Amélioration d'un logiciel d'émulation du fonctionnement d'un microprocesseur x86

SCHAEFFER Florian
ROUSSEAUX Billy
L3 Informatique

Responsable du projet: M. Jean-Michel RICHER

2007-2008

Université d'Angers
Département Informatique

Table des matières

I – Introduction

Il est parfois difficile pour un étudiant qui doit étudier un nouveau langage de maîtriser tous les outils à sa disposition : compilateurs, assembleurs, débuggeurs,… surtout s'ils sont dépourvus d'interface graphique alors que le monde de Windows a incité nos esprits d'informaticiens à oublier la ligne de commande.
C'est conscients de cette difficulté que nous avons travaillé sur ce projet d'émulateur de fonctionnement d'un microprocesseur x86.

Le projet avait été développé en java au cours de l'année 2001 par deux étudiants de licence : Frédéric BEAULIEU et Yan LE CAM. Notre objectif était bien défini : reprendre le code et y apporter des améliorations. D'une part des améliorations graphiques mais également des améliorations au niveau du moteur de l'émulateur, pour prendre en compte les évolutions technologiques des processeurs.

Nous ne sommes pas habitués à travailler sur un logiciel développé par d'autres. Il nous paraissait important de bien comprendre le programme que nous devions améliorer.
C'est pourquoi nous avons passé beaucoup de temps au début de notre projet à essayer d'entrer dans la pensée des précédents développeurs, à comprendre l'organisation de leur code. C'était une étape nécessaire pour ensuite aller plus loin dans les améliorations. C'est cela que nous allons présenter dans une première partie intitulée « Analyse de l'existant » : dans un premier temps, nous présenterons l'ancienne interface graphique et ensuite les différentes fonctionnalités de la version de base.

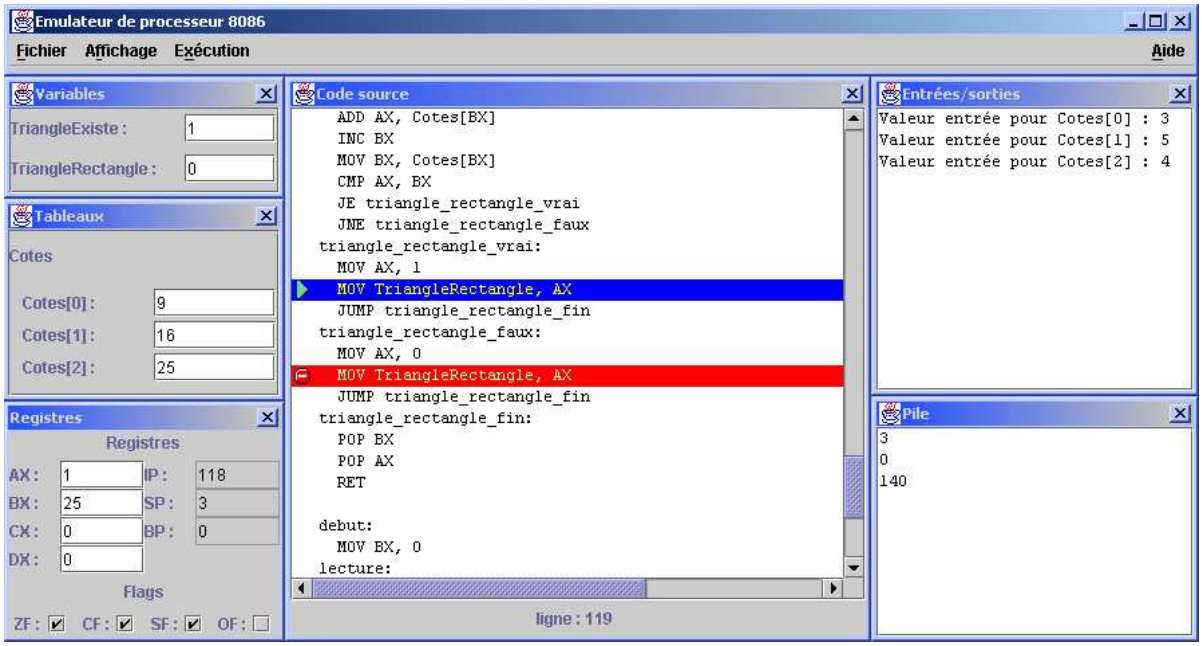
C'est une fois ce travail d'analyse en profondeur effectué que nous avons pu entrer dans la phase plus intéressante des modifications et améliorations. L'amélioration la plus visible est bien sûr celle de l'interface graphique. Nous allons donc vous présenter les modifications que nous avons jugées utiles d'apporter pour que l'interface soit plus claire, mieux organisée et plus pratique à utiliser.
Ensuite, nous exposerons les améliorations que nous avons implantées au niveau du cœur de l'application. En effet, depuis que le projet initial a été développé, des améliorations technologiques importantes ont été ajoutées aux microprocesseurs. D'une part, la taille des registres est passée de 16 à 32 bits et d'autre part de nouvelles instructions agissant sur de nouveaux registres fonctionnant différemment sont apparues : les instructions SSE. Nous allons donc expliquer toutes ces améliorations qui ont nécessité des changements conséquents dans le programme de base.

II – Analyse de l'existant

1 - L'interface

La première approche que nous avons eue avec l'application réalisée par Fréderic Baulieu et Yan Le Cam est bien sûr l'interface graphique. Nous avons en effet pris un peu de temps pour tester l'application avant de tenter de la modifier pour nous rendre compte par nous-mêmes de son fonctionnement ainsi que des points que nous allions avoir à modifier.

Au lancement de l'application, on observe sept fenêtres ayant chacune un contenu précis. Ces fenêtres sont indépendantes en terme de position, elle sont bien réparties sur l'écran au démarrage, avec la possibilité de choisir de n'afficher que certaines d'entre-elles, grâce au menu affichage.



