```
Le langage d'assemblage

I) Structure d'un programme, directives, étiquettes

  Le langage d'assemblage décrit une image mémoire stockée (sur disque) dans
un fichier exécutable
=> sera le contenu de la mémoire principale au début d'éxécution.

  En général, on met les instructions et constantes dans une zone (text)
en lecture seule et les variables dans une zone (data). Hypothèses de travail
sur adresses de text et data : à chaque exécution le chargeur/lanceur choisit
l'adresse de chaque section (ou dernière étape de compilation avec mémoire virtu
elle).

  Directives (ne génèrent pas d'instruction)

        .text
        .data
        .bss  idem mais initialisée à 0 (taille sans contenu dans fichier)
              historiquement directive "Block Started by Symbol"

   --> ce qui suit sera regroupé dans la section en question

  On donne un nom symbolique aux adresses : étiquette.

  On peut mettre plusieurs étiquettes au même endroit : synonymes.

A gauche, suivie de :           ==> definition de l'adresse associée
A droite, sans deux points      ==> utilisation : à remplacer par adr associée

  Dans la zone text, typiquement :

{Etiquette:}    instruction                     ! correspond à 1 mot 32 bits

  Dans zone data    :

{Etiquette:}    .word   v                       ! 1 mot 32bits initialisé à v

Exemple avec 2 executions : text/data  0x10000/0x40000  et 0x20000/0x50000

                                .text

0x10000 | 0xe0853007    etc_debut: add  r3,r5,r7        0x20000 | 0xe0853007
0x10004 | 0xe2853009    etc_suite: add  r3,r5,#9        0x20004 | 0xe2853009

                                .data

0x40000 | 0x00040004    etc_pt_x: .word etc_x           0x50000 | 0x00050004
0x40004 | 0x01234567    etc_x:    .word 0x01234567      0x50004 | 0x01234567
0x40008 | 0x00010004    etc_pt_s: .word etc_suite       0x50008 | 0x00020004

                                  add r3,r5,r7 est équivalent à .word 0xe0853007

Etiquettes :
  * plus lisible :  .word suite  versus .word 0x10004
  * source du programme indépendant des adresses de chargement

III) Jeu d'instructions RISC et CISC


* Reduced Instruction Set Computer / moderne / SPARC ou ARM :
       1 instruction = 1 mot (code-op) ~= 1 cycle
```



```
       load/store + calcul uniquement sur registres

* Complexe Instruction Set Computer / ancien / 80x86 680x0 :
       1 instructions = 1 ou plusieurs mots (code-op + specif operandes)/cycles
       calcul directement sur la memoire

  1 intructions CISC ~ 1 sequence d'instructions RISC


IV) Instructions arithmetiques

  operation     resultat, operande1, operande 2  (ARM)


  op1, resultat :       reg
  op2           :       reg ou constante contenue dans code-op
                        (cte : entier naturel sur 8 bits + décalage)

  add/sub       r0, r1, r2
  add/sub       r0, r1, #45

  addS/subS   : idem + mise à jour de ZNCV


V) Instructions d'accès a la memoire

  L'adresse est de la forme     reg + déplacement avec déplacement : reg ou cte_
8
  On prend le point de vue du processeur :

  charger un registre à partir de la memoire :

  ldr reg_dest,   [reg, reg_ou_cte]   @ reg_dest <- Mem [reg + reg_ou_cte]
  str reg_source, [reg, reg_ou_cte]   @ Mem [reg + reg_ou_cte] <- reg_source

  par défaut transfert sur 32 bits. Variantes :
  ldrsh, ldrh, strh     pour demi-mots de 16 bits signés/non signés
  ldrsb, ldrb, strb     pour octets signés/non signés


  Cycles mémoire : 1 cycle de lecture d'instruction (code-op)
                   1 cycle de lecture/écriture de donnée

VI) Chargement des constantes dans un registre

Adressage immediat : la constante est encodée dans l'instruction ou la suit immé
diatement

Cas simple : la cte tient sur 8 bits

    mov   reg, #cte

Cas general : la cte est quelconque

    ldr   reg, =cte    (équivaudrait a mov reg, #cte si cte sur 32 bits)
et  .ltorg en fin de .text
```