

Architecture (ASR1 et ASR2)

TP N° 3

1 Objectifs :

- Prise en main d'emu8086.
- Etude du langage machine
- Ecriture d'un premier programme en assembleur 8086
- Premières gestion des entrées/sorties

2 Organisation du TP :

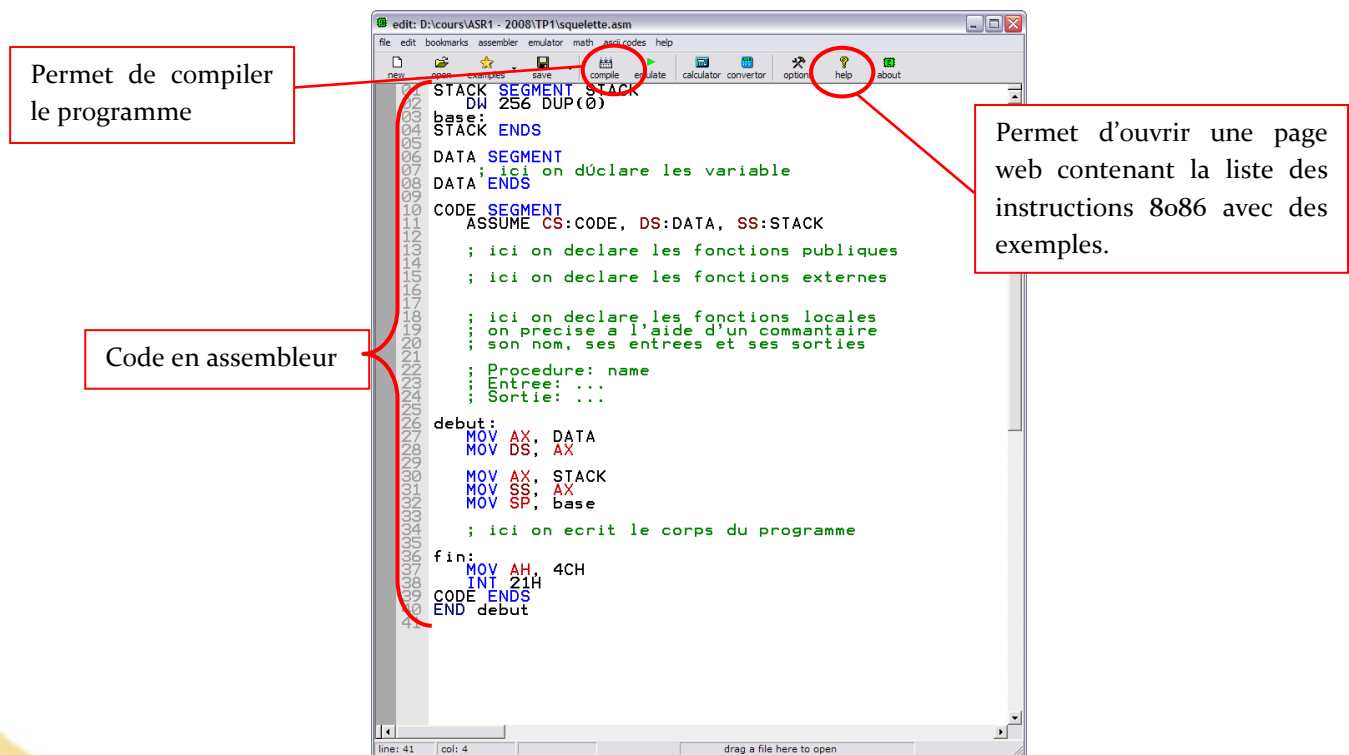
Le travail effectué durant cette séance sera à rendre à votre responsable de TP par email impérativement avant la fin de la séance. Le sujet de l'email sera [2012][Architecture][TP3] Nom Prénom .

Le TP comporte plus d'exercices que vous ne pourrez normalement en faire pendant la séance. Ces exercices supplémentaires vous sont donnés à titre d'entraînement et votre responsable de TP pourra vous les corriger si vous les lui rendez. Attention, corriger vos exercices ne veut pas dire qu'il vous fournira une correction type des exercices ! Il vous indiquera des pistes pour résoudre vos problèmes et pour vous aider à trouver une solution correcte qui fonctionne.