Vue écran du logiciel original

Le logiciel est donc composé à son lancement de 7 partie distinctes : en haut, la barre de menu, à droite, la fenêtre contenant les variables normales, la fenêtre contenant les tableaux et celle contenant les registres et les flags, au centre nous avons le code source et enfin a droite, les entrées/sorties et la pile.
Les informations sont relativement bien réparties, nous allons tenter de garder une disposition similaire. Mais nous allons devoir modifier cette interface et y ajouter certaines parties.

2 - Les fonctionnalités

2.1 - les caractéristiques

Avant de nous lancer dans la modification du projet effectué par nos prédécesseurs, nous avons dû nous pencher sérieusement sur la compréhension du travail qui avait déjà été effectué, sans quoi nous aurions perdu notre temps à modifier sans succès les fichiers qui le constituent.

Après le test du logiciel et la lecture du rapport de stage correspondant à ce projet, nous avons pu déterminer précisément les caractéristiques générales de l'émulateur.
Premier point important, le programme gère uniquement une taille de seize bits, non signé. C'est à dire que pour qu'un programme soit lisible pour l'émulateur, chaque variable ou élément de tableau doit être déclarée comme étant un mot (DW) ce qui autorise des valeurs allant de 0 à 65535. Nous voyons ici le premier point à améliorer.

Pour ce qui est de l'émulation du matériel, le processeur possède 7 registres, 4 registres de données (AX,BX,CX,DX), 2 registres pointeurs de pile (BP et SP) et 1 pointeur d'instruction (IP). Tous ces registres ont une taille de 16 bits et il est encore impossible de manipuler séparément les parties hautes et basses des registres de données. Voilà un second point sur lequel il a été nécessaire de nous pencher.

L'émulateur gère 4 flags :

- ZF : il est mis à 1 lorsque le résultat d'une opération vaut 0
- CF : il est mis à 1 lorsque le calcul comporte une retenue
- OF : il est mis à 1 s'il y a eu un débordement arithmétique (par exemple lors d'une addition
- SF : il est mis à 1 lorsque le résultat d'une opération est négatif

Il existe d'autres flags qui n'ont pas été implémenté initialement car ils n'intervenaient pas dans les instructions supportées par l'émulateur. Nous ne modifierons donc pas cette partie, les seules instructions que nous allons ajouter n'auront pas effet sur d'autres flags que ceux implémentés ici.

En ce qui concerne la mémoire, elle n'est pas gérée strictement comme le ferait un processeur de type 8086. Emul8086 sert à comprendre et déboguer un programme écrit dans un langage proche de l'assembleur, la gestion interne de la mémoire importe peu ici. Nous allons être amenés à modifier les méthodes concernant cette partie de l'application pour notre projet, nous garderons cependant le même principe de gestion.

2.2 - la structure d'un programme

Comme indiqué dans le rapport de projet, les programmes chargés par l'émulateur seront sous forme de fichiers texte, comportant deux section DATA et CODE, mots clé qui devront être dans cet ordre.

La partie DATA servira à déclarer les variables sous la forme suivante :

« nom_variable DW valeur initiale » ou « nom_tableau DW[taille] »

le « DW » servant à indiquer le type de variable. Nous allons être amenés à étendre les types acceptés par l'émulateur et donc permettre la gestion d'autres mots clé.

La partie CODE contiendra les instructions à exécuter, à raison d'une instruction ou étiquette par ligne.
Chaque ligne aura donc la forme suivante :
« Instruction param1 [param2] » ou « étiquette : » (param2 peut ne pas exister selon l'instruction)

Le programme peut également comporter des lignes vides ou des commentaires commençant par un point-virgule.

Le langage d'assemblage supporté par l'émulateur se trouve donc être très proche syntaxiquement d'un langage d'assemblage réel comme celui que nous avons pu étudier au premier semestre de la troisième année de licence.

Le langage utilisé comporte cependant 2 instructions supplémentaires : LIRE et ECRIRE qui permettent de simplifier le mécanisme d'entrées/sorties en laissant le traitement de ces opérations à l'émulateur plutôt qu'au processeur.

