a eu l'appel à `opcode30`, lequel a été suivi de l'appel à `assertAd`. On ne conserve que les trois bits du mode d'adressage et on effectue autant de décalages à gauche du contenu du registre A que la valeur du mode d'adressage, obtenant la valeur binaire 100000, avec laquelle on fait un ET logique du masque (0xFE) donnant la même valeur ; cette dernière n'étant pas nulle, on revient à l'appelant (adresse FDCD).

De retour dans le sous-programme de traitement de DECI, on va maintenant calculer l'adresse effective de l'opérande au moyen d'un appel à `setAddr` (figure 12.8). On y retourne chercher le registre instruction, pour ne conserver que le mode d'adressage, lequel est utilisé pour choisir l'adresse où continuer le traitement. Comme le mode d'adressage vaut 5, on utilise la sixième étiquette de la table, soit `addrX`, où l'on traite l'adressage indexé. On prend la valeur du compteur ordinal empilée (soit l'adresse de l'instruction suivant celle qui a provoqué l'interruption), on lui soustrait 2 pour retrouver l'adresse de la partie opérande de l'instruction traitée, laquelle est utilisée pour récupérer la partie opérande de l'instruction, c'est-à-dire ici 0x0048. On y ajoute le contenu du registre X sauvegardé dans `oldX4` (`oldX` dans la figure 12.3 aurait la valeur 3, et on y ajoute 4, car il y a eu empilement sur la pile système de deux adresses de retour : appels à `opcode30` et à `setAddr`) et on range le résultat dans `opAddr`.

De retour dans le sous-programme de traitement de DECI, on va maintenant réserver l'espace local nécessaire au traitement ; notez que l'on n'a pas fait cette réservation plus tôt pour permettre un accès plus facile aux éléments sauvegardés sur la pile par l'interruption. On met l'indicateur de débordement à faux, l'état à `init` et on place zéro dans l'octet précédant celui qu'on va lire. Dans la boucle `do`, on répète les choses suivantes : lecture d'un caractère, conservation des quatre derniers bits de ce caractère (valeur numérique du caractère) dans `valAscii`, et ensuite saut à la partie du traitement correspondant à l'état.

La première fois on saute à `sInit`, où l'on vérifie le premier caractère lu ; on supposera pour cet exemple, que l'on a tapé les caractères 493\n. On vérifie si ce premier caractère est un signe +, comme ce n'est pas le cas, on vérifie si c'est un signe - ; comme ce n'est pas le cas, on vérifie si c'est un caractère numérique, et comme c'est le cas, on place faux dans l'indicateur de négatif, on place `valAscii` dans `Total` et on change l'état à `digit` avant de retourner à `do`.

On lit le second caractère, on en place la valeur numérique dans `valAscii` et on branche à l'étiquette `sDigit` en fonction de l'état dans lequel on se trouve. On vérifie alors s'il s'agit d'un caractère numérique. C'est le cas, on récupère donc `Total` et on le multiplie par dix : on le multiplie par 2 et on sauvegarde cette valeur, on remultiplie la valeur obtenue par 2 à deux reprises ce qui donne une multiplication par 8, à laquelle on ajoute la valeur sauvegardée. Évidemment, après chaque multiplication, on doit vérifier s'il y a eu débordement et enregistrer ce fait. Une fois la multiplication par 10 effectuée, on ajoute `valAscii` au résultat que l'on range dans `Total`, avant de revenir à l'étiquette `do`.

On lit le troisième caractère, on en place la valeur numérique dans `valAscii` et on branche à l'étiquette `sDigit` en fonction de l'état, qui n'a pas changé. On vérifie qu'on a un caractère numérique, et comme c'est bien le cas, on répète le traitement précédent : multiplication par 10 et addition, et retour à `do`. On lit le prochain caractère (une fin de ligne) et on aboutit à `sDigit` ; on y découvre que le caractère lu n'est pas numérique, ce qui nous envoie à l'étiquette `deciNorm`.

Si le résultat est négatif, on le remplace par son complément à 2, sauf s'il s'agit de la valeur –32768 (0x8000), qui est conservée telle quelle. On met les bits NZV à zéro et si le résultat est négatif on met le bit N à 1 ; si le résultat est nul on met le bit Z à 1 ; s'il y a eu débordement au cours des calculs, ce qui a été enregistré dans `isOvfl`, on met le bit V à 1. On remplace alors les

indicateurs NZVC rangés sur la pile système, et on range, au moyen d'une indirection, le total calculé en mémoire, à l'adresse calculée par `setAddr` et rangée dans `opAddr`. On libère la mémoire locale et on retourne au sous-programme de traitement des interruptions à l'adresse FCC1, où l'on exécute une instruction RETTR qui restaure les registres, le compteur ordinal, les codes de condition, et le pointeur de pile avant de continuer le traitement par l'instruction suivant celle qui a provoqué l'interruption.

12.5.8 Exercices

1. Dans la figure 12.7, expliquez le rôle de la boucle de décalage `loop`. Dans cette même figure, expliquez l'appel `CALL prntMsg` ; pourquoi ne pas utiliser l'instruction `STRO` ?
2. Dans la figure 12.8, peu après l'étiquette `addrN`, expliquez pourquoi on a besoin de répéter l'instruction `LDX 0,x`. Dans la même figure, peu après l'étiquette `addrS`, dites quelle valeur est placée dans X par l'instruction `LDX 0,x`. Toujours dans cette figure, peu après l'étiquette `addrSX`, donnez le contenu de X après chacune des instructions `ADDX`. Enfin, peu après l'étiquette `addrSXF`, indiquez ce que contient X après exécution de la seconde instruction `LDX 0,x`.
3. Dans la figure 12.10, dites si les symboles `init`, `sign` et `digit` sont des variables locales. Dans cette même figure, peu après l'étiquette `do`, indiquez la raison d'être de l'instruction `ANDA 0x000F,i`. Peu après l'étiquette `ifDigit`, donnez le contenu de total après exécution de l'instruction `STX total,s`. Peu après l'étiquette `ifWhite`, pourquoi va-t-on à `deciErr` si on ne trouve pas d'espace ou de saut de ligne ? Peu après l'étiquette `deciNorm`, indiquez le rôle de la comparaison `CPA 0x8000,i`. Peu après l'étiquette `setNZ`, pourquoi effectue-t-on un ET logique avec la valeur 1 ? À l'étiquette `storeFl`, où range-t-on la valeur ?
4. Dans la figure 12.11, à l'étiquette `printDgt`, à quoi sert l'instruction `ORX 0x0030,i` ?
5. Dans la figure 12.12, expliquez la raison de l'instruction `ADDSP` située à l'adresse FFDE.

