CALM

Common **A**ssembly **L**anguage for **M**icroprocessors

Instructions création fichier .PRO

Patrick Fäh, La Colombière, CH-1783 Pensier, Suisse

Analyse d'instruction et génération de code

1. Introduction

Avant d'expliquer l'analyse réalisée d'une instruction, nous allons d'abord mettre en évidence le problème de l'analyse d'instruction. La structure générale d'une instruction est:

code opératoire opérande1, opérande2, ...

Éventuellement, il y a en plus des codes conditions et des indicateurs de données. Exemple:

```
      MOVE          B,A
On pourrait décomposer l'instruction comme suit:
      code opératoire:       MOVE
      opérande1:                  B
      opérande2:                  A
```

Cette décomposition ne pose pas de problèmes si les parties sont simples. Quelques exemples illustrent les problèmes:

```
      ADD.8         #10,{A0}+32^{D0}+9
      PUSHM.32      R0|R4..R7
      SKIP,EQ DJ.32,NMO      D0,ADRESSE
      RET
      MUL.32        R0,R1,R2
      TEST          {SP}+{R0}*4:{{SB}+10}+20.A8
      JUMP,EQ       R16^ADRESSE
```

Le code opératoire, les indicateurs de données éventuels et les codes permettent pas beaucoup de possibilités. L'analyse d'un opérande est nettement plus complexe. Le sens d'une instruction (= code machine à générer) est souvent déterminé par le code opératoire, les indicateurs de données et les opérandes. Par exemple, l'assembleur CALM ne peut pas - après avoir reconnu le code opératoire MOVE - générer le code machine correspondant et analyser indépendamment les opérandes.

Il est tout à fait possible que plusieurs noms de codes opératoires d'un fabricant correspond à un code opératoire unique de CALM. Par exemple, les codes opératoires du 68010 comme MOVE, MOVEA, MOVEC, MOVEQ, MOVES et LEA sont remplacés en CALM par MOVE. Les opérandes doivent donc apporter les caractéristiques. L'assembleur CALM doit alors savoir, "d'où il vient", c.à.d. ce qu'il a déjà généré.

De plus, il faut tenir compte que le programme principal (l'assembleur CALM) charge une description de processeur. Il n'y a pas de sousroutines spécifiques au processeur. Le même programme principal doit générer un code machine pour un 8080, 8086, 68000 ou un 32032. Uniquement la description de processeur s'adapte au processeur correspondant.

C'est à dire, nous avons le problème suivant: il faut analyser une chaîne de caractères dont nous supposons qu'elle contienne une instruction. Vu la multiplicité des structures d'instruction, il faut à la fois décomposer et analyser une instruction.

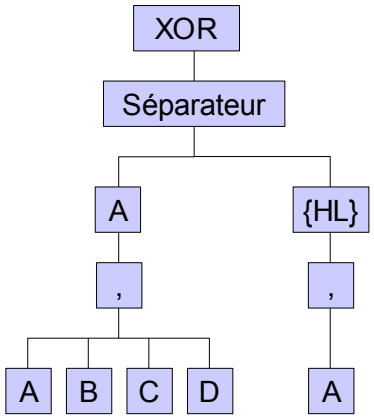
L'analyse d'instruction réalisée ici utilise une méthode qui consiste d'une série de décisions.
La description de processeur a une structure d'arbre. L'assembleur détermine le code opératoire et le compare à tous les noms possibles dans la table des codes opératoires de la description de processeur (fichier .PRO).
Si l'assembleur trouve le nom, l'analyse continue à cette branche. Ensuite, toute une liste de tests et d'instructions suit dans la description de texte suivant et ce qu'il doit générer.
La ligne d'entrée (ASLINE) contient l'instruction à analyser. Un pointeur (PLINE) montre sur le caractère actuel. Si le test est positif, le pointeur est avancé. Sinon, il reste immobile et le test suivant est exécuté.

2. Un exemple simple

Supposons qu'une instruction XOR a les possibilités suivantes:

```
XOR   A,A
XOR   A,B
XOR   A,C
XOR   A,D
XOR   {HL},A
```

Donc, la structure d'arbre a la structure suivante:



On peut représenter cette structure d'arbre d'une manière plus abstraite:

```
instruction   =   "XOR" séparateur (Xor1 | Xor2)
Xor1          =   RegA virgule RegABCD
Xor2          =   RegIHL virgule RegA
RegABCD       =   RegA | "B" | "C" | "D"
RegA          =   "A"
RegIHL        =   "{HL}"
virgule       =   [séparateur]","[séparateur]
séparateur    =   "<espace>" | "<TAB>"
```

Cette représentation paraît à première vue peu claire par rapport à la précédante. Mais cette représentation abstraite montre plus clairement comment on pourrait réaliser une telle structure. On remarque aussi qu'on peut rassembler des éléments. En principe, cette représentation contient tout pour analyser correctement une instruction XOR (la génération de code manque encore). L'assembleur devrait lire ces lignes et générer internement la structure d'arbre correspondante.

La version réalisée va au-delà. Elle passe directement la structure d'arbre. Cette structure d'arbre est basée sur un langage qui est interprétée par l'assembleur. Ce langage est codé. Par exemple, la description de processeur du Z80 contient environ 4,5 Koctet. La génération d'une description de processeur nécessite un assembleur qui peut traiter quelques pseudoinstructions et des et des étiquettes.

On pourrait coder, dans la description de processeur, l'instruction XOR comme suit :

```
MNE3:
        ...
        .ASCII  "XOR"
        .8      16'A8
        .16     AIXOR
        ...
REGABCD:.8      "B",VAL,16'1
        .8      "C",VAL,16'2
        .8      "D",VAL,16'3
REGA:   .8      "A",VAL,16'0
        .8      0
REGIHL: .8      "{","H","L","}",          VAL,16'7
        .8      0
        ...
AIXOR:  .16     REGA+OA,     M_MP_V+V1,    TERM,        GCJ+(AIXOR1-APC)/2
        .16     REGIHL+OA,   M_MP_V+V1,    A_OP_V+V1,   GCJ+(AIXOR2-APC)/2+ILOP
AIXOR1: .16     REGABCD+OA,  A_OP_V+V1,    M_P_L+V1+1,  EOA+ILOP
AIXOR2: .16     REGA+OA,     M_P_L+V1+1,   TERM,        EOA+ILOP
```

Nous retrouvons REGA, REGABCD, etc. L'assembleur commence chez MNE3, c.à.d. il cherche le code opératoire dans cette table car elle contient tous les codes opératoires avec 3 lettres. Il trouve l'instruction XOR.