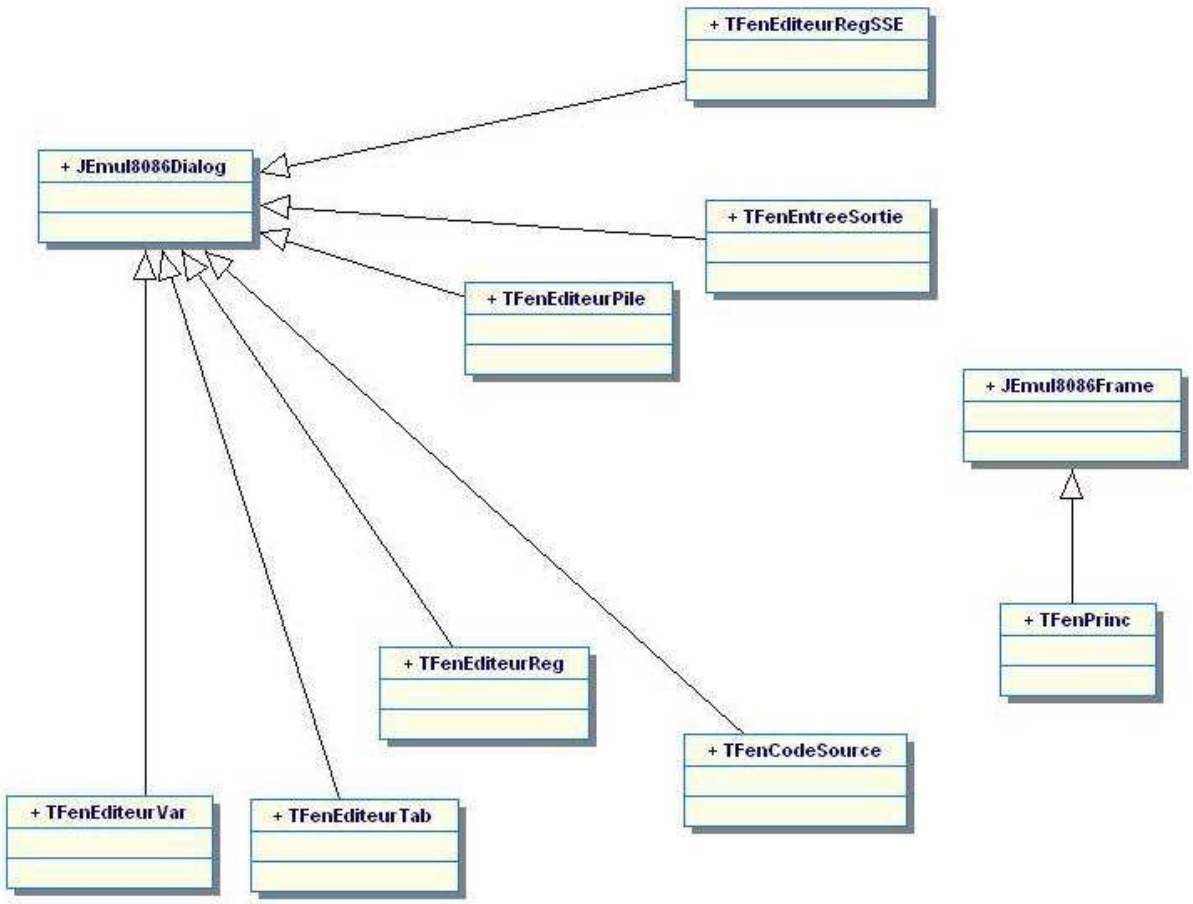
Voici à tire d'exemple un programme supporté par le logiciel original (calcul de la moyenne de cinq nombres demandés à l'utilisateur) :

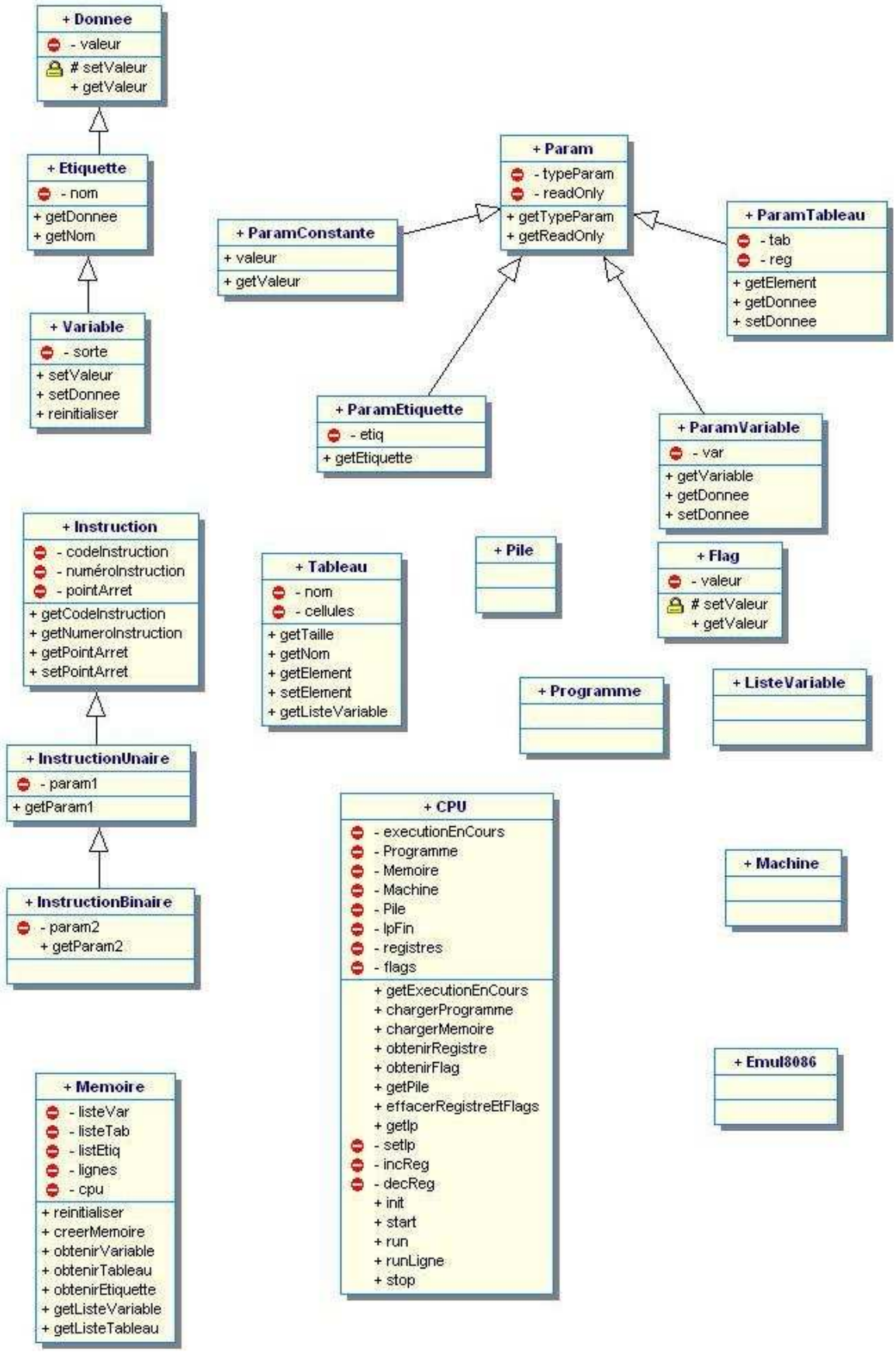
```
; exemple : calcul de moyenne
; déclaration des données
DATA
  Somme   DW 0
  Moyenne DW 0
  Maximum DW 5
  Notes   DW[5]

; instructions à exécuter
CODE
start:
  MOV AX, 0
  MOV BX, 0
loop:
  LIRE Notes[BX]
  ADD AX, Notes[BX]
  INC BX
  CMP BX, Maximum
  JNE loop

  MOV Somme, AX
  DIV BX
  MOV Moyenne, AX
  ECRIRE Moyenne
```

