INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

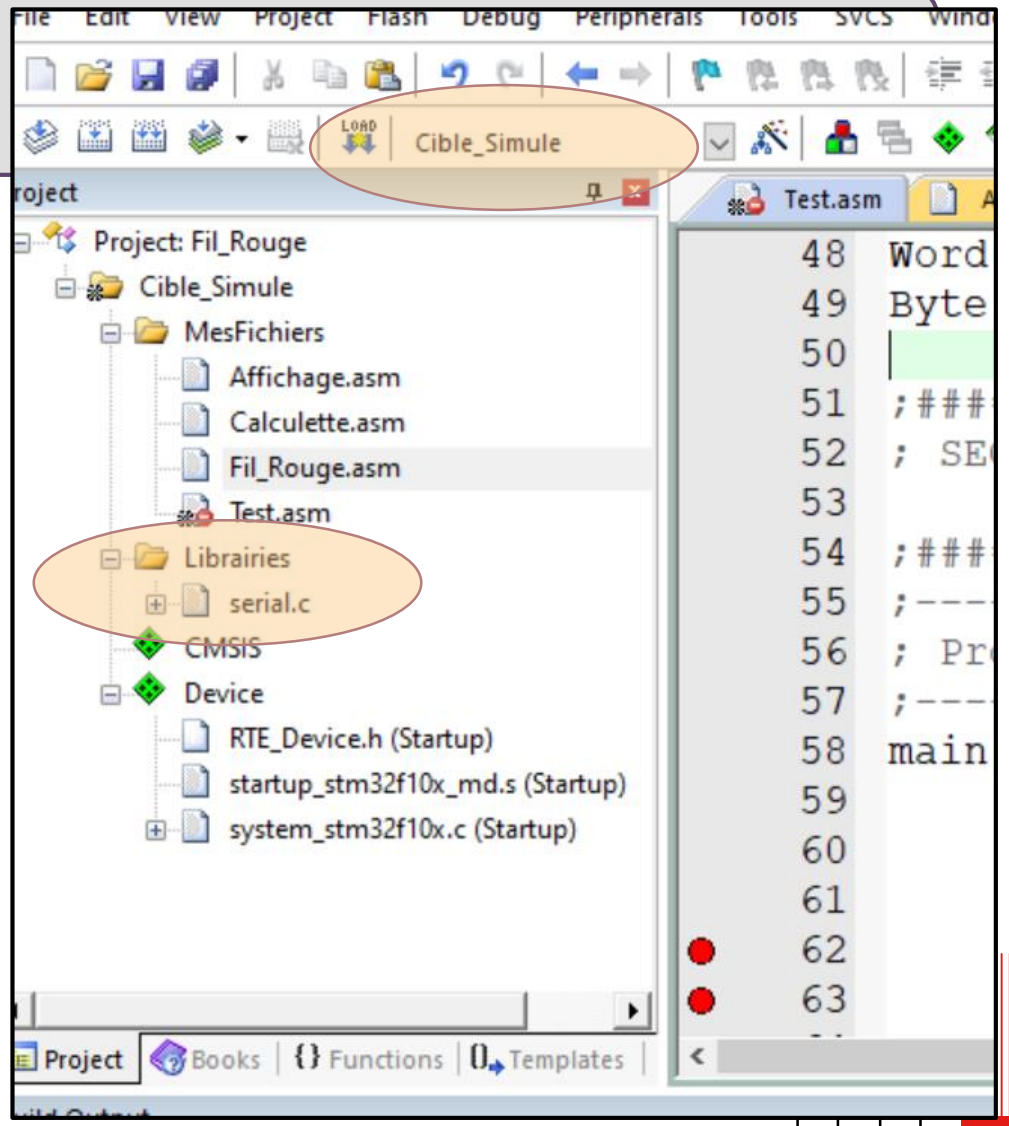
Ecrire une application

Du jeu d'instructions à l'exécutable en passant par l'algorithme

Vincent MAHOUT - INSA Toulousec

Constitution d'un projet

- Une application = un ensemble de fichiers
  - + il pourrait y en avoir qu'un seul mais…
  - + ils ne sont pas tous de même nature : LA, langage C, fichier obj, fichier lib,..
  - + certains sont donnés par le « constructeur » : *startup*….
- Des fichiers peuvent être communs à plusieurs projets
- Un projet peut être conçu plusieurs cible
  - + cible(s) simulée(s)
  - + cible(s) réelle(s)
  - + …



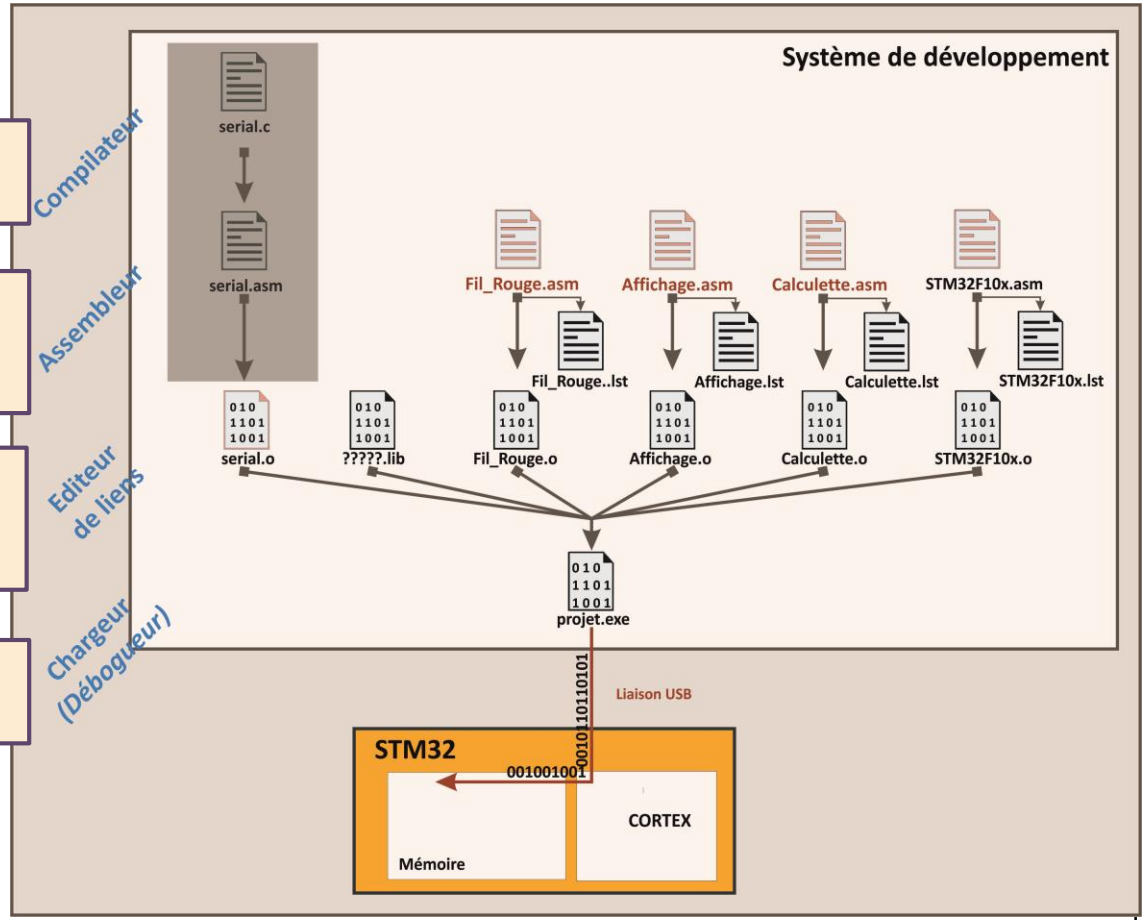
Les étages d'une IDE (Keil µVision)