Un octet et un mot 16 bit suivent le nom XOR dans cette description de processeur. L'octet contient le code opératoire correspondant. L'adresse de la branche (AIXOR) est indiquée dans le mot 16 bit où il faut continuer l'analyse. Ici on trouve le langange qui est interprété par l'assembleur. C'est la table d'interprétation. Chaque ligne contient quatre mots. Chaque mot contient 16 bit.

Le premier mot indique le test à effectuer (REGA). OA signifie analyse d'opérande. Donc, l'assembleur compare la ligne d'entrée, qui contient l'instruction, au registre A. Si le résultat du test est positif, l'assembleur interprète les deux mots prochains. Ils contiennent les instructions de génération de code. TERM signale à l'assembleur qu'il ne doit pas générer de code et qu'il peut interpréter le dernier mot de la ligne. Ce mot indique comment il faut continuer. Si le test était positif, GCJ (Get Comma and Jump, recherche virgule et saute) est exécuté et l'analyse continue à l'adresse AIXOR1. Si le test était négatif, la ligne suivante est interprétée.

ILOP (Illegal Operand, opérande illégal) termine l'analyse, c.à.d. il n'y a pas de possiblilités supplémentaires. Les registres A à D sont testés chez AIXOR1 et M_MP_V+V1 et A_OP_V+V1 génèrent le code correspondant.

La longueur de code est fixé à 1 octet avec M_P_L+V1+1. EOA (End Of Analysis, fin d'analyse) signale à l'assembleur une génération de code à succès et la fin de l'analyse.

3. Le principe

Le principe décrit dans l'exemple ci-dessus fonctionne pour un microprocesseur quelconque. Une multitude de tests et d'instructions de génération de code sont à disposition pour le programmeur. Ces données sont définies dans le fichier PRODEF.ASM. Jusqu'à présent, nous avons décrit seulement l'analyse d'instruction. La génération de code fonctionne comme suit:

Notre but est de générer le code d'instruction complet. Pour cela, la variable PROCBYTE est utilisée qui contient une série d'octets. Le programmeur peut charger des valeurs dans PROCBYTE moyennant plusieurs instructions de génération de code. Ces valeurs sont par exemple la valeur du code opératoire, des opérandes (voir chez REGABCD l'octet après VAL), des constantes, etc. Il faut combiner ces valeurs de telle sorte qu'on obtient le code d'instruction désiré.

La variable PROCBYTE est divisée en huit champs. Chaque champ contient plusieurs octets. Le programmeur peut fixer la longueur d'un champ dans la description de processeur. Les champs sont assemblés à la fin de l'analyse et ainsi le code d'instruction final est construit.

Dans notre exemple, nous avions:

```
M_MP_V+V1
```

Ceci signifie:

```
valeur (code opératoire) + valeur (P, ici 0) -> champ[1,1]
```

Fait la somme du code opératoire et de zéro et charge le résultat dans le premier champ, premier octet. Chaque lettre est une abbréviation (voir aussi PRODEF.ASM):

```
M_    Move
M     Mnémonique
P     Paramètre 8 bit
_V    champ [i,1]
V1    champ 1
```

ainsi que:

```
A_OP_V+V1
```

Ceci signifie:

```
valeur (opérande) + valeur (P, ici 0) + champ[1,1] -> champ[1,1]
```

La somme de l'opérande, de zéro et du premier champ, premier octet, est chargée dans le premier champ, premier octet. En détails:

```
A_    Ajouter
O     Opérande
P     Paramètre 8 bit
_V    champ [i,1]
V1    champ 1
```

Et finalement:

```
M_P_L+V1+1
```

signifie:

```
valeur (P, ici 1) -> longueur de champ[1]
```

La lettre L signifie Length (longueur).

Notre instruction XOR est tellement simple qu'il n'est pas nécessaire d'utiliser plusieurs champs. En effet, nous pouvons assembler le code d'instruction complet dans le champ 1. Les microprocesseurs 16 bit (p.ex.le 8086, le 68000, etc.) exigent plus de programmation (voir l'exemple complexe).

Imprimez s.v.p. le fichier PRODEF.ASM. De même, imprimez les descriptions de processeur Z80*.ASM et NS32000*.ASM qui vous donnent des exemples complets.

4. La structure de la table d'interprétation

La table d'interprétation contient deux choses: l'arbre pour l'analyse du texte et les instructions pour la génération de code. La structure générale d'une ligne de la table d'interprétation a la forme suivante:

```
.16     test,   instruction_de_génération_de_code1, ...    décision
```

Selon la compléxité d'un procésseur, on peut adapter le nombre des instructions pour la génération de code (pour le Z80, il y a deux par ligne).

Si le résultat d'un test est postitif, le pointeur dans la ligne d'entrée est avancé. Il y a quatre types de test:

1) l'appel d'une sousroutine dans la table d'interprétation.
   Des attributs communs peuvent être réunis dans des sousroutines.
   L'adresse de la sousroutine est indiquée directement.

2) l'analyse d'opérande
   compare une liste d'opérandes (avec les valeurs correspondantes) avec le texte de la ligne d'entrée. Si un opérande est trouvé, un caractère spécial doit suivre l'opérande: <espace>, <TAB>, virgule, point-virgule, deux-points, signe plus, signe moins, astérisque ou point. Une valeur 8 bit est assignée à chaque opérande. Le zéro termine la liste des opérande. On indique l'adresse et ajoute OA.

3) la comparaison de texte
   compare le texte de la ligne d'entrée avec une chaîne de caractères quelconques. Contrairement à 2), des caractères spéciaux ne sont pas nécessaires après le texte. On indique l'adresse et ajoute CT.

4) une fonction d'assembleur
   appelle une fonction d'assembleur (p.ex. le calcul d'expression, tester des résultats ou des caractères, etc.). Toutes les fonctions d'assembleur commencent avec MAR (voir la liste dans PRODEF.ASM).

Une instruction de génération de code est composée de trois parties:

a) l'opération
   indique l'opération à effectuer par l'assembleur
   (voir liste dans PRODEF.ASM).

b) le numéro de champ
   indique un champ parmi huit qui est utilisé lors de l'opération. Les huit champs n'ont pas tous la même longueur (fixé par la description de processeur). Remarque: Avec l'instruction de génération de code M_P_I on peut choisir un champ d'une manière fixe indépendamment du numéro de champ indiqué.

c) une valeur 8 bit quelconque qui est utilisée selon l'opération.

Les parties a), b) et c) peuvent être combinées à volonté. Les trois parties sont simplement additionnées.