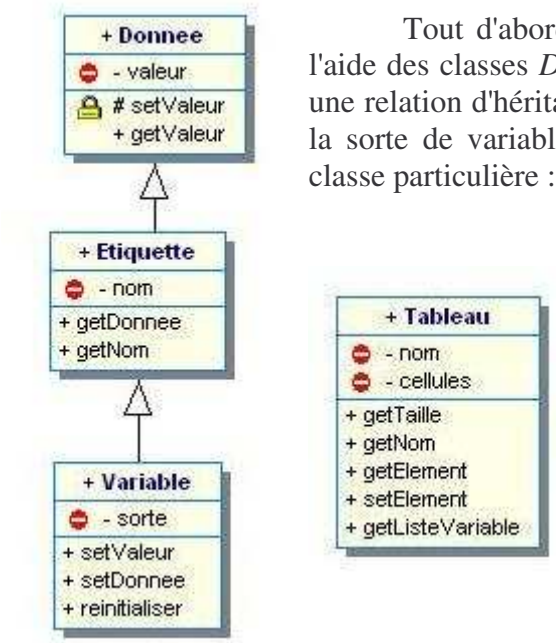
3 - le diagramme de classes




+ Donnee
- valeur
setValeur
+ getValeur
+ Etiquette
- nom
+ getDonnee
+ getNom
+ Variable
- sorte
+ setValeur
+ setDonnee
+ reinitialiser
+ Param
- typeParam
- readOnly
+ getTypeParam
+ getReadOnly
+ ParamConstante
+ valeur
+ getValeur
+ ParamTableau
- tab
- reg
+ getElement
+ getDonnee
+ setDonnee
+ ParamEtiquette
- etiq
+ getEtiquette
+ ParamVariable
- var
+ getVariable
+ getDonnee
+ setDonnee
+ Instruction
- codeInstruction
- numéroInstruction
- pointArret
+ getCodeInstruction
+ getNumeroInstruction
+ getPointArret
+ setPointArret
+ Tableau
- nom
- cellules
+ getTaille
+ getNom
+ getElement
+ setElement
+ getListeVariable
+ Pile
+ Flag
- valeur
setValeur
+ getValeur
+ Programme
+ ListeVariable
+ InstructionUnaire
- param1
+ getParam1
+ InstructionBinaire
- param2
+ getParam2
+ CPU
- executionEnCours
- Programme
- Memoire
- Machine
- Pile
- lpFin
- registres
- flags
+ getExecutionEnCours
+ chargerProgramme
+ chargerMemoire
+ obtenirRegistre
+ obtenirFlag
+ getPile
+ effacerRegistreEtFlags
+ getIp
- setIp
- incReg
- decReg
+ init
+ start
+ run
+ runLigne
+ stop
+ Machine
+ Memoire
- listeVar
- listeTab
- listEtiq
- lignes
- cpu
+ reinitialiser
+ creerMemoire
+ obtenirVariable
+ obtenirTableau
+ obtenirEtiquette
+ getListeVariable
+ getListeTableau
+ Emul8086

4 - Explication du fonctionnement

4.1 - les types



Tout d'abord, les variables et les registres sont représentés à l'aide des classes *Donnee*, *Etiquette* et *Variable*. Ces classes liées par une relation d'héritage contiennent respectivement la valeur le nom et la sorte de variable correspondante. Les tableaux font l'objet d'une classe particulière : ils contiennent un vecteur de variables.

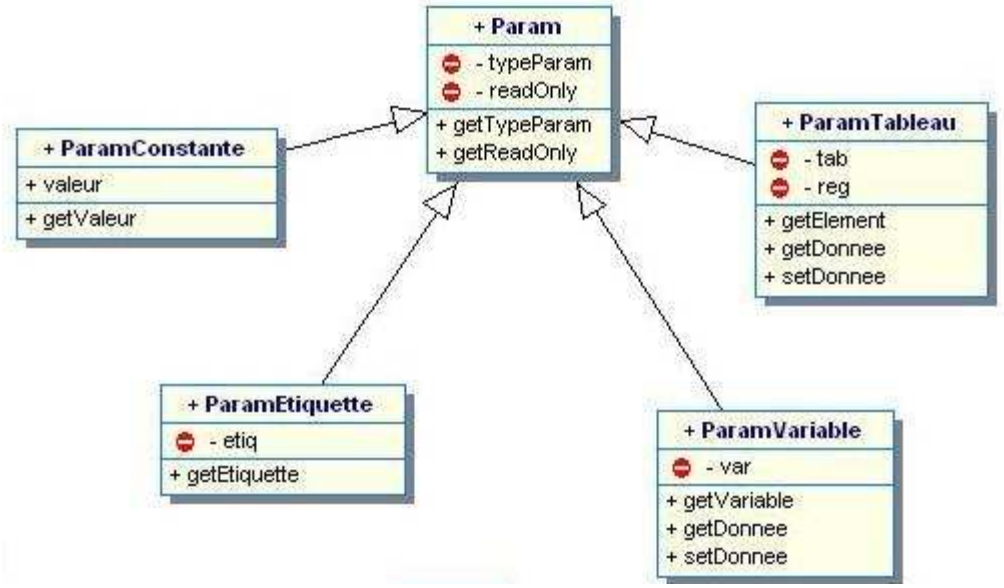
4.2 - Les flags

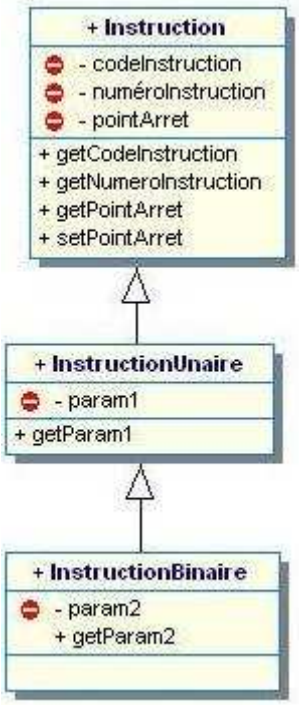
Les flags sont représentés par une classe composée seulement d'un booléen et de deux accesseurs.