Traduit le listing en langage d'assemblage
Les fichiers « asm » pas obligatoirement créés

Traduit le listing en langage machine…
Les fichiers « objet » construit sur des adresses relatives ou incomplètes

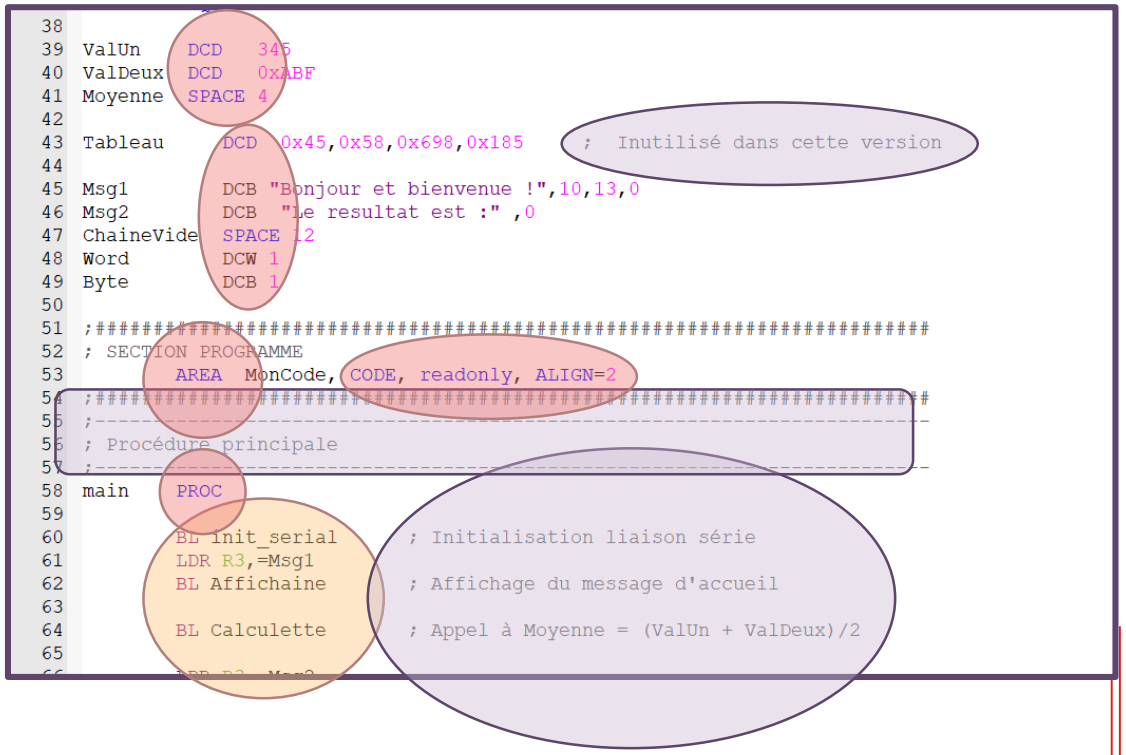
•Réunit les différents objets…
- Les adresses deviennent absolues
- Les « liens » sont faits

•Charge le fichier exécutable en mémoire
- Protocole (physique et logiciel) de communication



La notion de directive en L.A.

- Lecture d'un bout de listing
  - + les instructions et les opérandes : OK
  - + des commentaires : OK
  - + des mots-clés du langage (en bleu) qui ne sont pas des instructions : ???
- Ce sont des DIRECTIVES d'assemblage
  - + ne créent pas de code directement
  - + informations données à l'assembleur/linker pour construire l'executable
- Exemple : `PROC` indique que l'étiquette auquelle elle est attachée pourra être une procédure



La notion de directive en C

- La notion de directive existe en langage structuré
- Exemple en langage C

```
#define NoteMax  15
#include <stdio.h>
#if cond   ….  #endif
int Dugenou;
void  Carabosse (char *);
```

- Aucune ligne ne sera « exécutée »…mais sans ces lignes le programme n'est pas compilable.
- Notamment : déclaration de variables !

Directive de création de variables…ou presque !

- En LA. : la notion de variables n'existe pas !
- Variable en C = un nom, un type, une adresse, une valeur
  - + Le compilateur se charge
    - de la réservation mémoire
    - de l'initialisation éventuelle
    - la cohérence de l'utilisation de la variable
- En LA : pas de notion de type
- « Variable »= simple association d'un nom symbolique avec une adresse mémoire
  - + L'assembleur/linker se charge
    - de la réservation mémoire (en gérant l'alignement)
    - de l'initialisation éventuelle

Réservation mémoire : principe

- Idée : ne pas gérer soi-même la répartition mémoire
- Déclarer des zones mémoires en leur donnant :
  - + un nom symbolique
  - + une dimension (en octet)
  - + éventuellement des valeurs initiales
- Laisser à l'assembleur (et surtout à l'éditeur de liens) le soin d'y attribuer une adresse
- Le programmeur ne connait que le symbole
- L'assembleur/linker crée une table de correspondance entre le nom symbole et l'adresse mémoire => fichier MAP

```
SetSysClock                             0x08000149   Thumb Code     8  system_stm32f10x.o(.text.SetSysClock
[Anonymous Symbol]                      0x08000148   Section        0  system_stm32f10x.o(.text.SetSysClock
SetSysClockTo72                         0x08000151   Thumb Code   290  system_stm32f10x.o(.text.SetSysClock
[Anonymous Symbol]                      0x08000150   Section        0  system_stm32f10x.o(.text.SetSysClock
[Anonymous Symbol]                      0x08000274   Section        0  system_stm32f10x.o(.text.SystemInit)
[Anonymous Symbol]                      0x080002dc   Section        0  serial.o(.text.init_serial)
[Anonymous Symbol]                      0x0800037c   Section        0  serial.o(.text.sendchar)
MonCode                                 0x080003ac   Section       76  affichage.o(MonCode)
MonCode                                 0x080003f8   Section       36  calculette.o(MonCode)
MonCode                                 0x0800041c   Section       56  fil_rouge.o(MonCode)
i.__scatterload_copy                    0x08000454   Section       14  handlers.o(i.__scatterload_copy)
i.__scatterload_null                    0x08000462   Section        2  handlers.o(i.__scatterload_null)
i.__scatterload_zeroinit                0x08000464   Section       14  handlers.o(i.__scatterload_zeroinit)
MesDonnees                              0x20000000   Section       87  fil_rouge.o(MesDonnees)
Msg1                                    0x2000001c   Data          25  fil_rouge.o(MesDonnees)
Msg2                                    0x20000035   Data          18  fil_rouge.o(MesDonnees)
ChaineVide                              0x20000047   Data          12  fil_rouge.o(MesDonnees)
STACK                                   0x20000058   Section     1024  startup_stm32f10x_md.o(STACK)

Global Symbols

Symbol Name                             Value     Ov Type        Size  Object(Section)

BuildAttributes$$THM_ISAv4$P$D$K$B$S$PE$A:L22UL41UL21$X:L11$S22US41US21$IEEE1$IW$~IW$USESV6$~STKCKD$USESV7$~
__cpp_initialize__aeabi_                  - Undefined Weak Reference
__cxa_finalize                            - Undefined Weak Reference
__decompress                              - Undefined Weak Reference
_clock_init                               - Undefined Weak Reference
_microlib_exit                            - Undefined Weak Reference
__Vectors_Size                          0x000000ec   Number         0  startup_stm32f10x_md.o ABSOLUTE
__Vectors                               0x08000000   Data           4  startup_stm32f10x_md.o(RESET)
__Vectors_End                           0x080000ec   Data           0  startup_stm32f10x_md.o(RESET)
__main                                  0x080000ed   Thumb Code     0  entry.o(.ARM.Collect$$$$00000000)
```