Une décision contient les indications pour la suite de l'analyse d'instruction. Si le résultat du test était positif, il y a quatre possibilités pour la suite:

Sur test positif

1) la fin de l'analyse (EOA)
   l'analyse et la génération de code ont été terminés à succès.

2) recherche une virgule et saute à l'adresse indiquée (GCJ)
   si l'assembleur trouve une virgule (avec des séparateurs éventuels avant et après la virgule), l'analyse continue à l'adresse indiquée. L'adresse est calculée relativement à l'APC. Seulement des sauts en avant (adresses supérieures) sont possibles. S'il n'y a pas de virgule, l'analyse est terminée avec une indication d'erreur.

3) saute à l'adresse indiquée (JMP)

4) la fin d'une sousroutine (RETOK)
   c'est la fin à succès dans une sousroutine (table d'interprétation) et le saut de retour au test qui a fait l'appel (continue avec la génération de code après le test).

Remarque: La différence adresse moins APC (avec GCJ et JMP) est divisée par deux et permet ainsi une longeur de saut maximale de 8 Koctets.

Si le résultat du test était négatif, il y a trois possibilités:

1) essaie la ligne suivante (-)

2) des opérandes illégaux (ILOP)
   L'analyse d'une instruction n'a pas pu être terminé correctement, c.à.d. il n'y a plus de tests. Une erreur est signalée.

3) la fin d'une sousroutine (RETKO) c'est la fin dans une sousroutine sans succès (table d'interprétation) et le saut de retour au test qui a fait l'appel (continue avec la décision dans cette ligne).

Le mot de décision est donc composé d'une des quatre décisions positives et d'une des trois décisions négatives et éventuellement d'une adresse.

5. Un exemple complexe

Nous avons vu sous 3 comment l'analyse d'instruction fonctionne. Nous voulons maintenant examiner, comment on peut analyser des types d'adressage complexes. Nous prenons comme exemple la famille de processeur NS32000.

Nous analysons l'instruction PUSH. Il est vrai que cette instruction n'existe pas dans la liste d'instruction du NS32000. Mais on peut simuler PUSH par l'instruction suivante:

```
      MOVE  source,{-SP}
```

Les instructions PUSH sont plus courts et plus simples à utiliser. L'instruction PUSH a les possibilités suivantes (voir la liste d'instructions du NS32000):

```
      PUSH        .8       rd
                  .16      rd
                  .32      rd
                           list
```

Les trois premières possibilités correspondent à l'instruction du fabricant:

```
      MOVi        rd,TOS
```

et la quatrième possibilité:

```
      SAVE        [liste de registres]
```

Le programmeur écrit en CALM:

```
      PUSH.32     R0
```

on a deux instructions équivalentes à disposition:

```
      MOVD        R0,TOS
ou    SAVE        [R0]
```

La description de processeur pourrait avoir la forme suivante (extrait):

```
R0R7:
RI:     .8      "R","0",VAL,0,"R","1",VAL,1
        .8      "R","2",VAL,2,"R","3",VAL,3
        .8      "R","4",VAL,4,"R","5",VAL,5
        .8      "R","6",VAL,6,"R","7",VAL,7,0
        ...
MNE4:
        ...
        .ASCII  "PUSH"
        .8      16'14,
        116     AIPUSH
        ...
PMSP:   .ASCIZ  "{-SP}<VAL><16'17>"
PSPP:   .ASCIZ  "{SP+}<VAL><16'17>"
PSP:    .ASCIZ  "{SP}<VAL><16'17>"
IRI:    .ASCII  "{R0}<VAL><16'0>{R1}<VAL><16'1>"
        .ASCII  "{R2}<VAL><16'2>{R3}<VAL><16'3>"
        .ASCII  "{R4}<VAL><16'4>{R5}<VAL><16'5>"
        .ASCIZ  "{R6}<VAL><16'6>{R7}<VAL><16'7>"
SI:     .8      "*","1",VAL,16'1C,"*","2",VAL,16'1D
        .8      "*","4",VAL,16'1E,"*","8",VAL,16'1F,0
IMI:    .ASCIZ  "{FP}<VAL><16'18>{SB}<VAL><16'1A>{SP}<VAL><16'19>"
IIMI:   .ASCIZ  "{{FP}<VAL><16'10>{{SB}<VAL><16'12>{{SP}<VAL><16'11>"
IEXT:   .ASCIZ  "{EXT<VAL><16'16>"
P8:     .8      ".","8",VAL,0,0
P16:    .8      ".","1","6",VAL,1,0
P32:    .8      ".","3","2",VAL,3,0

TA:     .ASCIZ  "A^"
TA4:    .ASCIZ  "A4^"
TA7:    .ASCIZ  "A7^"
```

```
TA14:   .ASCIZ  "A14^"
TA25:   .ASCIZ  "A25^"
TR:     .ASCIZ  "R^"
TR7:    .ASCIZ  "R7^"
TR14:   .ASCIZ  "R14^"
TR25:   .ASCIZ  "R25^"
TU:     .ASCIZ  "U^"
T6:     .ASCIZ  "6^"
T13:    .ASCIZ  "13^"
T24:    .ASCIZ  "24^"
        ...
AIPUSH:
.16     P8+OA,  M_P_V+16'5+V0,T_V_V+UP+1,JMP+(AIPUS1-APC)/2
.16     P16+OA, M_P_V+16'5+V0,T_V_V+UP+1,JMP+(AIPUS2-APC)/2
.16     P32+OA, M_P_V+16'5+V0,T_V_V+UP+1,JMP+(AIPUS3-APC)/2+ILOP
AIPUS1:
.16     MARTGS, M_OP_V+V0+16'14+16'C0,TERM,JMP+(AIPU1A-APC)/2+ILOP
AIPUS2:
.16     MARTGS, M_OP_V+V0+16'14+16'C0,TERM,JMP+(AIPU2A-APC)/2+ILOP
AIPUS3:
.16     MARTGS, M_OP_V+V0+16'14+16'C0,TERM,JMP+(AIPU3A-APC)/2+ILOP
AIPU1A:
.16     RDI8,   M_SV_D+V7+LEFT+11,A_D_V+V0,JMP+(AIG2-APC)/2+ILOP
AIPU2A:
.16     RDI16,  M_SV_D+V7+LEFT+11,A_D_V+V0,JMP+(AIG2-APC)/2+ILOP
AIPU3A:
.16     MARTGL+R0R7,M_D_V+V0+2+LSBFIRST,M_P_V+V0+16'62,JMP+(AIG2-APC)/2
.16     RDI32,  M_SV_D+V7+LEFT+11,A_D_V+V0,JMP+(AIG2-APC)/2+ILOP
        ...
AIG2:
.16     MAROK,  M_P_L+V0+2, TERM,          EOA+ILOP
```

Nous reconnaissons qu'il y a trois altérnatives sous AIPUSH après avoir trouvé le code opératoire PUSH. Les trois longueurs de données .8, .16 et .32 y sont testées. Nous allons voir plus tard que les trois branches ne peuvent pas tout simplement appeler RD. Par contre, il doivent appeler RDI8, RDI16 et RDI32 selon la longueur de données reconnue.