4.3 - La mémoire

Une des parties les plus importantes du logiciel est la mémoire. Une instance de la mémoire est créée par la machine et liée au processeur. Cette classe contient la liste des variables utilisées et la liste des tableaux (déclarées dans la section DATA du programme) ainsi que la liste des étiquettes (déclarées dans la section CODE) et un vecteur des différentes lignes du programme dans son ensemble. Cette classe contient également les méthodes pour remplir ces champs lors de la traduction du programme et les méthodes pour y accéder lors de son exécution.
La méthode *creerMemoire* effectue une première analyse syntaxique du fichier texte fourni lors d'un chargement. Elle crée des instances de variables, de tableau et d'étiquettes qui seront ensuite utilisées deux fois : dans un premier temps pour vérifier la partie CODE du programme, en vérifiant l'existence des variables utilisées, et ensuite lors de l'exécution pour agir sur ces variables.

4.4 - Instructions et paramètres





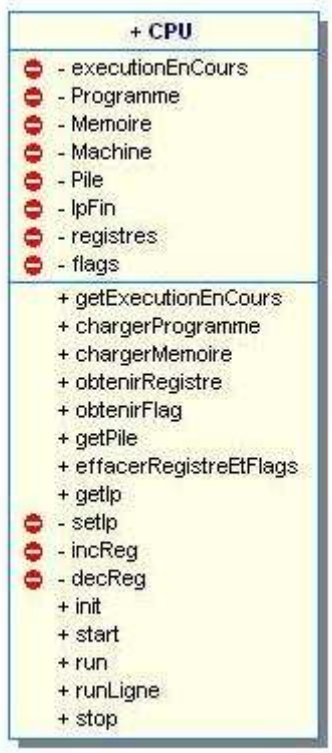
Les classes *Instruction*, *InstructionUnaire* et *InstructionBinaire* servent à stocker les instructions du programme en instructions exécutables par le microprocesseur de l'application. Ces classes contiennent des instances des objets sur lesquels elle agit ainsi qu'un code d'instruction destiné au processeur.

Les paramètres sont une instance d'une des classes précédentes en fonction de leur type (i.e. le type de la variable sur lequel elles agissent)
Chacune de ces classes contient donc le paramètre situé en mémoire sur lequel porte l'instruction.

4.5 - Programme

Une autre partie conséquente du programme est la classe *Programme*. Après le chargement d'un programme et suite à la création de la mémoire comme expliqué précédemment, une deuxième analyse syntaxique est effectuée pour cette fois-ci traduire les instructions en créant des instances de le classe *Instruction*. C'est ici que les erreurs de paramètres, ou les utilisations de variables non déclarées sont détectées.

4.6 - le CPU



Le cœur de l'application est bien évidemment le CPU. Il permet d'exécuter les instructions traduites par une instance de la classe *Mémoire* et une instance de la classe *Programme*. On retrouve également parmi les champs les registres et les flags.

On y trouve aussi les méthodes permettant les différents types d'exécution.
La méthode getIp permet de connaître l'exécution en cours, ce qui permet les trois modes de déroulement du programme: Exécution complète, exécution pas à pas et exécution jusqu'à une ligne donnée.

III – Les modifications apportées

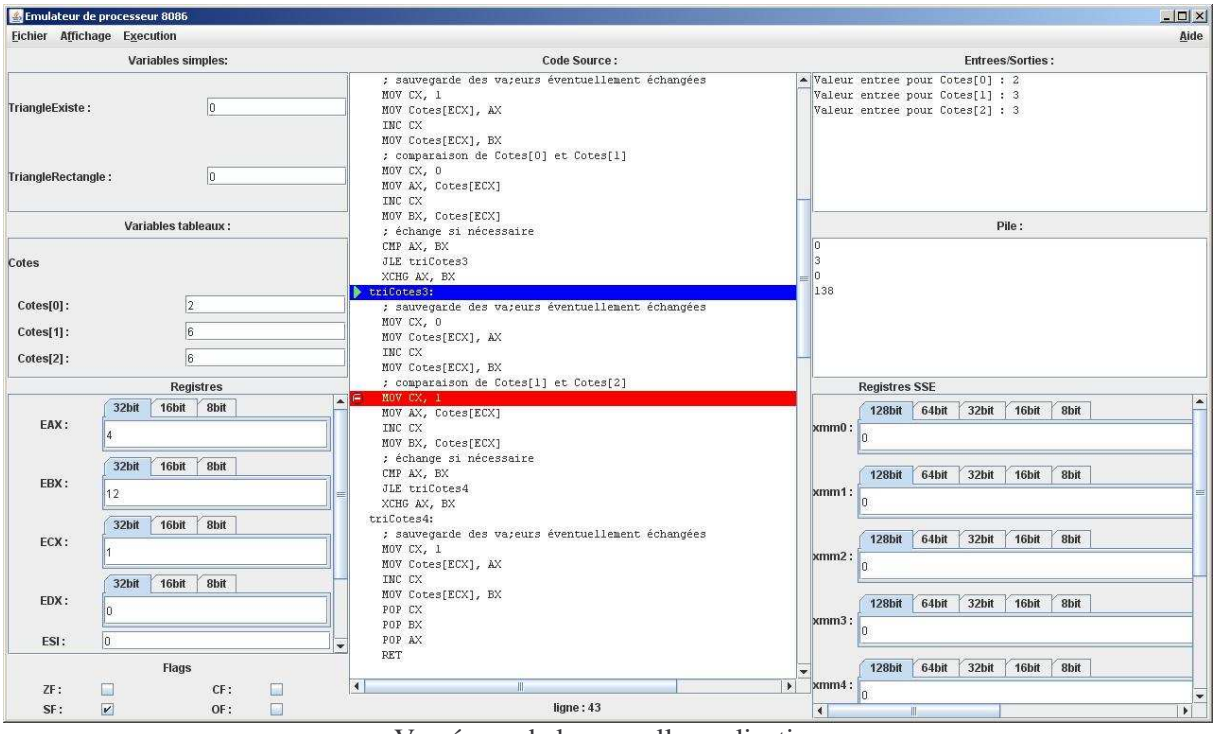
Le but de notre stage était donc d'apporter des modifications et des améliorations à ce logiciel. Nous allons vous présenter les différents points sur lesquels nous avons effectué des changements, que ce soit au niveau de l'interface ou au niveau du cœur de l'application.