Exemple : Msg1 créé dans fil_rouge est une DATA stocké à l'@ 0x2000001C

Réservation simple

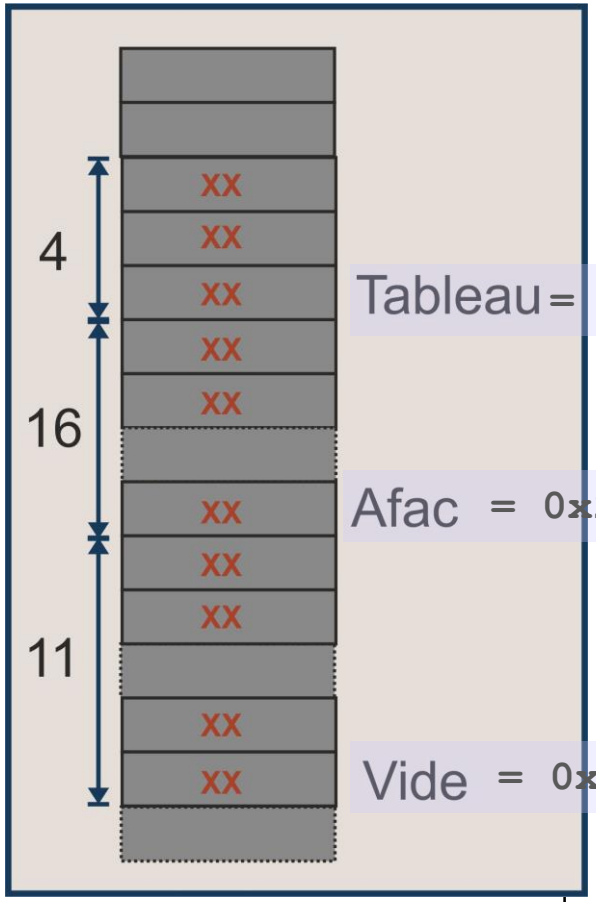
- Directive d'assemblage : **SPACE**
- Syntaxe :

```
{nom} SPACE expr
```

  - **expr** exprime un taille en case mémoire (octet)
  - **expr** est donnée généralement en base 10 (mais pas que!) et peut contenir des expressions mathématiques simples

- Exemple :

```
Vide SPACE 11
Afac SPACE 0x0B+(2*2)+1
Tableau SPACE 2_100
```



Si Vide est placé en mémoire par le linker en `0x20000000`
⇒ Afac sera en `0x2000000B` et Afac sera en `0x2000001B`

Réservation avec initialisation

- Directive d'assemblage : **DCB, DCW, DCD, DCQ**
- Syntaxe :

  {nom} DCx expr1,[expr2],[expr3],…

  - **DCB** pour Byte (1 octet),
  - **DCW** pour Word (2 octets),
  - **DCD** pour Double (4 octets),
  - **DCQ** pour Quadruple (8 octets)

  + **expri** exprime une valeur initiale
  + **expr1** est obligatoire, **expr2, expr3,…** facultatives
  + Autant de réservations que de valeurs données
  + Alignement obligatoire….

Grosse incohérence du nom des directives DCW et DCD

Pour les instructions LDRH : H pour half-word Récupération d'un 16 bits ⇒ Word = 32 bits !

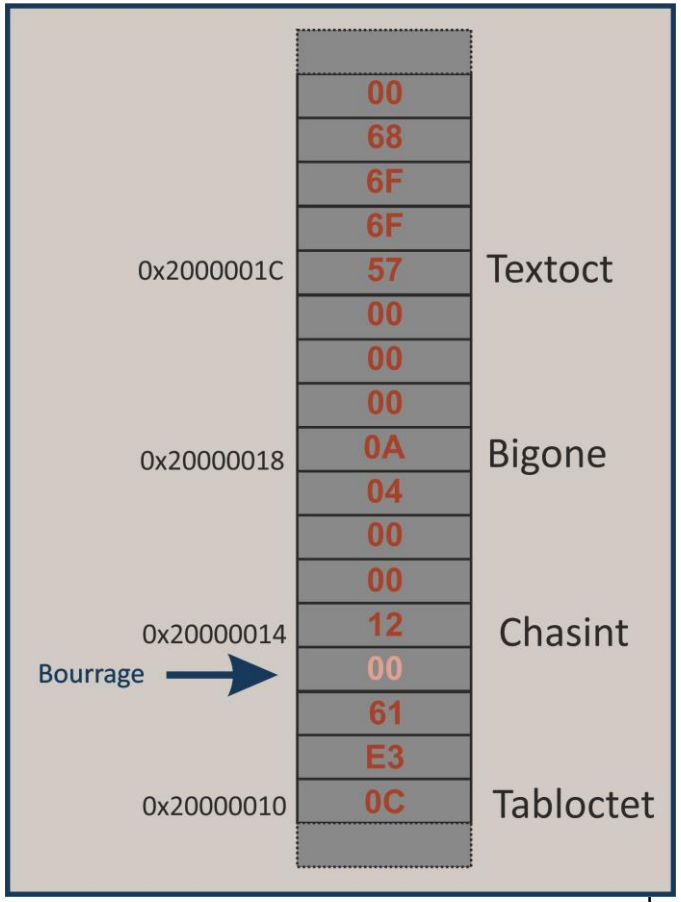
Pour les directives le Word n'est que de 16 bits !