RDI8, RDI16 et RDI32 sont des sousroutines qui analysent toutes les possibilités d'adressage de l'opérande de source et génèrent le code correspondant. Les adressages suivantes sont possibles pour RD:

#valeur
Ri
{SP+}
{SP}+{Ri}*x
{SP}+déplacement
{Ri}*x+adresse_absolue
{Ri}*x+adresse_relative
{Ri}+{Rj}*x+déplacement
{Ri}+{Rj}*x
{Ri}+déplacement
{Mi}+{Ri}*x+déplacement
{Mi}+déplacement
{{Mi}+déplacement1}+{Ri}*x+déplacement2
{{Mi}+déplacement1}+déplacement2
{EXT+déplacement1}+{Ri}*x+déplacement2
{EXT+déplacement1}+déplacement2
adresse_absolue
adresse_relative

Avant de parler des détails, voici quelques informations concernant la famille NS32000. Ces processeurs disposent d'une multitude de types d'adressages très performants. Tout est adressable

relativement. Pour cette raison, l'assembleur génère par défaut l'adressage relatif.

L'ordre des types d'adressage dans liste ci-dessus n'est pas un hasard. L'adressage relatif est la dernière alternative. Ainsi, si les autres types d'adressage n'ont pas été trouvé, l'adresse en question doit être une adresse qui est adressée relativement.

La sousroutine RD a la forme suivante:

```
RDI8:
     .16  MARTGC+"#", M_P_V+V7+16'14,   TERM,          JMP+(RDI8A-APC)/2
     .16  RD,         TERM,             0,             RETKO+RETOK
RDI8A:
     .16  MARGSV+8,   M_E_V+V3+1, M_P_L+V3+1,         ILOP+RETOK
RDI16:
     .16  MARTGC+"#", M_P_V+V7+16'14,   TERM          JMP+(RDI16A-APC)/2
     .16  RD,         TERM,             0,             RETKO+RETOK
RDI16A:
     .16  MARGSV+16,  M_E_V+V3+2+MSBFIRST,            M_P_L+V3+2,      ILOP+RETOK
RDI32:
     .16  MARTGC+"#", M_P_V+V7+16'14,   TERM,          JMP+(RDI32A-APC)/2
     .16  RD,         TERM,             0,             RETKO+RETOK
RDI32A:
     .16  MARGSV+32,  M_E_V+V3+4+MSBFIRST,            M_P_L+V3+4,      ILOP+RETOK
RD:
; out                 V7      source gen mode
;         V1      index mode (if any) and length set
;         V3/V4   disp (if any) and length set
     .16  RI+OA,      M_OP_V+V7,        TERM,          RETOK
     .16  PSPP+OA,    M_OP_V+V7,        TERM,          RETOK
     .16  PSP+OA,     M_OP_V+V7,        TERM,          JMP+(RD1-APC)/2
RDAD:
     .16  IRI+OA,     M_OP_V+V7,        TERM,          JMP+(RD2-APC)/2
     .16  IMI+OA,     M_OP_V+V7,        TERM,          JMP+(RD3-APC)/2
     .16  IIMI+OA,    M_OP_V+V7,        TERM,          JMP+(RD4-APC)/2
     .16  IEXT+OA,    M_OP_V+V7,        TERM,          JMP+(RD4-APC)/2
     .16  ABSAV3,     M_P_V+V7+16'15,   TERM,          RETOK
     .16  RELAV3,     M_P_V+V7+16'1B,   TERM,          RETKO+RETOK
RD1:
     .16  MARTGC+"+", TERM,             0,             JMP+(RD1A-APC)/2
     .16  MARTC+"-",  M_P_V+V7+16'19,   TERM,          JMP+(RD3B-APC)/2+ILOP
RD1A:
     .16  IRI+OA,     M_SV_D+V7+LEFT+3, A_SO_D+0,      JMP+(RD1B-APC)/2
     .16  DISPV3P,    M_P_V+V7+16'19,   TERM,          RETKO+RETOK
RD1B:
     .16  SI+OA,      M_OP_V+V7,        M_D_V+V1+1,    JMP+(SETV11-APC)/2+RETKO
RD2:
     .16  SI+OA,      M_V_V+7+V1,       M_OP_V+V7,     JMP+(RD2A-APC)/2
     .16  MARTGC+"+", TERM,             0,             JMP+(RD2B-APC)/2
     .16  MARTC+"-",  TERM,             0,             JMP+(RD2C-APC)/2
     .16  MAROK,      A_P_V+V7+8,       M_P_L+V3+1,    RETKO+RETOK

RD2A:
     .16  MARTGC+"+", TERM,              0,            JMP+(RD2AA-APC)/2+ILOP
RD2AA:
     .16  ABSAV3,     A_P_V+V1+16'15*8, M_P_L+V1+1,    RETOK
     .16  RELAV3,     A_P_V+V1+16'1B*8, M_P_L+V1+1,    RETKO+RETOK
RD2B:
     .16  IRI+OA,     M_SV_D+V7+LEFT+3, A_SO_D+0,      JMP+(RD2BI-APC)/2
     .16  DISPV3P,    A_P_V+V7+16'8,    TERM,          RETKO+RETOK
RD2BI:
     .16  MAROK,      M_D_V+V1+1,       TERM,          JMP+(RD2BIA-APC)/2+RETKO
RD2BIA:
     .16  SI+OA,      M_OP_V+V7,        M_P_L+V1+1,    JMP+(RD2BII-APC)/2+RETKO
RD2BII:
     .16  TPLMI,      TERM,             0,             JMP+(RD2BIII-APC)/2
```

```
     .16  MAROK,       TERM,             0,              RETKO+RETOK
RD2BIII:
     .16  DISPV3,      A_P_V+V1+16'8*8,  TERM,           RETKO+RETOK
RD2C:
     .16  DISPV3,      A_P_V+V7+8,       TERM,           RETKO+RETOK
RD3:
     .16  MARTGC+"+",  TERM,             0,              JMP+(RD3A-APC)/2
     .16  MARTC+"-",   TERM,             0,              JMP+(RD3B-APC)/2
     .16  MAROK,       M_P_V+V3+0,       M_P_L+V3+1,     RETKO+RETOK
RD3A:
     .16  IRI+OA,      M_SV_D+V7+LEFT+3, A_SO_D+0,       JMP+(RD3AI-APC)/2
     .16  DISPV3P,     TERM,             0,              RETKO+RETOK
RD3B:
     .16  DISPV3,      TERM,             0,              RETKO+RETOK
RD3AI:
     .16  SI+OA,       M_OP_V+V7,        M_D_V+V1+1,     JMP+(RD3AII-APC)/2+RETKO
RD3AII:
     .16  DISPV3,      M_P_L+V1+1,       TERM,           RETKO+RETOK
RD4:
     .16  DISPV3,      0,                TERM,           JMP+(RD4A-APC)/2+RETKO
RD4A:
     .16  MARTGC+"}",  TERM,             0,              JMP+(RD4B-APC)/2+RETKO
RD4B:
     .16  MARTGC+"+",  TERM,             0,               JMP+(RD4C-APC)/2
     .16  DISPV4,      TERM,             0,              RETKO+RETOK
RD4C:
     .16  IRI+OA,      M_SV_D+V7+LEFT+3,                 A_SO_D+0,JMP+(RD4CI-APC)/2
     .16  DISPV4P,     TERM,             0,              RETKO+RETOK
RD4CI:
     .16  SI+OA,       M_OP_V+V7,        M_D_V+V1+1,     JMP+(RD4CII-APC)/2+RETKO
RD4CII:
     .16  DISPV4,      M_P_L+V1+1,       TERM,           RETKO+RETOK
```