1 - Interface

La première modification, la plus visible au premier abord, est l'amélioration de l'interface graphique. Comme nous l'avons vu dans la partie « Analyse de l'existant », le logiciel original se composait de 7 fenêtres séparées. Au lancement du programme, cela ne gênait en rien : les fenêtres étaient automatiquement placées d'une façon plutôt fonctionnelle. Cependant, cette indépendance des fenêtres se révèle vite être un défaut à l'utilisation. Dès que l'on commence à les déplacer, on se retrouve vite avec une interface peu pratique et des fenêtres qui se superposent.
C'est pourquoi nous avons commencé par modifier cette interface pour rassembler toutes les fenêtres en une seule.

Nous avons donc modifié la relation d'héritage pour que *JEmu8086Dialog* hérite de *JPanel* et *Jemul8086Frame* de *JFrame*, puis ensuite modifié en conséquence tous les appels de méthodes spécifiques à *JPanel*. Ce changement semble au final assez logique, les deux étant des conteneurs.

Cette première modification effectuée, nous avons dû ajouter un gestionnaire de mise en page (GridBagLayout) à la fenêtre principale (représentée par la classe *TFenPrinc*) afin d'y ajouter nos nouveaux éléments de façon à respecter au mieux une disposition proche de ce que proposait l'application originale.



Vue écran de la nouvelle application

2 - Gestion du 32 bits

2.1 – Passage en 32 bits

Une autre grande modification apportée à ce logiciel est la gestion du 32 bits, mais aussi des parties hautes et basses des nouveaux registres généraux. Pour obtenir ce résultat nous avons commencé par faire une simple modification pour passer du 16 bits uniquement au 32 bits uniquement.
Cela a consisté principalement à changer le nom des registres aux différents endroits dans le logiciel (AX devient EAX) ainsi que modifier la plage de nombres acceptés (0 à 2^{32}-1 au lieu de 0 à 2^{16}-1). Nous avons ici été confrontés à une première difficulté liée à la gestion des types dans le langage JAVA. Pour le moment, chaque valeur était stockée dans un type **int**. En Java, contrairement à certains langages comme le C, les types sont de tailles fixes, peut importe la machine sur laquelle le programme est exécuté. En l'occurrence, le type **int** de java est codé sur 32 bits dont un bit de signe. Nous ne pouvons donc pas stocker des nombres entiers positifs de 32 bits, il nous a fallu utiliser le type **long**, et faire les modifications nécessaires dans les différentes classes.

Cependant cette modification n'est pas sans conséquences, en effet, nous utilisons un vecteur (classe *Vector*) pour représenter un tableau. La méthode Java qui permet d'accéder à un élément d'un vecteur ne peut pas prendre de **long** en paramètre, nous sommes donc contraints de faire un cast en **int**, ce qui interdit d'avoir des tableaux de taille supérieure à la valeur maximale d'un **int** en Java. Cela dit, il est rare de manipuler des tableaux de cette taille dans un programme de base en assembleur.

2.2 – La gestion des parties hautes et basses des registres

La seconde partie du travail, la plus importante en terme de fonctionnalité des registres 32 bits, a consisté à pouvoir gérer séparément les parties hautes et basses des registres généraux ce qui permet d'ajouter de l'intérêt au logiciel en s'approchant un peu plus de l'utilisation réelle du langage assembleur.

Pour cela nous avons choisi de ne pas modifier le diagramme de classe établi par les précédents développeurs, qui nous semblait être adapté tel qu'il était pour l'ajout de cette fonctionnalité. Nous avons donc ajouté et modifié les méthodes impliquées dans la traduction de la partie DATA en zones mémoires et de la traduction du programme en instruction lisibles par l'émulateur. Lors de l'analyse, on accepte maintenant les mots clés DD (double mot : 32 bits) et DB (byte : 8 bits) en plus de DW (mot : 16 bits). Cette modification s'applique aussi bien aux variables classiques qu'aux tableaux, auquel cas cela indique la taille d'un élément du tableau.

Ensuite, il a fallu continuer nos modifications dans ce sens pour permettre au programme de comprendre les instructions impliquant des parties hautes et basses de registres. (Exemple MOV var, AX). Pour cela, nous avons créé une méthode qui renvoie le nom du registre 32 bit, s'il existe, dont nous prenons la sous-partie. Ainsi le logiciel accepte un programme assembleur contenant ce genre de nom de registre : nous pouvons maintenant utiliser, par exemple pour EAX, la partie basse de 16 bits AX, mais aussi les partie AH et AL qui sont les parties hautes et basses de AX.