Réservation avec initialisation : exemple

```
Tabloctet DCB 12,0xE3, 'a'
Chasint DCW 0x12, 1024
Bigone DCD 10
Textoct DCB "Wooh",0
```

Supposons que le linker commence à l'adresse mémoire `0x20000010`

- Remarques :
  - l'affichage est ici « normalisée » en hexadécimal
  - les caractères sont possibles, on visualise ici leur code ascii ('a' =0x61)
  - les ≠initialisations sont séparées par une virgule sauf pour les chaines de caractères
  - respect de la convention little-endian
  - la réservation respecte l'alignement :
    - un `DCW` doit correspondre à une adresse paire
    - Un `DCD` doit correspondre à une adresse doublement paire
    - Si ce n'est pas le cas ⇒insertion d'octet(s) dit de Bourrage (initialisé à `0`)



Rappel sur l'utilisation d'une variable

- Accès uniquement par LDR/STR

```
ADD R4,Taboctet
```

- Impossible de les utiliser en accès direct

```
LDR R4,Taboctet
```

```
LDR R4,[Taboctet]
```

- Exemple : incrémentation d'un octet signé
  - + récupération de l'adresse :

```
LDR R0,=Taboctet
```

  - + lecture de l'octet

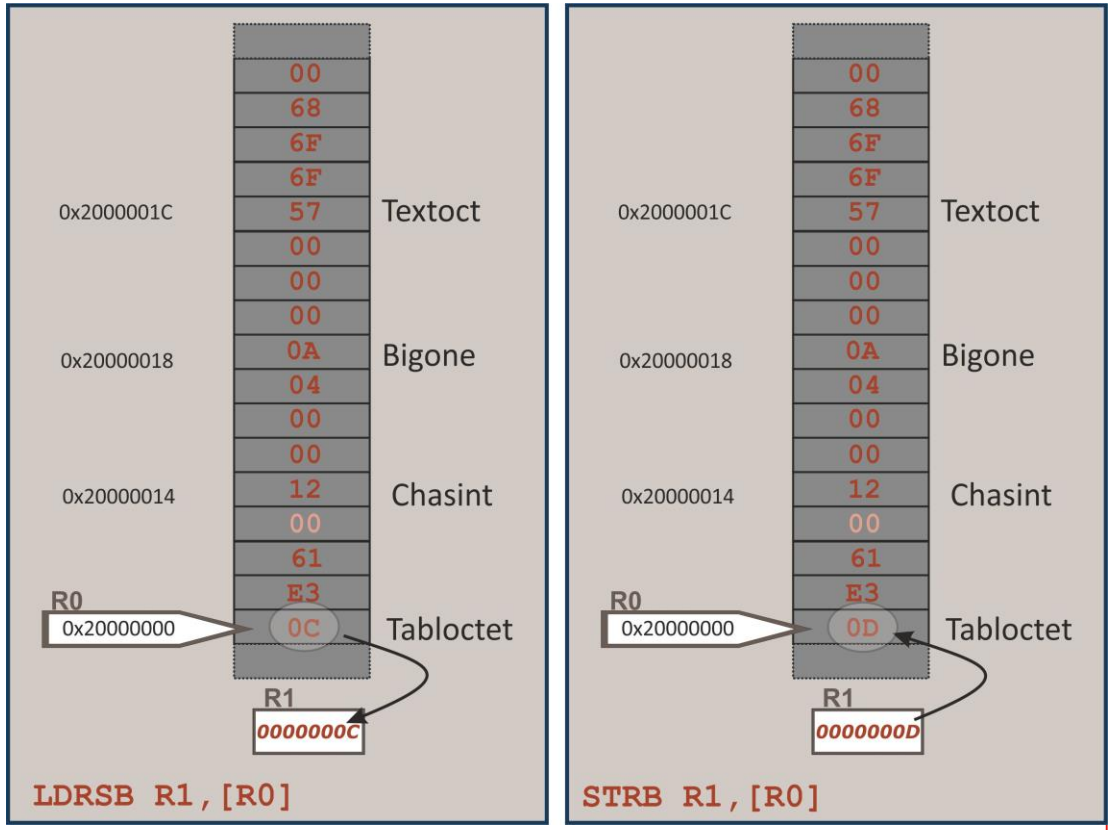
```
LDRSB R1,[R0]
```

  - + incrémentation

```
ADD R1,#1
```

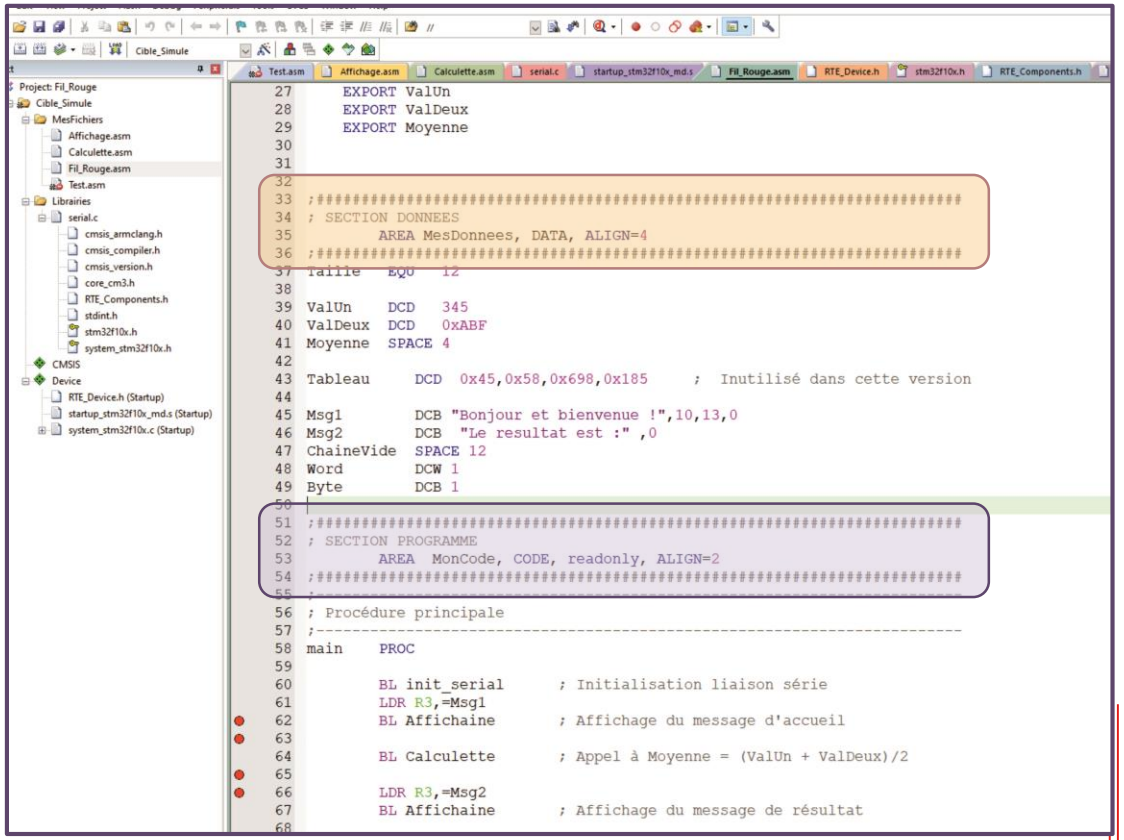
  - + stockage du résultat

```
STRB R1,[R0]
```



La notion de section

- Architecture Harvard
  - + séparation nette CODE et DATA
- Séparation également sur le listing
  - + notion de section (AREA)
- Directive AREA essentielle pour le linker
  - + attribution des adresses mémoire au différentes étiquettes
- Un programme = minimum 2 sections
  - + une section de code contient la liste des instructions.
  - + une section de données où se trouve la description des données (nom, taille, valeur initiale)