3 Exercice 1 : Emu8086 (Obligatoire)

Dans un premier temps, pour simplifier l'apprentissage de l'assembleur x86, nous allons travailler sur un émulateur logiciel qui simule un processeur 8086. L'utilisation de cette émulation n'est pas obligatoire car nos programmes, écrits en assembleur 8086, fonctionnent normalement sur un processeur moderne comme l'Intel core i7, ou l'AMD Phenom.

Emu8086 se présente dans un premier temps comme un éditeur de texte classique, avec le support d'une colorisation syntaxique du code assembleur (mode éditeur).



Architecture (ASR1 et ASR2)

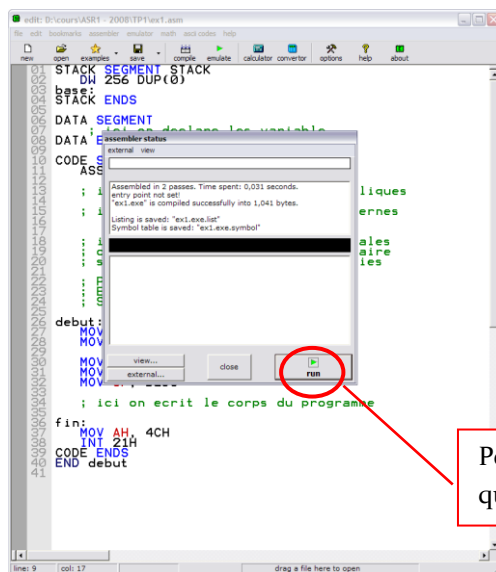
TP N° 3

Copiez le répertoire `r:/emu8086` sur votre compte (`p:/Archi/`). Copier le fichier `squelette.asm` dans `p:/Archi/TP1/`. Ouvrez le fichier avec `emu8086` (celui de votre compte).

Le fichier `squelette.asm` est un fichier que nous vous fournissons pour vous aider à écrire vos programmes assembleur. Il contient la déclaration d'un segment de pile de 512 octets, la déclaration d'un segment de données vide ainsi que la déclaration d'un segment de code. Dans ce dernier, on retrouve la directive `ASSUME` permettant d'initialiser les trois registres de segments `CS`, `DS` et `SS`. On trouve aussi une zone de déclaration de fonctions publiques, une zone de déclaration de fonctions externes, une zone de déclaration de fonctions locales, puis du code assembleur.

Ce code débute par l'étiquette `debut`, puis on retrouve l'initialisation des segments de données et de pile (cette initialisation est doublée pour des raisons de compatibilité). Un espace réservé à l'écriture de vos programmes suit ces éléments. Enfin, l'étiquette `fin` indique (informellement) la fin du programme. Les deux lignes suivantes correspondent à l'appel standard de l'interruption DOS pour indiquer la fin d'un programme. Ensuite on trouve la fin du segment de code puis celle du programme.

Pensez lorsque vous écrirez un programme à bien partir de ce fichier `squelette` en le sauvegardant sous un autre nom avant de le modifier.