Prenons comme exemple les instructions suivantes:

```
PUSH.8  #10
PUSH.16 #1000
PUSH.32 #100000
```

Dans le premier cas, l'assembleur attend une valeur 8 bit. Pour cette raison, la sousroutine RDI8 est appelée qui calcule une expression 8 bit avec l'appel d'assembleur MARGSV+8. De même, l'assembleur attend pour PUSH.16 une valeur 16 bit et pour PUSH.32 une valeur 32 bit. Uniquement pour cette raison, il fallait subdiviser l'instruction PUSH selon la longueur de données. De plus, seulement avec PUSH.32 il y a la possibilité d'une liste de registre.

Si l'assembleur ne trouve pas après le dièze une valeur valide, l'analyse est tout simplement arrêtée (ILOP).

Considérons maintenant les adressages:

{Ri}*x+adresse_absolue	(a)
{Ri}*x+adresse_relative	(b)
{Ri}+{Rj}*x+déplacement	(c)
{Ri}+{Rj}*x	(d)
{Ri}+déplacement	(e)

qui sont possibles pour toutes les trois longueurs de données.

Après l'étiquette RD, la ligne d'entrée est comparée successivement à RI, PSPP, PSP et IRI. Dans notre cas, IRI est exact et l'assembleur suit cette ligne. Une instruction de génération de code se trouve dans cette ligne. Il charge la valeur du registre IRI dans le champ 7. Le mot de décision nous amène à l'étiquette RD2. Quatre alternatives sont possibles. On teste avec SI s'il s'agit du cas

{Ri}*x

Si oui, deux instructions de génération de code sont exécutées et ensuite on saute à RD2A. Ici, il y a qu'une possiblité: il faut qu'un signe plus doit être présent. Si qui, nous arrivons à RD2AA. Nous avons donc

{Ri}*x+

Ceci correspond aux cas (a) et (b). Deux tests doivent maintenant suivre qui attendent une adresse qui est adressée soit d'une manière absolue ou relative. Effectivement: les sousroutines ABSAV3 et RELAV3 sont appelées l'une après l'autre.

Considérons d'abord l'adressage absolu:

```
ABSAV3:
    .16  MAROK,         M_P_I+3,             TERM,           JMP+(ABSAD-APC)/2+ILOP
ABSAV5:
ABSAD:
;   out  V5 (Vi if deviated by M_P_I)
;   default is relative addressing -> prefix is obligatory
    .16  T6+CT,         TERM,                0,              JMP+(ABSA2-APC)/2
    .16  T13+CT,        TERM,                0,              JMP+(ABSA3-APC)/2
    .16  T24+CT,        TERM,                0,              JMP+(ABSA4-APC)/2
    .16  TU+CT,         TERM,                0,              JMP+(ABSA5-APC)/2+RETKO
ABSA5:
    .16  MARGPV+TFR+6,  M_E_V+V5+1,          N_P_V+V5+16'7F, JMP+(ABSA6-APC)/2
    .16  MARGPV+TFR+13, M_E_V+V5+2+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA7-APC)/2
    .16  MARGPV+24,     M_E_V+V5+4+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA8-APC)/2+RETKO
ABSA2:
    .16  MARGPV+6,      M_E_V+V5+1,          N_P_V+V5+16'7F, JMP+(ABSA6-APC)/2+RETKO
ABSA3:
    .16  MARGPV+13,     M_E_V+V5+2+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA7-APC)/2+RETKO
ABSA4:
    .16  MARGPV+24,     M_E_V+V5+4+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA8-APC)/2+RETKO
ABSA6:
    .16  MAROK,         M_P_L+V5+1,          TERM,           JMP+(CLRM_P_I-APC)/2+RETKO
ABSA7:
    .16  MAROK,         O_P_V+V5+16'80,      M_P_L+V5+2,     JMP+(CLRM_P_I-APC)/2+RETKO
ABSA8:
    .16  MAROK,         O_P_V+V5+16'C0,      M_P_L+V5+4,J    MP+(CLRM_P_I-APC)/2+RETKO
```

Il y a une petite particularité dans ABSV3: L'instruction de génération de code M_P_I+3 dirige toutes les instructions suivantes dans le champ 3. C'est à dire que toutes les opérations dans ABSAD ne se font pas comme indiqué dans le champ 5 (V5), mais dans le champ 3. Un saut à CLRM_P_I à la fin d'ABSAD annule avec M_P_I+8 la déviation.

Nous nous rappelons que l'adressage relatif est normalement choisi dans le cas de la famille NS32000. Il est donc nécessaire de spécifier un indicateur d'adresse pour générer l'adressage absolu. Il y a quatre possibilités:

6^
13^
24^
U^

Ces quatre possibilités sont comparées successivement avec la ligne d'entrée. Il s'agit d'une comparaison de texte (CT = comparaison de texte). Donc, un caractère quelconque peut suivre après l'indicateur d'adresse. Par contre, un caractère spécial doit suivre lors d'une comparaison d'opérande (OA = operand analysis, analyse d'opérande).