La directive AREA

- La directive d'assemblage **AREA**
  - marque le debut d'une section
- Une section se termine :
  - avec l'ouverture d'une autre section
  - avec le fin du fichier
- Syntaxe :

```
AREA Nom_Section {,attr}{,attr} ....
…
… Corps de la section
… définitions des
… données ou instructions
…
```

- Les options (`{attr}`) principales
  - type : section de code (**CODE**) ou une section de données (**DATA**)
  - Conditions de placement en mémoire.
    - `align = n` : débute à une @ modulo 2^n
  - Accès : en lecture seule (**readonly**) ou en lecture/écriture (**readwrite**)
- ∃ nombreuses et complexes : sauf avis contraire : on laisse l'assembleur et le linker se débrouiller …
- Le boulot de l'éditeur de liens :
  - un projet = plusieurs fichiers, donc plusieurs sections de code et data.
  - les réunir, les associer et faire les liens pour que les différentes parties correspondent.

```
AREA MesDonnees, DATA, ALIGN=4
AREA MonCode, CODE, readonly, ALIGN=2
```

Création de constantes

- Constante : association d'un symbole et d'une valeur
- Intérêt : lisibilité et maintenance
- En langage C : `#define`
- Syntaxe :

```
Symbole EQU Valeur
```

- Où :
  - n'importe où avant l'utilisation (même hors section)
- Ce que fait l'assembleur :
  - Find(`Symbole`) - Replace(`Valeur`)
- Utilisation dans le code
  - Le symbole en valeur immédiate

- Exemples :

```
Boucle EQU 12
VRAI   EQU 0xFF
FAUX   EQU #0
Redite EQU Boucle
```

- Attention aux #

```
MOV R3, Faux ⇔ MOV R3, #0
```

Celle-ci me semble plus lisible…

```
MOV R3, #Vrai ⇔ MOV R3, #0xFF
```

```
MOV R3, Vrai ⇔ MOV R3, 0xFF
```

La directive PROC

- On a vu l'instruction **BL Nom** qui permet de faire un saut avec lien (**LR**)
- **Nom** est une simple étiquette
  - + Pas besoin de plus pour l'aspect LA
- On peut « encapsuler » l'ensemble de la procédure avec :

```
PROC
… corps de la procédure
BX LR
ENDP
```

- Utilité :
  - + lisibilité
  - + au niveau du débogueur

- Exemple :

```
56  Convertit   PROC
57              PUSH {R1-R3,R5,R8-R11}
58              MOV R2,R0
59              ADD R0,#5
60              MOV R8,#0
61              STRB R8,[R0,#-1]!
62              MOV R6,#0X000F
63  SurQuatre   AND.W R8,R7,R6
64              CMP R8,#9
65              BHI Hexa
66              ADD R8,#'0'
67              B Suite
68  Hexa        ADD R8,#('A'-10)
69  Suite       STRB R8,[R0,#-1]!
70              LSR R7,#4
71              CMP R2,R0
72              BNE SurQuatre
73              POP {R1,R2,R6,R7,R8}
74              BX LR
75              ENDP
76
```

Directive d'IMPORT /EXPORT

- La compilation (assemblage) séparée :
  - un projet = plusieurs fichiers
  - chaque fichier doit pouvoir être *compilé* (*assemblé*) indépendamment des autres ⇒ création du fichier `obj`.
  - Les fichiers peuvent avoir des variables et/ou des fonctions partagées
- Le problème :
  - Si un nom de variable est le même dans 2 fichiers, est-ce la même variable ?
- La réponse :
  - dire qu'une variable (ou une fonction) est visible de l'extérieur : **EXPORT**
  - dire qu'une variable (ou une fonction) utilisée viendra de l'extérieur : **IMPORT**

```
;*************************************
; IMPORT/EXPORT Système
;*************************************

    EXPORT main

;*****************************
; IMPORT/EXPORT Propre au pro
;*****************************

    IMPORT Affichaine
    IMPORT Convertit
    IMPORT Calculette
    IMPORT init_serial

    EXPORT ValUn
    EXPORT ValDeux
    EXPORT Moyenne
```

Etiquette main : point d'entrée de votre application

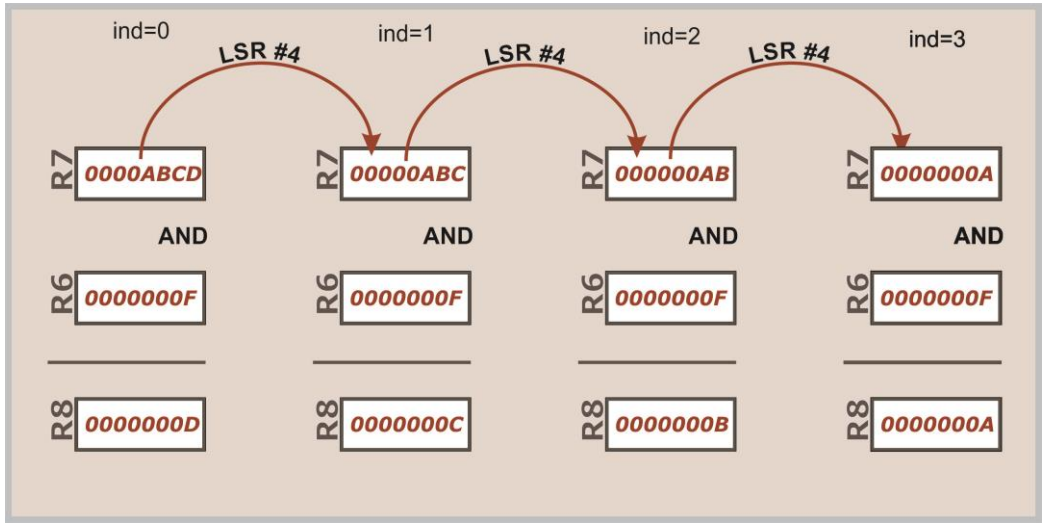
Comprendre ?