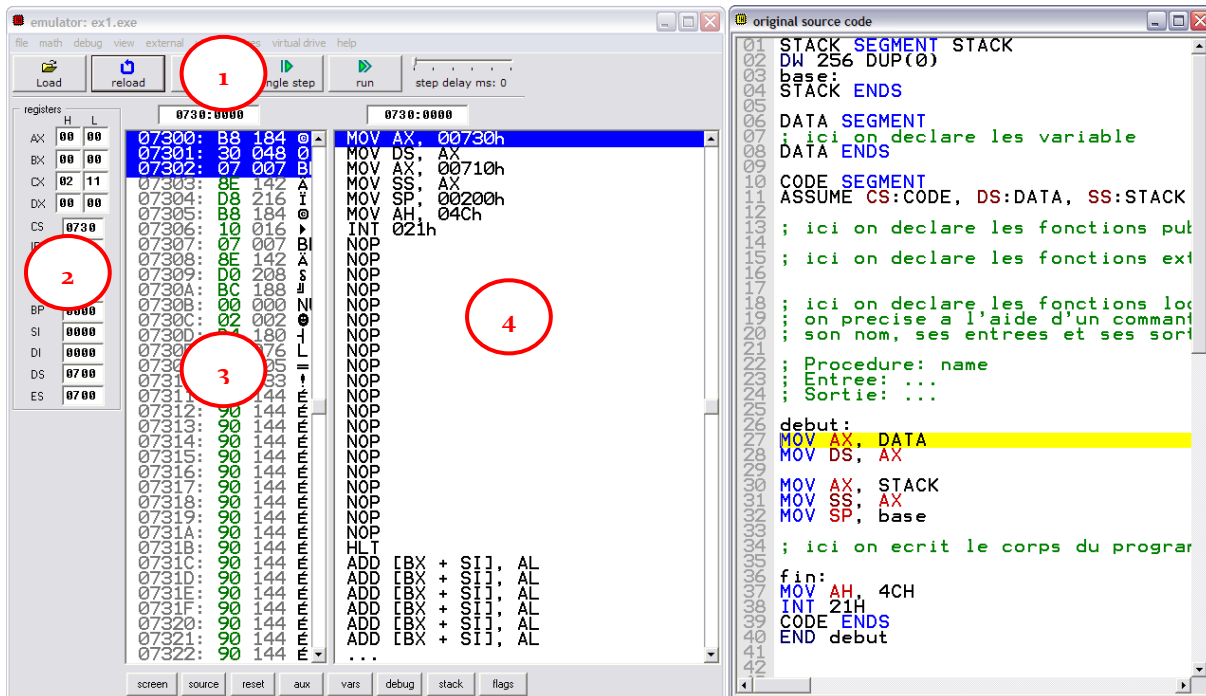
Permet d'exécuter le programme qu'on vient de compiler

Créez un fichier `ex1.asm` identique au fichier `squelette.asm`. Compilez et exécutez le programme (bouton `RUN`). Validez la fin de l'exécution avec `OK`.

Deux nouvelles fenêtres viennent de s'ouvrir, il s'agit du mode exécution de l'émulateur. La fenêtre « *original source code* » contient le code tel que vous l'avez écrit. La ligne surlignée en jaune est la prochaine instruction qui va être exécutée. Comme on le verra plus loin dans ce TP, il est possible d'interagir avec ce code en cliquant dessus. La zone mémoire de la fenêtre « *emulator :* » change pour correspondre à la partie sélectionnée par le clic.

Architecture (ASR1 et ASR2)

TP N° 3



La fenêtre « emulator : » contient 4 zones que nous allons détailler ci-dessous :

1. Les boutons vont nous permettre d'exécuter notre programme soit *Pas à Pas*, avec la possibilité de revenir en arrière, soit automatiquement (en pouvant régler un délai d'attente entre chaque instruction)

Réinitialisez votre programme (reload) puis exécutez-le pas à pas (single step et back step) jusqu'à la fin. Faites une nouvelle réinitialisation, réglez le délai sur 400 ms et faites une exécution normale (run)

2. La deuxième zone contient une représentation des registres, et permet de visualiser leur valeur hexadécimale tout au long de l'exécution de votre programme. On notera que les registres AX, BX, CX et DX sont représentés coupés en deux. Cela permet de visualiser simplement leur décomposition en registres 8 bits. Il est possible de modifier la valeur des registres de cette manière (simple clic) ou d'ouvrir une vue détaillée du registre (double clic).

Exécutez votre programme et observez les valeurs des registres durant l'exécution. Vous porterez une attention particulière au registre IP.

Après l'exécution de la première instruction, quelle est la valeur décimale du registre AX ? Quelle est l'adresse de l'étiquette Base dans le segment de Pile ? Et quelle est son adresse dans la mémoire ?

3. La zone 3 représente la mémoire octet par octet. Sur une ligne, on retrouve l'adresse mémoire sur 20 bits, la valeur de l'octet en hexadécimale, en décimale et sa représentation en ASCII.

Pourquoi l'adresse est-elle représentée sur 20 bits ? Quel est le code instruction en langage machine qui correspond à l'instruction MOV DS, AX ?

4. La 4ème zone contient le code de votre programme une fois les traductions d'adresses terminées. Il n'y a plus aucune étiquette ni nom de variable. Il s'agit du code réellement exécuté.

Quel est le code réellement exécuté correspondant à votre programme (entre l'étiquette debut et la fin du segment de code) ? Quelle serait la première instruction (en code réellement exécuté) si la pile avait une taille de 1024 octets ?