Si l'assembleur ne trouve pas des indicateurs d'adresse absolus, nous retournons avec RETKO à la première ligne après RD2AA. Ensuite, la deuxième ligne est essayée qui nous amène à la sousroutine RELAV3:

```
RELAV3:
    .16   MAROK,         M_P_I+3,             TERM,           JMP+(RELAD-APC)/2+ILOP
RELAV5:
RELAD:
; out    V5 (Vi if deviated by M_P_I)
    .16   TR7+CT,        TERM,                0,              JMP+(RELA2-APC)/2
    .16   TR14+CT,       TERM,                0,              JMP+(RELA3-APC)/2
    .16   TR25+CT,       TERM,                0,              JMP+(RELA4-APC)/2
    .16   TR+CT,         TERM,                0,              JMP+(RELA5-APC)/2
RELA5:
    .16   MARGRV+TFR+6,  M_E_V+V5+1,          N_P_V+V5+16'7F, JMP+(ABSA6-APC)/2
    .16   MARGRV+TFR+13, M_E_V+V5+2+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA7-APC)/2
    .16   MARGRV+24,     M_E_V+V5+4+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA8-APC)/2+RETKO
RELA2:
    .16   MARGRV+6,      M_E_V+V5+1,          N_P_V+V5+16'7F, JMP+(ABSA6-APC)/2+RETKO
RELA3:
    .16   MARGRV+13,     M_E_V+V5+2+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA7-APC)/2+RETKO
RELA4:
    .16   MARGRV+24,     M_E_V+V5+4+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA8-APC)/2+RETKO
```

Ici, nous avons sept possibilités:

R7^
R14^
R25^
R^
adressage relatif possible R7 ?
adressage relatif possible R14 ?
adressage relatif R25

Comme mentionné, des indicateurs d'adresse explicites ne sont pas nécessaires. Les possibilités cinq à sept testent maintenant successivement s'il y a un adressage relatif dans les gammes -64..+63 (R7), -8192..+8191 (R14) ou -16777215..+16777215 (R25). R7 et R14 ne sont que des possibilités. Une référence en avant nous donne toujours un code R25 car la valeur n'était pas encore connue à la première passe.

Si un adressage relatif est possible, nous retournons par RETOK à RD2AA et par le RETOK de RD2AA à RDI8, RDI16 ou RDI32.

Si aucun adressage relatif n'est possible, nous retournons par le même chemin, mais cette fois-ci avec RETKO, c.à.d. avec une erreur. Un ILOP signale ici qu'il n'y a pas d'autres alternatives et un message d'erreur est affiché.

Ainsi nous avons traité les cas (a) et (b). Retournons à l'étiquette RD2. Le signe plus est testé dans la deuxième ligne:

{Ri}+

C'est le début pour les cas (c), (d) et (e). Mais attention! Avant de continuer avec RD2B, nous constatons avec stupéfaction qu'il y a deux autres possibilités. Le signe moins est testé dans la troisième ligne! Ceci signifie:

{Ri}-

et c'est une possibilité supplémentaire du cas (e). Car nous pouvons écrire:

{Ri}-10

ce qui est plus naturel que

{Ri}+(-10)

De plus, la sousroutine DISPV3P est appelée par RD3C. Nous en parlerons plus tard. La quatrième ligne ne teste rien du tout. C'est la fin de l'analyse. Ceci correspond à

{Ri}

et c'est aussi moins crispé que d'écrire

{Ri}+0

C'est de nouveau le cas (e)! Ainsi, un comfort additionel est offert à peu de frais. Si nous vérifions les types d'adressage, nous constatons qu'il y a partout les possibilités suivantes avec déplacement:

+déplacement
-déplacement
rien

Continuons sous RD2B. Nous obtenons avec IRI

{Ri}+{Rj}

Par RD2BI nous arrivons à RD2BIA. Seulement l'opérande SI est possible et nous amène à RD2BII (cas c).

{Ri}+{Rj}*x

Deux possibilités existent sous RD2BII. L'assembleur teste avec TPLMI, de nouveau une sousroutine, s'il y a un signe plus ou un signe moins.

```
TPLMI:
     .16  MARTC+"+",  TERM,            0,              RETOK
     .16  MARTC+"-",  TERM,            0,              RETKO+RETOK
```

Si oui, nous appelons sous RD2BIII la sousroutine DISPV3 (cas (c)). Sinon, il s'agit du cas (d) et l'analyse est terminée.

```
DISPV3P:
     .16  MAROK,         M_P_I+3,            TERM,           JMP+(DISP1-APC)/2+RETKO
DISPV4P:
     .16  MAROK,         M_P_I+4,            TERM,           JMP+(DISP1-APC)/2+RETKO
DISPV6P:
     .16  MAROK,         M_P_I+6,            TERM,           JMP+(DISP1-APC)/2+RETKO
DISPV3:
     .16  MAROK,         M_P_I+3,            TERM,           JMP+(DISP-APC)/2+RETKO
DISPV4:
     .16  MAROK,         M_P_I+4,            TERM,           JMP+(DISP-APC)/2+RETKO
DISPV6:
     .16  MAROK,         M_P_I+6,            TERM,           JMP+(DISP-APC)/2+RETKO
DISPV5:

DISP:
; out   V5 (Vi if deviated by M_P_I)
     .16  MARTGC+"+",    TERM,               0,              JMP+(DISP1-APC)/2
     .16  MARTC+"-",     TERM,               0,              JMP+(DISP5-APC)/2
     .16  MAROK,         M_P_V+V5+0,         M_P_L+V5+1,     RETKO+RETOK
DISPV5P:
DISP1:
     .16  TA7+CT,        TERM,               0,              JMP+(DISP2-APC)/2
     .16  TA14+CT,       TERM,               0,              JMP+(DISP3-APC)/2
     .16  TA25+CT,       TERM,               0,              JMP+(DISP4-APC)/2
     .16  TA+CT,         TERM,               0,              JMP+(DISP5-APC)/2
DISP5:
```

```
    .16  MARGSV+TFR+6,  M_E_V+V5+1,          N_P_V+V5+16'7F, JMP+(ABSA6-APC)/2
    .16  MARGSV+TFR+13, M_E_V+V5+2+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA7-APC)/2
    .16   MARGSV+24,    M_E_V+V5+4+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA8-APC)/2+RETKO
DISP2:
    .16  MARGSV+6,      M_E_V+V5+1,          N_P_V+V5+16'7F, JMP+(ABSA6-APC)/2+RETKO
DISP3:
    .16  MARGSV+13,     M_E_V+V5+2+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA7-APC)/2+RETKO
DISP4:
    .16  MARGSV+24,     M_E_V+V5+4+MSBFIRST, N_P_V+V5+16'3F, JMP+(ABSA8-APC)/2+RETKO
```

DISPV3 ressemble aux sousroutines présentées ABSAD et RELAD. Comme décrit ci-dessus, aussi les cas suivants sont traités:

+déplacement
-déplacement
rien

En outre, DISPV3P tient compte du fait que le signe a déjà été testé.