- Lecture d'un bout de code :

```
56  Convertit   PROC
57              PUSH {R1-R3,R5,R8-R11}
58              MOV R2,R0
59              ADD R0,#5
60              MOV R8,#0
61              STRB R8,[R0,#-1]!
62              MOV R6,#0X000F
63  SurQuatre   AND.W R8,R7,R6
64              CMP R8,#9
65              BHI Hexa
66              ADD R8,#'0'
67              B Suite
68  Hexa        ADD R8,#('A'-10)
69  Suite       STRB R8,[R0,#-1]!
70              LSR R7,#4
71              CMP R2,R0
72              BNE SurQuatre
73              POP {R1,R2,R6,R7,R8}
74              BX LR
75              ENDP
76
```

Boucle

- Comment le comprendre ?
  - Connaître les conventions d'appels
    - R7 valeur à convertir
    - R0 tableau de la chaine de caractères
  - Il y a une boucle de 4 tours



Comprendre : ce qui manque

- LA ne fait pas apparaitre l'algorithme

```
56  Convertit   PROC
57              PUSH {R1-R3,R5,R8-R11}
58              MOV R2,R0
59              ADD R0,#5
60              MOV R8,#0
61              STRB R8,[R0,#-1]!
62              MOV R6,#0X000F
63  SurQuatre   AND.W R8,R7,R6
64              CMP R8,#9
65              BHI Hexa
66              ADD R8,#'0'
67              B Suite
68  Hexa        ADD R8,#('A'-10)
69  Suite       STRB R8,[R0,#-1]!
70              LSR R7,#4
71              CMP R2,R0
72              BNE SurQuatre
73              POP {R1,R2,R6,R7,R8}
74              BX LR
75              ENDP
76
```

```
Convertit(Chaine,Mot)
    Ptr ← @Chaine +5
    *(Ptr) ← 0
    Pour ind =1 à 4
      Val = Octet(Mot,ind)
      Si Val < 10
              *(Ptr) ← Val +'0'
      Sinon
             *(Ptr) ← Val +'A'-10
      FinSi
      Prt ← Ptr -1
    FinPour
```

Pas de procédure Octet… combinaison déclage et masque

L’algorithmie : la base

- Uniquement la structuration de base
  - + Structures alternatives
    - Si…Alors
    - Si…Sinon…Alors
    - Switch Case
  - + Structures itératives
    - Tant que…Faire
    - Répéter…Jusqu’à
    - Pour….
- Si plus compliqué….imbrication

Tout se construit à partir de sauts conditionnés

Bxx Point_RdV

Alternative : la simple

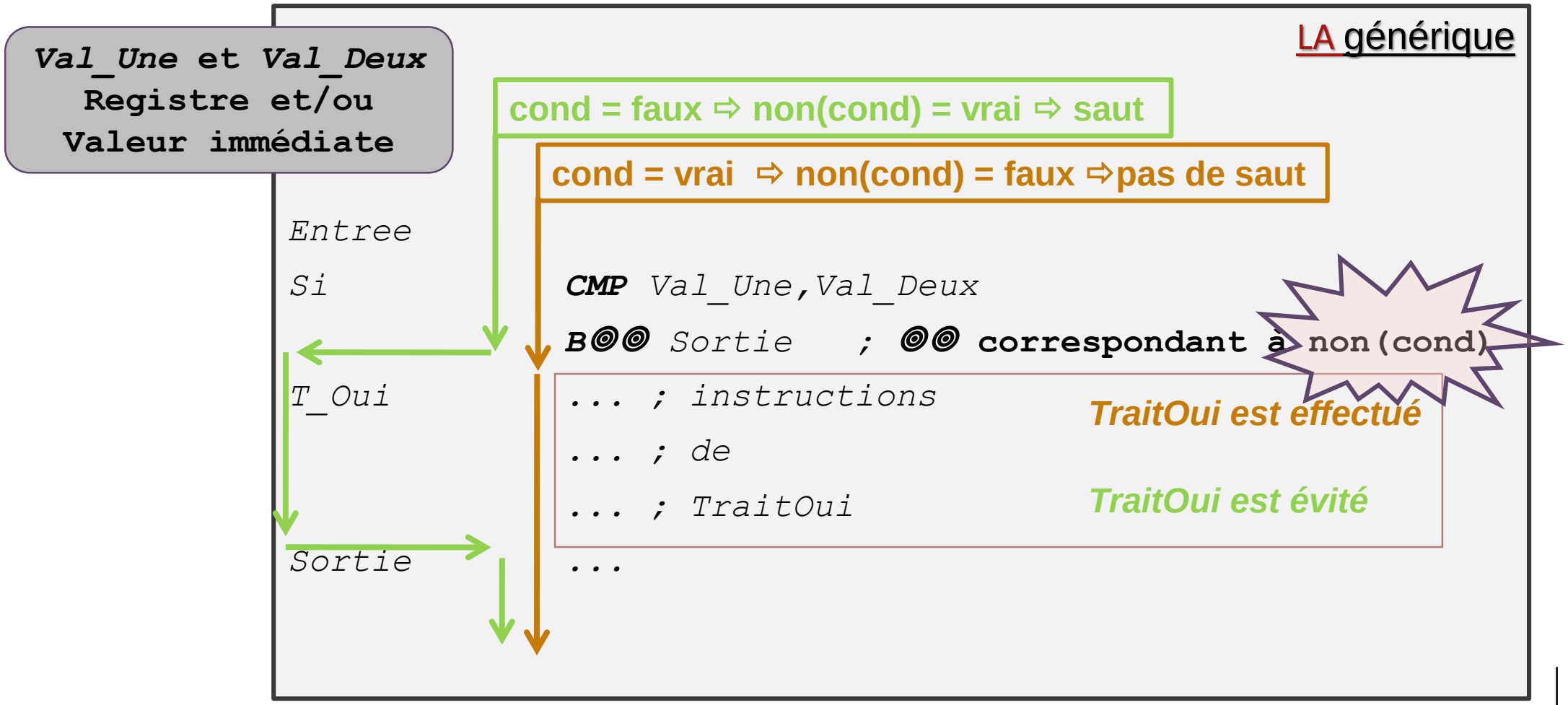
- Algorithme :

```
Si (cond) Alors
|   Faire Trait
Fin Si
```

- Le test (`cond`) :
  - + état des fanions
  - + suite à une instruction de test : CMP, TST,...
  - + suite à une instruction qcq + suffixe S
- le saut conditionné = en fonction des fanions

ATTENTION : le saut permet d'éviter les instructions qui suivent ⇒ complément logique de la condition exprimée dans l'algo