4 Exercice 2 : Les données (Obligatoire)

Nous allons maintenant nous intéresser au segment de données. Pour plus d'information sur la création de variables, reportez-vous à votre cours.

Architecture (ASR1 et ASR2)

TP N° 3

En partant du squelette que vous sauvegarderez sous le nom `ex2.asm` (par la suite cette information ne sera pas rappelée, à vous d'y penser), créez dix variables de type octet, nommées `bo` à `b9` et contenant les valeurs décimales 1, 255, -1, 'e', les valeurs hexadécimale 32, `ff` et les valeurs binaires 1, 1111111.

Créez les mêmes variables de type mot que vous nommerez `do`, `di`, ...

Créez ensuite un tableau de 10 octets nommé `tb` qui contient lui aussi ces valeurs. Faites de même avec un tableau de 10 mots (`td`).

Créez ensuite un tableau de 152 octets contenant la valeur décimale 111 et un tableau de 150 mots contenant la valeur décimale 43981.

Compilez et exécutez votre programme en mode pas à pas. En cliquant sur le nom des variables (fenêtre original source code) vérifiez les valeurs en mémoire. Que dire des valeurs signées ? Et des caractères ? Quel est la différence entre une variable de type mot et de type octet ? Quel est la différence entre un tableau et une suite de variables ? Avez-vous remarqué autre chose ? Quelle est la taille du segment de données ?

5 Exercice 3 : Etude d'un programme simple (Obligatoire)

Laissons de côté `emu8086` pour le moment. Regardez attentivement ce programme et répondez aux questions suivantes sans compiler ce programme.

```

FILE SEGMENT STACK
    DW 256 DUP(?)
Base:
FILE ENDS

DATA SEGMENT
    N1 DW 5
    N2 DW 3
    N3 DW 3 DUP(?)
    N4 DB ?
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:PILE
main:
    MOV AX,DATA    ; initialisation du segment de données
    MOV DS,AX

    MOV AX,PILE
    MOV SS,AX      ; initialisation du segment de pile
    MOV SP,Base    ; Initialise la pile à vide

    MOV AX,N1
    CMP AX,N2
    JE etiq
    MOV N4,0
    JMP fin

etiq:
    MOV N4,1

fin:
    MOV AH,4CH     ; Les 2 lignes nécessaires pour la...
    INT 21H        ; ...fin du programme
CODE ENDS
END main

```

Architecture (ASR1 et ASR2)

TP N° 3

Que fait ce programme ? Quelle sera la valeur de N_4 à la fin de l'exécution du programme ? Les valeurs de N_1 , N_2 , N_3 sont-elles modifiées ? La pile est composée de combien d'octets ?

Combien y a-t-il de variables créées dans le segment de données ? Quelle taille occupent-elles ? Que se passe-t-il si vous enlevez l'instruction `JMP fin` ? Quelle sera la valeur de N_4 dans ce cas-là à la fin de l'exécution du programme ?

Revenons maintenant à emu8086. Nous allons voir que certaines options du menu `view` peuvent être pratiques. Voyons rapidement les principales options disponibles (certaines sont accessible directement par des boutons situés sous la zone 3) :

- *Stack*: affiche la pile.
- *Variable*: affiche la liste des variables et leurs valeurs.
- *Memory*: affiche une partie de la mémoire. La visualisation en mode table permet plus facilement de se repérer.
- *Flag*: affiche la valeur des différents drapeaux.
- *Ascii code*: affiche une table ascii au cas où vous auriez oublié la vôtre.

Pour en finir avec les menus, dans `math`, vous trouverez deux outils pour vous aider dans les conversions de base et les calculs.

Complétez le fichier `squelette.asm` pour qu'il corresponde au programme ci-dessus. Attention, à ne pas modifier inutilement le fichier `squelette`. Le résultat sera sauvegardé dans le fichier `ex3.asm`. Exécutez pas à pas ce programme et vérifiez les réponses aux questions.

6 Exercice 4 : Ecriture d'un programme simple (obligatoire)

6.1 Somme de 2 variables de type mot

Ecrivez un programme `ex4a.asm` qui fait la somme de 2 variables de type mot (précédemment déclarées) et qui sauvegarde le résultat dans une troisième variable de type mot. Faites des tests avec des valeurs positives et négatives.

