RÉSUMÉ DU LANGAGE D'ASSEMBLAGE POUR LA MACHINE RISC

Ce langage d'assemblage[1] a été défini et réalisé par Karol PROCH

`instruction ::= [étiquette] mnémo_opération opérande {, opérande} [// commentaire] ↵`

SYNTAXE DES OPÉRANDES DELON LE MODE D'ADRESSAGE

Le mode d'adressage est la manière dont on détermine la place de l'opérande.

MODE D'ADRESSAGE	SYNTAXE	EXEMPLE	SÉMANTIQUE DE L'OPÉRANDE
Immédiat	`#constante`	`#3`	*constante dans le mot d'extension*
Direct	`@adresse`	`@0xFF02`	*contenu case dont l'"adresse" est dans mot d'extension*
Registre	`registre`	`R2`	*contenu du registre*
Basé	`(registre)`	`(R2)`	*contenu case pointée par registre*
Basé post-incrémenté	`(registre)+`	`(R2)+`	*contenu case pointée par registre;* *registre incrémenté ensuite*
Basé pré-incrémenté	`-(registre)`	`-(R2)`	*registre incrémenté d'abord;* *contenu case pointée par registre*
Indexé	`(registre)déplacement`	`(R2)3`	*contenu case d'adresse registre + déplacement*
Indirect pré-indexé	`*(registre)déplacement`	`*(R2)3`	*contenu case pointée par* *case d'adresse registre + déplacement*
Rapide ("Quick")	`constante`	`3`	*constante dans l'octet droit de l'instruction*

SYNTAXE DES GROUPES D'INSTRUCTIONS *(cf. carte de programmation)*

GROUPE	SYNTAXE	EXEMPLES
1	mnémo_op3 Rsa, Rsb, Rd	**`ADD R2, R4, R1`** *// somme des contenus de R2 et R4 → R1* **`SUB R0, R5, R0`** *// contenu R0 - contenu R5 → R0*
2	mnémo_op2 Rs, Rd	**`NEG R2, R1`** *// opposé du contenu de R2 → R1* **`CMP R3, R0`** *// compare (contenu R3 - contenu R0) avec 0*
	mnémo_op2 #constant, Rs, Rd	**`ADI #8, R4, R5`** *// 8 + contenu de R4 → R5* **`ANI #0xFF56, R5, R2`** *// et bit à bit de FF56h et contenu R5 → R2*
3	mnémo_Dmnémo_Type Ra, B mnémo_D = **LD** ou **ST** mnémo_Type = **B** *pour octet* **W** *pour mot*	**`LDW R2, R3`** *// charge R2 avec le contenu de R3* **`LDW R2, (R3)`** *// charge R2 avec le mot M[R3]* **`LDB R2, (R3)+`** *// charge R2 avec l'octet M[R3] puis incrémente R3* **`LDW R2, #56`** *// charge R2 avec 56* **`LDW R2, @0xFFEC`** *// charge R3 avec le mot M[FFECh]* **`STW R2, -(R3)`** *// déc. R3 puis sauve le contenu R2 dans `M[R3]`*
4	**J**mnémo_CC déplacement	**`JMP #-128`** *// saute (128-2)/2 mots plus haut* **`JEQ #34`** *// résultat précédent=0 ⇒ saute (34+2)/2 mots plus bas* **`JNE R3`** *// résultat précédent≠0 ⇒ saute (R3+2)/2 mots*
5	mnémo_op1 A	**`JEA @0xF3E2`** *// saute à l'instruction d'adresse F3E2h* **`TRP #5`** *// lance le programme de service de n° d'exception 5*
6	mnémo_op0	**`RTI`** *// retourne du programme de service d'exception* **`RTS`** *// retourne du sous-programme* **`NOP`** *// pas d'opération*
7	mnémo_opq valeur, R	**`LDQ 5, R2`** *// charge 5 dans R2* **`ADQ -3, R4`** *// ajoute -3 à R4*
8	**B**mnémo_CC déplacement	**`BGT 32`** *// résultat précédent >0 ⇒ saute (32+2)/2 mots plus bas* **`BMP -40`** *// saute (40-2)/2 mots plus haut*

Notes:

- La **syntaxe est inspirée de C++** ou Java.

- Il y a **bijectivité entre code machine et forme syntaxique**. Chaque forme syntaxique (avec #, @, * ...) représente **un et un seul** mode d'adressage que la machine ***peut*** effectuer. Il n'y a pas de valeur fonctionnelle et l'on ne peut donc pas écrire ((R1)) pour exprimer M[M[R1]] par exemple. De même, on ne peut pas permuter registre et déplacement dans le mode indexé.

ÉTIQUETTE & COMMENTAIRE

- Toute instruction peut être précédée d'une **étiquette**, symbole qui représente alors l'adresse de l'instruction.

- Tous les caractères entre // et la fin de ligne sont considérés comme un **commentaire**.