Alternative : la simple



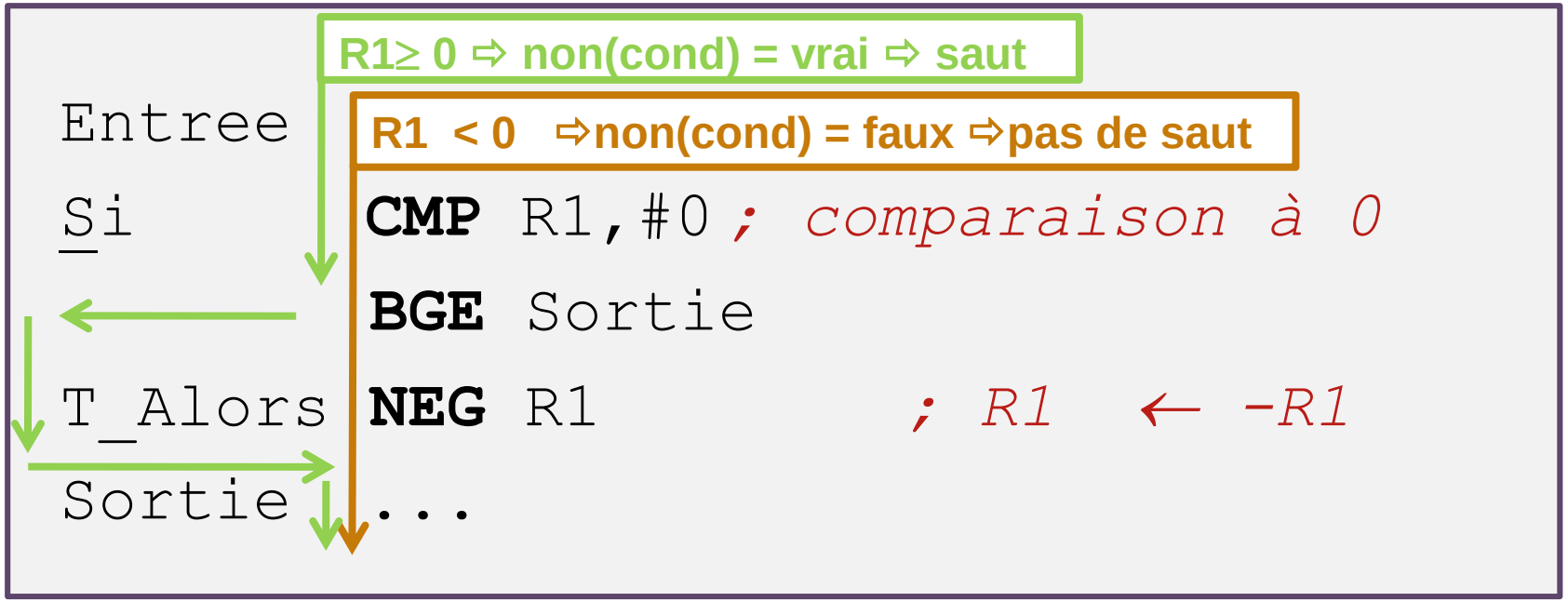
Alternative simple : exemple

```
Si (Entier < 0) Alors
| Entier ← -Entier
Fin Si
```

Entier : stocké dans **R1**

Cond : négatif

non(cond) : positif ou nul = **GE**



Alternative : la complète

- Algorithme :

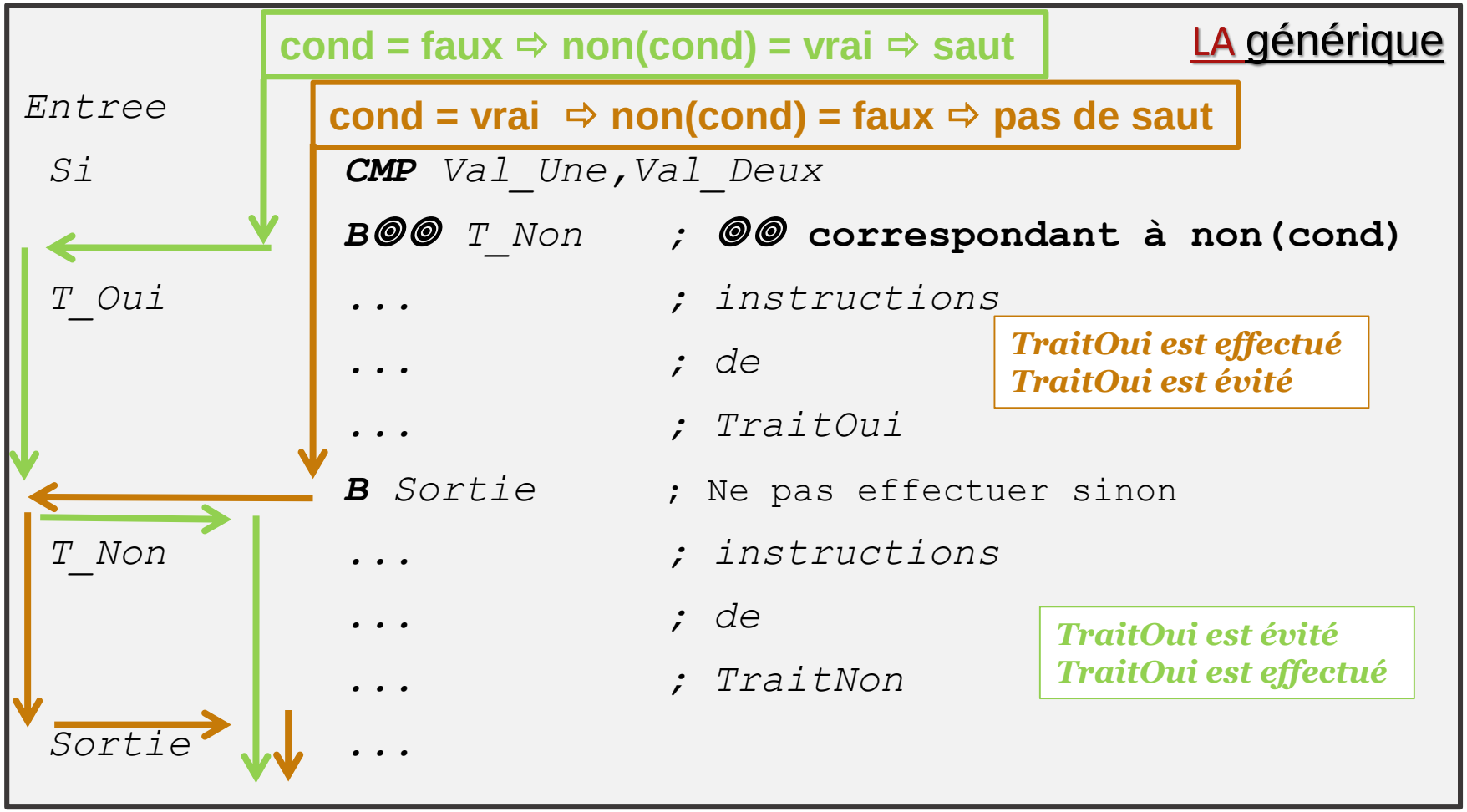
```
Si (cond) Alors
|   Faire Traitoui
|Sinon
|   Faire Traitnon
Fin Si
```

- Le test (`cond`) :
  - Alternative toujours « inversible »

```
Si non(cond)
|Alors
|Faire Traitnon
|Sinon
|Faire Traitoui
Fin Si
```

- Idée du codage :
  - avoir le même ordre d'apparition de `Traitoui` et `Traitnon` sur l'algo que dans le code.

Alternative : la complète



Alternative complète : exemple

```
Si (Entier ≤ 0) Alors
|       Ptr ← @Msg1
| Sinon Ptr ← @Msg2
Fin Si
Affiche(Ptr)
```

Entier : stocké dans **R3**

Ptr : stocké dans **R0**

Cond : négatif ou nul

Non(cond) : positif = **GT**

```
Entree
Si      CMP R3,#0          ;Comp. à 0
        BGT Sinon
Alors   LDR R0, =Msg1      ; Ptr ← @Msg1
        B Sortie           ; On "saute" le bloc Sinon
Sinon   LDR R0, =Msg2      ; Ptr ← @Msg2
Sortie  BL Affichaine      ; Appel à la procédure d'affichage
        ...+
```

R3 > 0
⇨ non(cond) = vrai
⇨ ⇨ saut

R3 ≤ 0
⇨non(cond) = faux
⇨pas de saut

Alternative : la complète allégée

- Algorithme :

```
Faire Traitnon
Si (cond) Alors
|    Faire Traitoui
Fin Si
```