6.2 Somme de 2 variables de type octet

Ecrivez un programme `ex4b.asm` qui fait la somme de 2 variables de type octet (précédemment déclarées) et qui sauvegarde le résultat dans une troisième variable de type mot.

6.3 Somme de 2 variables

Ecrivez un programme `ex4c.asm` qui fait la somme de 2 variables de type mot (précédemment déclarées) et qui sauvegarde le résultat dans une variable de type octet si cela est possible ou dans une variable de type mot si le résultat ne tient pas dans une variable de type octet. Faire des tests avec les valeurs négatives.

7 Exercice 5 : Ecriture d'un programme avec Entrées/Sorties (optionnel)

Pour la gestion des entrées (lecture de texte au clavier) ou les sorties (affichage à l'écran), nous allons utiliser les interruptions DOS. Comme le mécanisme d'interruption ne sera vu que plus tard dans le cours, il ne vous est pas demandé de connaître actuellement le fonctionnement interne des interruptions mais d'être capable d'utiliser celles-ci (de la même manière que vous êtes capable d'utiliser Windows sans en connaître le fonctionnement en détail).

En assembleur, on appelle/exécute une interruption à l'aide de la commande `INT`. Cette instruction à un seul paramètre qui est le numéro de l'interruption. La valeur de ce numéro étant codé sur 8 bit, seul 255 interruptions différentes sont possibles (noté de 0 à FFh). Par exemple, `INT 8` appelle l'interruption numéro 8.

Architecture (ASR1 et ASR2)

TP N° 3

Pour ne pas être limité à cette valeur, chaque interruption va également lire une valeur dans un registre (dans notre cas ça sera AH) qui lui indique quelle sous interruption est réellement appelé. Par exemple, AH, 01h / INT 21h permet de lire un caractère au clavier alors que AH, 02h / INT 21h permet d'afficher un caractère à l'écran.

Regardons maintenant quelques exemples d'interruption DOS :

- **Exemple 1** : lecture d'un caractère au clavier
 - `MOV AH, 01H`
 - `INT 21H`
 - Cette fonction permet de lire un caractère sur l'entrée standard. Ce caractère est récupéré sous la forme de son code ASCII dans AL.
 - Entrée : AH = 01h
 - Sortie : AL = contient le code ASCII du caractère saisi au clavier
- **Exemple 2** : affichage d'un caractère à l'écran
 - `MOV DL, 'A'`
 - `MOV AH, 02H`
 - `INT 21H`
 - Cette fonction permet de sortir (afficher) un caractère sur la sortie standard. Ce caractère doit être stocké sous la forme de son code ASCII dans DL.
 - Entrée : AH = 02h
 - DL = Code ASCII du caractère à afficher
- **Exemple 3** : Lecture d'un caractère au clavier
 - `MOV DL, offset MaChaine`
 - `MOV AH, 09H`
 - `INT 21H`
 - Cette fonction permet d'afficher une chaîne de caractères sur la sortie standard. Cette chaîne doit être stockée sous la forme d'une séquence d'octets correspondant aux codes ASCII des caractères composant la chaîne. La fin de la chaîne de caractères doit être signalée à l'aide du caractère '\$'.
 - Entrée : AH = 09h
 - DS = Adresse de segment de la chaîne de caractères
 - DX = Adresse d'offset de la chaîne de caractères

7.1 Version A

Ecrivez un programme `ex3a.asm` qui fait la somme de 2 valeurs (entre 0 et 4) qui seront lues au clavier. Ensuite vous afficherez le résultat (entre 0 et 8) à l'écran.

7.2 Version B

Ecrivez un programme `ex3b.asm` qui fait la somme de 2 valeurs (entre 0 et 9) qui seront lues au clavier. Un message d'invite sera affiché avant la lecture des valeurs au clavier. Ensuite vous afficherez le résultat (entre 0 et 18) à l'écran avec un message clair.

7.3 Version C

Ecrivez un programme `ex3c.asm` qui est une extension du programme précédent dans lequel vous gérerez la lecture clavier d'un nombre sur 8 bits (entre 0 et 255) et dont le résultat pourra être entre 0 et 510.