Nous avons en parallèle modifié les classes *Variable* et *Tableau* ainsi que les classes *ParamVariable* et *ParamTableau*, pour pouvoir prendre en compte ce nouvel aspect dans le traitement des instructions par la classe CPU.
Nous avons dû de la même manière modifier les méthodes set et get en ce qui concerne la modification de valeurs en mémoire pour bien modifier la bonne portion de mémoire. En effet, bien que le nom EAX, AH, AH et AL agissent au final sur un seul et même registre, il est maintenant nécessaire de filtrer de façon précise quel élément va être modifié en mémoire.

Maintenant la traduction de programme en instruction exécutable se charge de déterminer la validité des paramètres pour une instruction. Ensuite lors de l'appel à une méthode get, on filtre la valeur de la variable ou du registre concerné en fonction de sa taille à l'aide d'un "et" logique (bit à bit) avec une valeur dépendant de la taille. Puis, lors de l'appel à une méthode set, on modifie uniquement la partie concernée en commençant par récupérer la valeur actuelle et en utilisant un procédé similaire à la méthode get.

Suite à ces modifications, nous avons dû adapter le traitement des instructions par le CPU en fonction des tailles des paramètres.
Ces modifications ont été l'occasion d'ajouter deux nouvelles instructions: SHR et SHL.

De cette façon l'émulateur gère maintenant les registres, variables et tableaux de doubles mots, de mots et de bytes.

Un exemple de programme accepté suite aux modifications:

```
; exemple pour les registres 32 bit
; définition des données
DATA
 Var1 DB  0
 Var2 DW 0
 Var3 DD  5
 Tab1 DB[5]
 Tab2 DW[8]
 Tab3 DD[2]
; début des instructions
CODE
start:
;exemple PUSH/POP
 MOV AH, 1
 MOV AL,  1
 PUSH AX
 MOV BX, 35
 PUSH BX
 POP ECX
;exemple MOV
 MOV EDX,2
etiquette:
 SHL EAX,3
 MOV ECX,0
 MOV Tab1[CL],EAX
 DEC EDX
 ECRIRE AX
 CMP EDX,0
 JNE etiquette
```

3 - SSE

3.1 – Le SSE en quelques mots

L'une des modifications importantes que nous avons apportée au programme est la gestion de quelques instructions SSE (Streaming SIMD Extensions)
« SSE est un jeu de 70 instructions supplémentaires pour microprocesseurs x86, apparu en 1999 sur le Pentium III en réponse au 3DNow! d'AMD apparu 1 an plus tôt. » [1]
L'intérêt principal de ces instructions est le fonctionnement de type SIMD (*Single Instruction on Multiple Data*). Cela signifie que la même instruction est appliquée simultanément à plusieurs données pour produire plusieurs résultats.

Le SSE a donc ajouté huit nouveaux registres de XMM0 à XMM7. Ce sont des registres 128 bits. Ils peuvent donc compacter ensemble par exemple 4 nombres flottants 32 bits simple précision (en utilisant le standard IEEE 754).

Apparu avec le Pentium 4 en 2001, le jeu d'instructions SSE2 apporte des avancées significatives : il ajoute des instructions flottantes double précision (64 bits) mais surtout, il étend les instructions MMX qui utilisaient des registres 64 bits aux registres XMM (128 bits).

3.2 – Notre représentation

L'un de nos buts lors du développement du projet était de réussir à insérer quelques unes de ces instructions SSE dans le programme. Cela nous obligeait à compléter le schéma des classes existant pour gérer des registres 128 bits. La difficulté étant de choisir comment représenter 128 bits sachant que le type le plus grand en java est le type « long » sur 64.

Pour contourner cela, nous avons choisi de représenter un registre 128 bits par un tableau de 16 cases de 8 bits chacun. Mais plutôt qu'un tableau de **byte**, nous définissons un tableau de **short.** En effet, en java, les types simples sont signés et pour pouvoir travailler sur des entiers signés, nous avons dû contourner le problème. Avec un octet on peut stocker 2^8 valeurs soit de 0 à 255 si elles sont non-signées mais de -128 à 127 dans le cas contraire. Ce problème inhérent à Java (où les types simples sont obligatoirement signés) fait perdre en clarté et complique le code, mais se contourne donc facilement.

3.3 - Les classes modélisant les données 128 bits.

A l'image des données 32 bits, plusieurs classes seront utiles pour travailler avec des données de 128 bits : la classe mère : *Donnee128* comporte comme champ le tableau de 16 cases de **short**. Héritant de cette classe, *Etiquette128* ajoute simplement un champ **nom** (String) à la donnée.
Enfin la classe *Variable128*, héritant de *Etiquette128*, comporte un champ **sorte** qui permet de caractériser la donnée : elle peut être un registre ou bien un emplacement mémoire (en réalité un tableau, selon le principe de notre émulateur.)

Maintenant que nous avons la structure de base pour traiter nos registres 128 bits, nous allons nous pencher sur les différentes instructions SSE que nous avons ajoutées.

[1] Définition Wikipedia, Article "Streaming SIMD Extensions"

3.4 – Les instructions

L'instruction MOVDQU

Instruction	Description
MOVDQU *xmm1*, *xmm2/m128*	Move unaligned double quadword from *xmm2/m128* to *xmm1*.
MOVDQU *xmm2/m128*, *xmm1*	Move unaligned double quadword from *xmm1* to *xmm2/m128*.

Tab1 : Description de l'instruction MOVDQU [2]

L'instruction MOVDQU (Move Unaligned Double Quadword) est comparable à l'instruction MOV du x86. Elle permet de transférer un *Double Quad*, c'est-à-dire 128 bits soit d'un registre SSE vers un autre soit d'un emplacement mémoire vers un registre SSE (ou vice-versa).