- Idée :
  - faire un choix par défaut
  - se ramener à une alternative simple.
- Contrainte :
  - `Traitoui` et `Traitnon` sans interférence mutuelle
- Exemple qui n'irait pas :

```
Si x<0
| Alors
|         X = 1/(4+x²)
| Sinon
|         X = 1/(4+√x)
Fin Si
```

Alternative complète : exemple modifié

```
Si (Entier ≤ 0) Alors
|       Ptr ← @Msg1
| Sinon Ptr ← @Msg2
Fin Si
Affiche(Ptr)
```

Entier : stocké dans **R3**

Ptr : stocké dans **R0**

Cond : négatif ou nul

Non(cond) : positif = **GT**

```
Entree  LDR R0,= Msg2   ; choix par defaut
Si      CMP R3,#0       ; comparaison à 0
        BGT Sortie
Alors   LDR R0= Msg1    ;Ptr ← @Msg1
Sortie
        BL Affichaine   ; Appel à la procédure d'affichage
        ...+
```

R3 > 0
⇨ **non(cond) = vrai**
⇨ **saut**

R3 ≤ 0
⇨**non(cond) = faux**
⇨**pas de saut**

Le *switch case :* 3 possibilités

- Imbrication de Si…Sinon…Si…Sinon….
  - + pénible à écrire et à maintenir
- Utilisation des instructions TBB et TBH
  - + usage limité (comme pour l'instruction IT)
  - + peu générique
- Fabrication d'une table de saut
  - + générique
  - + peut même se mettre dans une procédure-type :
    - `Souitche (TableSaut, ValTest)`

Le *switch case :* exemple

- Je veux faire :

```
Selon que variable vaut
    valeur_0 : faire traite_0
    valeur_1 : faire traite_1
    valeur_2 : faire traite_2
    . . .
    autres : faire Tdefaut
Fin Selon que
```

- ...et avec le code assembleur :

```
traite_0    . . .
            B Out
traite_1    . . .
            B Out
traite_2    . . .
            B Out
Tdefaut     . . .
Out           . . .
```

B Out correspond au break du langage C

Le *switch case : codage de l'*exemple

- Réalisation de la table de saut
- Les étiquettes sont des adresses (32 bits)
- Autant d'entrées que de valeurs….

```
Table_de_Saut
    DCD traite_0
    DCD traite_1
    DCD traite_2
    …
    DCD traite_n
    DCD Tdefaut
```

- Utilisation :
  - la table facilement accessible si :
    Val_0 = 0
    Val_1 = 1
    …
    Val_n = n
  - Sinon : rajout de l'indexation
- La valeur (supposons R6) sert d'index dans la table

```
LDR R9, = Table_de_Sots
LDR R2,[R9,R6,LSL #2]
BX R2
```

- Saut par double indirection !
  - `R9` pour accéder à la table
  - `R2` pour se brancher sur le point de RdV

Sélective : Tant que ... faire ou Répéter … jusqu'à

- Algorithme :

```
Tant que(cond)
|   Faire Trait
Fin Tant que
```

- Le test est fait au début
- Evitement possible dès le premier tour

Le passage doit modifier l'objet du test

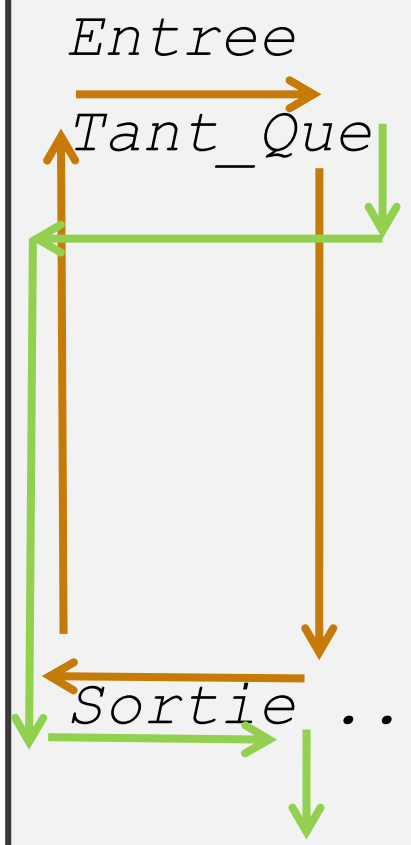
- Algorithme :

```
Répéter
|   Trait
Jusqu'à (cond)
```

- Le test est fait à la fin
- Au moins un passage dans la boucle

Alternative : test en début

LA générique



```
Entree
Tant_Que  CMP  Val_1,Val_2 ; @@ correspondant à
          B@@  Sortie      ;          non(cond)
               ...         ; Instructions
               ...         ; de
               ...         ; traitement
               B  Tant_Que
Sortie ...
```

Cycle : tant que cond = vrai
⇒ non(cond) = faux
⇒ pas de saut

Sortie : cond = faux
⇨ non(cond) = vrai
⇨ saut

Boucle Pour….

```
Pour Ind allant de I0 à In
|                     pas de dI
|    Faire Trait
Fin Pour
```

N'existe pas pour cet assembleur !

⇒ Transforme en **Tant que (cond)… Faire :**

```
Ind = I0
Tant Que (Ind < In)
|    Faire Trait
|    Ind = Ind + dI
Fin TQ
```

Conditions composées

- Que se passe t'il si :
  - tant que (CondA ET CondB)….
  - si (CondA OU CondB) ….
  - bref …la condition n'est pas unique et simple
- Cela peut devenir fort complexe :
  - si ((Marcel < 10 ET (Edith = Calcul(Marcel) < 12))
- La réponse (encore une fois) :
  - rien de « natif » en assembleur
  - ⇒enchaînement des instructions

Prudence est mère de sûreté !

Composition en ET

```
Si (Cond1 ET Cond2)
|    Alors Trait
FinSi
```

La seconde expression n'est pas évaluée si la première est fausse !

LA générique

```
Entree
        CMP Val,ValCon_1
        B@@ Sortie   ; @@ correspondant à non(cond1)
        CMP Val,ValCon_2
        B@@ Sortie   ; @@ correspondant à non(cond2)
          ...        ; Instructions
          ...        ; de
          ...        ; Traitement
Sortie    ...
```

Enchaînement de deux si

```
Si (Cond1) Alors
| Si(Cond2) Alors
| |   Trait
| FinSi
FinSi
```

Composition en ET

```
Si (Cond1 OU Cond2)
|    Alors Trait
FinSi
```

Toujours enchaînement de deux si :

Imbrication plus « tordue »

```
Si (Cond1) Alors
|     goto OK
|  Sinon Si(Cond2)Alors
|  |    OK : Trait
|  FinSi
FinSi
```

C'est le seul cas de figure du codage explicite de Cond et pas de non(Cond)

LA générique

```
Entree
                CMP Val,ValCon_1
                B✪✪ Zyvas   ; ✪✪ correspondant
                                à Cond1
                CMP Val,ValCon_2
                B◎◎ Sortie    ; ◎◎ correspondant
à non(cond1)
 Zyvas    ...     ; Instructions
          ...     ; de
          ...     ; Traitement
 Sortie   ...
```

La seconde expression n'est pas évaluée si la première est vraie !