Nous avons vu comment les cas (a) à (e) sont analysés et nous avons découverts quelques détails. Pourtant les cas décrits ici sont relativement simples. Par exemple, pour faire la liste des possibilités d'adressages il faut passablement de travail préliminaire. Si nous comparons cette liste avec la liste d'instruction de la famille NS32000, nous constatons deux points:

L'ordre des types d'adressage est différent. Dans la liste d'instruction, les types d'adressage sont ordonnés selon l'adressage immédiat, direct, indirect, et relatif et selon la complexité de l'adressage. De plus, il semble qu'il y a moins de types d'adressage dans la liste d'instruction.

Ces différences ont les raisons suivantes: Une liste d'instruction a comme objectif de rassembler un maximum de types d'adressage. Il en résulte une meilleure clarté. En outre, le programmeur s'intéresse uniquement pour la question suivante: "Quelles sont les possibilités d'adressages avec le registre XY?" Nous pouvons expliquer avec l'exemple suivant pourquoi notre liste des possibilités d'adressage est plus longue que celle de la liste d'instruction. Dans la liste d'instruction nous avons:

{Ai}+déplacement

Dans notre liste, ceci correspond à deux types d'adressage:

{Ri}+déplacement
{Mi}+déplacement

Il faut distinguer ces deux cas car la suite après {Ri} ou {Mi} est différente:

{Ri}*x+adresse_absolue	{Mi}+{Ri}*x+déplacement
{Ri}*x+adresse_relative	{Mi}+déplacement
{Ri}+{Rj}*x+déplacement	
{Ri}+{Rj}*x	
{Ri}+déplacement	

Nous avons simplifié la situation avec {Mi}. Car il faut traiter séparément l'opérande {SP} sous le cas {Mi}. Effectivement, il s'agit d'un autre adressage (le fabricant National Semiconductor appelle ce cas TOS) et offre d'autres posssibilités.

Pour le moment, nous n'avons pas traité la génération de code. Nous invitons le lecteur à vérifier les instructions de génération de code.

Grâce aux champs ajustables, il était possible de coder ce processeur. Tous les huit champs ont été utilisé. Une fonction spécifique a été assigné à chaque champ:

champ 0	instruction de base, longeur 3 octet
champ 1	octet d'index source, 1 octet
champ 2	octet d'index destination, 1 octet
champ 3	déplacement 1, source, 4 octet
champ 4	déplacement 2, source, 4 octet
champ 5	déplacement 1, destination, 4 octet
champ 6	déplacement 2, destination, 4 octet
hamp 7	valeur directe, 4 octet

Il ressort de la disposition des champs que les données sont bien imbriquées. Les informations de source et de destination se relayent plusieurs fois. Ce chaos est toutefois bien utile: Par exemple, RDI8 remplit les champs 1, 3 et 4. La longueur des champs 1, 3 et 4 est fixé dans RDI8, respectivement RD. AIPUSH ne "sait" rien de tout cela. AIPUSH s'occupe uniquement de l'instruction de base qui contient le code opératoire et les codes de source et de destination. L'assembleur assemble l'instruction après EOA, met les champs à la file. Grâce à cette astuce et à la structure de la famille NS32000, chaque sousroutine peut indépendamment des autres faire sa partie de l'analyse et de la génération de code.

Dans bien des cas une instruction est encore plus complexe, c.à.d. Dispose de possibilités d'adressage plus nombreuses comparé à notre exemple. Par exemple, nous pouvons étendre l'instruction PUSH comme suit:

PUSH	|.8 |	#A4^valeur	(I)
	|.16|	Mi.32	
	|.32|	MOD.16	
		IB.32	
		F.8	
		SF.16	
PUSH	.32	#ad	(II)
		#(ad)*8+{Ri}	

Dans les extensions (I) et (II), nous avons simplement repris les instructions MOVE avec l'opérande de destination {-SP}. Les complications sont maintenant plus visibles. L'exemple

```
PUSH.16 #A4^valeur
```

est une concurrence pour

```
PUSH.16 #16^valeur
```

Les deux instructions ont le même effet. La différence est la suivante: La valeur dans le premier cas doit être dans la gamme -8..+7 et le code de machine final est plus compact. Pour cette raison, il n'est plus possible d'appeler simplement RDI16. De plus, la première instruction a besoin d'un autre code opératoire de machine et possède un autre format de machine. La séparation nette entre l'analyse d'opération et l'analyse d'opérande n'est plus possible.

L'instruction

```
PUSH.32 #ad
```

nous mène à un dilemme encore plus grave. AD possède les possibilités d'adressage identiques à RD (sauf pour l'adressage immédiat et {SP+}, voir aussi la liste d'instruction du NS32000). Nous nous rappelons que l'adressage relatif est généré sans indicateurs spécifiques. Mais, comment distingue-t-on l'adressage relatif d'une valeur?

```
PUSH.32 #adresse
PUSH.32 #valeur
PUSH.32 #adresse+valeur
```

Dans ce cas, il faut utiliser la classification des symboles. Normalement, l'adressage relatif n'est possible que pour des étiquettes qui ont été définies dans le même module de programme. L'assembleur peut donc distinguer à partir du type d'opérande quelle instruction il faut générer. Par contre, l'analyse des expressions deviennent nettement plus difficile si l'on ne peut pas classer correctement une expression comme adresse ou valeur.

Cet exemple nous montre que l'analyse d'une instruction peut devenir très complexe et ainsi volumineux. Dans bien des cas il y a un degré de difficulté de plus: La plupart des instructions possède un opérande de source et un opérande de destination. Notre exemple avec l'instruction PUSH se limitait à un opérande de source.

Finalement, nous voulons traiter un problème qui n'a jamais été mentionné: Dans notre liste des types d'adressage nous avons deux types d'adressage relatifs:

```
{Ri}*x+adresse_relative
adresse_relative
```

Question: La notation suivante, serait-elle possible?

```
adresse_relative+{Ri}*x
```

Cette notation est plus naturelle car elle correspond plus précisement à la manière comment l'adresse est composée. Le seul désavantage de cette notation est la difficulté d'analyser correctement cette expression. L'assembleur attend pour adresse_relative une expression et bute contre le signe plus car {Ri}*x n'est pas un terme correct. Pour cette raison, il est préférable de noter tous les types d'adressage de telle manière qu'on peut toujours évaluer une expression correctement. Par ailleurs, l'assembleur CALM analyse toujours correctement ces expressions, c.à.d. pour des expressions comme

```
adresse_relative+{Ri}*x
adresse_relative+A16^{Ri}*x
adresse_relative+32^{Ri}*x
```

l'assembleur CALM calcule seulement adresse_relative. Le pointeur de texte pointe après sur le signe plus.