`toto` `ADD R1, R2, R3 //` *toto est un **symbole** qui représente désormais l'adresse de cette instruction*

DIRECTIVES D'ASSEMBLAGE

Elles ne sont **pas des instructions** exécutées par le CPU au moment de l'exécution, mais des **directives** à l'assembleur (c'est à dire le programme **de traduction)** pour la traduction en codes machine puis leur assemblage dans la mémoire.

NOM	EXPLICATION	EXEMPLES
equ	Remplace le symbole de gauche par l'expression de droite partout à la suite.	`SP equ R15` *// SP sera remplacé par R15* `TOTA equ 0xFF34` *// TOTA = FF34h*
rsw	**Réserve une zone de mots** en mémoire et affecte l'adresse de début de cette zone au symbole d'étiquette.	`WORDA rsw 234` *// réserve 234 mots*
rsb	**Réserve une zone d'octets** en mémoire et affecte l'adresse de début de cette zone au symbole d'étiquette.	`OCTA rsb 538` *// réserve 538 octets* `table2_adresse rsb 82`
string	**Réserve une zone pour une chaîne de caractères** ASCII terminée par NUL, et affecte l'adresse de début de cette zone au symbole d'étiquette.	`DROITA string "libres et egaux"`
org	Spécifie l'adresse de la première case mémoire assemblée (initialise le compteur de cases d'assemblage) et donc implicitement l'**adresse de chargement**.	`org 0xFF80` *// charge le programme en FF80h* `org PROGA` *// charge le programme en PROGA*
start	Indique l'**adresse de démarrage** du programme assemblé (avec laquelle le PC sera chargé lors du lancement).	`start 0xFF88` *// démarre le prog en FF88h* `start STARTA` *// démarre le prog en STARTA*

EXPRESSIONS

• Les **expressions** ne portent ***que*** sur des **constantes** entières, en utilisant les opérateurs habituels de C, C++ ou Java. L'assembleur (i.e. le programme qui assemble les mots de code machine) peut donc calculer ***à l'avance*** ces expressions. Par exemple, si l'on avait écrit `toto equ 4`, alors `(5 * toto - 3) / 2 + 1` serait remplacé par 9. Mais si l'on écrit: `SP EQU R15`, SP est remplacé par R15, et SP+1=R15+1 n'est pas constant, donc pas calculable à l'avance ! En revanche, `LDW R3, (SP)+` sera remplacé par `LDW R3, (R15)+` qui est valide.

• `0x`89FE signifie que la valeur 89FE est codée en **hexadécimal**

• **$** représente **le compteur de case d'assemblage.** Il est **initialisé à la valeur indiquée par org** (qui est aussi l'adresse de chargement du premier mot assemblé) et s'incrémente à chaque mot assemblé: il signifie donc normalement **l'adresse de l'instruction où il apparaît**.

EXEMPLES D'INSTRUCTIONS

```
loop  ADD R1, R2, R3      // R1 + R2→ R3; le symbole loop  en étiquette prend l'adresse de cette instruction ADD
      ADQ -3, R1          // ajoute -3 à R1 : R1- 3→R1
      ADQ titi*5-1, R1    // ajoute l'expression constante calculée par l'assembleur (titi * 5 -1) à R1
      SUB R1, R2, R3      // R1-R2→R3
      CMP R1, R2          // calcule R1-R2 et change les indicateurs ZF, CF, VF, NF de SR mais ne change pas R1 ni R2
      BMP -56             // branche inconditionnellement  avec un déplacement de -56, donc à l'adresse PC - 56 =
                          // adresse de l'instruction BMP + 2 - 56 = $ + 2 -56 = $ -54 (soit 54 octets = 27 mots plus haut)
      BMP loop-$-2        // branche inconditionnellement avec un déplacement de loop - $ + 2
                          // donc à l'instruction d'adresse PC + déplacement = PC + (loop - $ - 2) ;
                          // or PC pointe sur le mot qui suit, dont l'adresse est
                          // celle de l'instruction courante + 2, soit $ + 2; donc PC = $ + 2 .
                          // En bref, cette instruction branche donc à l'adresse relative $ + 2 + loop - $ - 2 = loop.
                          // Le déplacement est rapide ("quick"): il est dans le mot d'instruction .
                          // L'expression loop - $ - 2 représente une constante qui est  calculée à l'avance par
                          //  l'assembleur  pour produire le code machine et pas par le CPU à l'exécution.
      BEQ toto-$-2        // branche à l'adresse relative toto si l'indicateur ZF=1, donc si le résultat précédent = 0
      JLE #toto-$-2       // saute à l'instruction d'adresse relative toto si le résultat précédent = 0
                          // le déplacement toto-$-2 est immédiat: il est dans le mot qui suit le mot d'instruction.
      JEA @toto           // saute inconditionnellement à l'instruction d'adresse absolue toto
      JSR (R2)            // lance le sous-programme d'adresse absolue  contenue dans R2
```

[1] Le jeu d'instructions, son codage et implémentation matérielle ont été définis par Alexandre PARODI.