6. Remarques

La description de processeur présentée ici est utilisable pour presque tous les microprocesseurs disponibles. Le format de machine de la description de processeur est très compacte et nécessite peu de place mémoire et de temps de chargement.

De plus, la description de processeur est complètement indépendante de l'assembleur utilisé (l'interprétation doit simplement être correcte).

7. Générer un module CALM

Si quelqu'un veut générer un module CALM, le procédé suivant est proposé

(xxx = processeur quelconque, p.ex. Z80):

1) Etablir la liste d'instruction CALM (CALM Reference Card): documentation de toutes les instructions dans la notation CALM, xxxA.TXT.

2) Liste de comparaison: notation du fabricant -> notation CALM (Instruction Comparison, manufacturer notation versus CALM notation): Cxxx.TXT.

3) Fichier de test: liste de toutes les instructions y compris le code machine à générer (comme commentaire): Txxx.ASM. Ce fichier est assemblé plus tard avec ASCALM Txxx/L et est testé avec TESTLIST Txxx.

4) Faire le module: xxx.ASM. Le fichier est assemblé avec ASPRO xxx qui génère le fichier xxx.PRO. Ensuite faire des tests avec Txxx.ASM. Voir les prototypes MODUL8.ASM et MODUL16.ASM.

5) Fichier de test ordonné: STxxx.ASM. Il faut trier toutes les instructions de Txxx.ASM avec la forme finale notation CALM - code machine. Utilisez le programme SORT_TXT.

6) Lliste de code machine: Ixxx.TXT. Contient toutes les instructions de STxxx.ASM ordonné alphabétiquement et dans la forme code machine - notation CALM. Utilisez les programmes INSTRUCT et SORT_TXT.

7) Traduire un petit programme de la notation du fabricant en CALM et comparer les codes machines générés: Exxx.ASM.

8) Faire la description pour le nouveau module: Bxxx.TXT, c.à.d. quoi offre le module, comment faut-il utiliser .PROCSET, etc.

9) Faire le traducteur: xxx_CALM.PAS: traduit un programme de la notation du fabricant en notation CALM. Selon les cas, il faut faire aussi l'inverse: CALM_xxx.PAS: traduit un programme de la notation CALM en notation du fabricant.

8. Les listes d'instruction CALM

Toutes les listes d'instruction CALM sont composées de la même manière. Ceci donne toujours la même apparence. De plus, on peut directement comparer. Les fichiers I8080A.PRN, NS32000A.PRN et Z80A.PRN sont fournis comme exemples (sans les commandes de formatage).

Les listes d'instruction sont rédigées en anglais. Les commentaires (si nécessaire) utilisent des mots simples. Une liste d'instruction CALM est subdivisée comme suit (les soustitres du processeur 8080 nous servent comme exemple):

8080/8085
CALM REFERENCE CARD

Indique le processeur pour lequel la liste d'instruction est valide.

8080/8085 Description

Une description générale du processeur suit. S'il y a plusieurs microprocesseurs compatibles, les différences sont indiquées.

Programming Modell

Un modèle du processeur pour le programmeur est présenté. Ce modèle contient les registre du processeur considéré (noms, longueur, etc.) et les contenus des registres d'indicateur et d'état.

General

Les longueurs d'adresse et de données sont indiquées.

Abbreviations used

Les abbréviations utilisées sont indiquées.

Modifications versus CALM Standard

Une liste des différences par rapport à la norme CALM est donnée. Ceci concerne les bits d'indicateur qui ont été définis autrement par rapport au fabricant. De même, on indique si les indicateurs de données sont obligatoires et sinon, comment on détermine la longueur de données (p.ex. d'après le nom de registre).

Remark

Contient des remarques générales; par exemple, la numérotation des octets, une liste des codes d'indicateur équivalents additionels, les priorités d'interruption et une description des séquences de mise à zéro (reset) et d'interruption. De même, on indique si et quels registres et bits d'indicateur ont été renommé.

Une liste des instructions suit maintenant. L'ordre correspond largement à l'ordre de définition en CALM.

Transfer instructions

contient toutes les instructions de transfert

Arithmetic instructions

contient toutes les instructions arithmériques

Logical instructions

contient toutes les instructions logiques

Shift instructions

contient toutes les instructions de décalage

Test instructions

contient toutes les instructions de test

Program flow instructions

contient toutes les instructions de déroulement de programme

Special instructions

contient toutes les instructions spéciales qu'on ne peut pas classer sous les catégoires ci-dessus. Les nouvelles instructions qui sont composées de plusieurs mots CALM (p.ex. ADDJUMP) ne se trouvent pas ici mais sont classées sous une des catégories ci-dessus. Voici quelques instructions spéciales: DAA (8080), LINK (68000), ENTER (iAPX86).

.....

d'autres instructions selon la complexité, p.ex. des instructions de bloc.

Notes

contient toutes les remarques concernant les instructions précédantes et la documentation du fabricant utilisée.

La structure d'une instruction est comme suit (exemple):

```
EX          DE,HL     []
            {SP},HL
(XCHG XTHL)
```

EX est le code opératoire CALM. DE,HL et {SP},HL sont les opérandes. [] signifie que les bits d'indicateur ne sont pas modifiés. XCHG est la notation du fabricant pour EX DE,HL et XTHL pour EX {SP},HL. Autre exemple:

```
INC |       r16       []
DEC |
(INX DCX)
```

signifie que INC r16 et DEC r16 sont possible et que dans les deux cas les bits d'indicateur ne sont pas modifiés. r16 remplace BC, DE, HL et SP. Donc, huit instructions sont présentées ici. Encore un exemple:

```
MOVE        #VAL8 |,s8 []
            s8    |
            {HL}  |
(MVI MOV)
```

est identique à:

```
MOVE        #VAL8,s8  []
MOVE        s8,s8     []
MOVE        {HL},s8   []
```

VAL8 est une valeur 8 bit (8^ est optionnel avant VAL8). s8 remplace B, C, D, E, H ou L. Les bits d'indicateur ne sont modifiés. Finalement:

```
DAA         A         [N,Z,H,P,C]
(DAA)                 Decimal Adjust A, only valid after ADD and ADDC
```

Les bits d'indicateur N, Z, H, P et C sont modifiés.