Les instructions *PADDB / PADDW / PADDD*

Instruction	Description
PADDB *xmm1,xmm2/m128*	Add packed byte integers from *xmm2/m128* and *xmm1*.
PADDW *xmm1, xmm2/m128*	Add packed word integers from *xmm2/m128* and *xmm1*.
PADDD *xmm1, xmm2/m128*	Add packed doubleword integers from *xmm2/m128* and *xmm1*

Tab 2 : Description des instructions PADDB / PADDW / PADDD[3]

Les trois instructions PADDB / PADDW / PADDD sont très proches : elles effectuent des additions d'entiers par paquets : c'est l'application du principe SIMD *(Single Instruction on Multiple Data).* L'instruction PADDB additionne les données de 128 bits par paquets de byte. En cas d'overflow, c'est-à-dire si la somme est trop grande pour être représentée sur 8 bits, seuls les 8 bits les plus faibles sont écrits dans l'opérande de destination, les retenues sont ignorées. De même, avec l'instruction PADDW, l'addition est faite par paquets de mots (16 bits) et avec l'instruction PADDD, l'addition est faite par paquets de doubles mots (32 bits).

Pour implémenter ces opérations, on créé des accesseurs getByte, getWord, getDouble dans la classe Donnee128. Chaque accesseur a comme paramètre un entier qui désigne sa place. Voici par exemple getWord (dans la classe Donnee128)

```
//retourne le mot n° n (entre 0 et 7)
int getWord(int n){
      return (int) ((reg[2*n+1]+256*reg[2*n])& 0x0ffff);
}
```

[2] D'après la documentation : http://www.ews.uiuc.edu/~cjiang/reference/vc184.htm
[3] D'après la documentation : http://www.ews.uiuc.edu/~cjiang/reference/vc223.htm

Ainsi, dans la classe CPU.java, on retrouve les opérations permettant d'effectuer l'instruction PADDW :

```
for (int i=0; i<8; i++){
      sum = (int) (d128.getWord(i)+d128b.getWord(i));
      d128.reg[2*i]= (short) (sum/256 & 0xff);
      d128.reg[2*i+1]= (short) (sum & 0xff);
}
```

- d128b est la donnée 128 bits source
- d128 est la donnée 128 bits de destination
- sum est une variable temporaire stockant la somme des deux mots.

Il suffit ensuite d'affecter le résultat obtenu dans le registre 128 bits de destination, mot par mot.

3.5 – Exemple

Voici un exemple pour illustrer le fonctionnement de la partie SSE :

```
DATA
   T1   DD[4]
   T2   DD[4]
   T3   DW[8]
   T4   DW[8]
CODE
   MOV ECX, 0
   MOV EAX, 1
   MOV EBX, 65535
loop:
   MOV T1[ECX], EAX
   MOV T2[ECX], EBX
   MOV T3[ECX], EAX
   MOV T4[ECX], EBX
   INC ECX
   INC EAX
   CMP ECX, 4
   JL loop

   MOV ECX, 0
   MOVDQU xmm0, T1[ECX]
   MOVDQU xmm1, T2[ECX]
   PADDD xmm0, xmm1
   MOVDQU xmm2, T3[ECX]
   MOVDQU xmm3, T4[ECX]
   PADDW xmm2, xmm3
   MOVDQU T1[ECX], xmm0
   MOVDQU T3[ECX], xmm2
```

Ce programme illustre l'utilisation des instructions SSE. Les deux premières instructions MOVDQU déplacent 128 bits d'un tableau vers un registre. Les instructions PADDD et PADDW vont effectuer des additions en parallèle : par paquets de 16 bits pour l'instruction PADDW et par paquet de 32 bits pour l'instruction PADDD.

3.6 - *Conclusion de la partie SSE*

Intégrer les instructions SSE à notre programme était un défi intéressant : en effet, tout le code fonctionnait avec les registres traditionnels et il fallait pouvoir adapter les méthodes pour qu'elles agissent aussi sur ces nouveaux registres. Outre une bonne connaissance et une bonne compréhension du code, cela nécessitait également de bien se documenter sur le fonctionnement du SSE. Lors de cette partie, la première difficulté était de trouver comment représenter un registre 128 bits, en intégrant la particularité de Java pour lequel les types simples sont nécessairement signés. Ensuite, il a fallu adapter chacune des méthodes qui fonctionnaient pour les registres 32 bits : la récupération des paramètres, les accesseurs pour les données 128 bits et les utiliser pour chaque opération.
Bien sûr la liste des instructions SSE ajoutées n'est pas exhaustive ! Il s'agissait simplement d'illustrer les modifications que nous avions apportées au programme initial pour la prise en charge des registres 128 bits et des instructions SSE. Ainsi, il est très rapide maintenant que ce travail est fait d'ajouter de nouvelles instructions. Cela pourrait être par exemple le travail d'autres étudiants à l'avenir. Il nous semblait important de ne pas imposer à d'éventuels successeurs les mêmes difficultés de compréhension que nous avions subies, par manque de commentaires ou d'organisation des classes.

4 - Gestion des erreurs

Avec le logiciel original, quand nous essayions de charger un programme qui comportait une erreur de syntaxe, le logiciel qui rencontrait donc une erreur à la traduction de ce programme semblait ne rien charger, et c'était dans la console que se trouvait la réponse à cette erreur. Nous avons donc pris le temps de gérer les exceptions provoquées par ces erreurs. Désormais, on obtient un message d'erreur lors du chargement d'un fichier qui comporte des erreurs. L'utilisateur sait maintenant le numéro de la ligne de la première erreur ainsi que son type quand il est connu (erreur de paramètre pour une instruction, variable non déclarée…)

Cette nouvelle fonctionnalité s'est avérée être très pratique pour nos propres tests et nous pensons qu'elle complète le logiciel d'une façon intéressante. En plus de voir l'effet de programme tout fait sur le processeur, l'utilisateur débutant en assembleur peut plus facilement apporter des modifications ou même construire son propre programme en corrigeant aisément ses fautes.

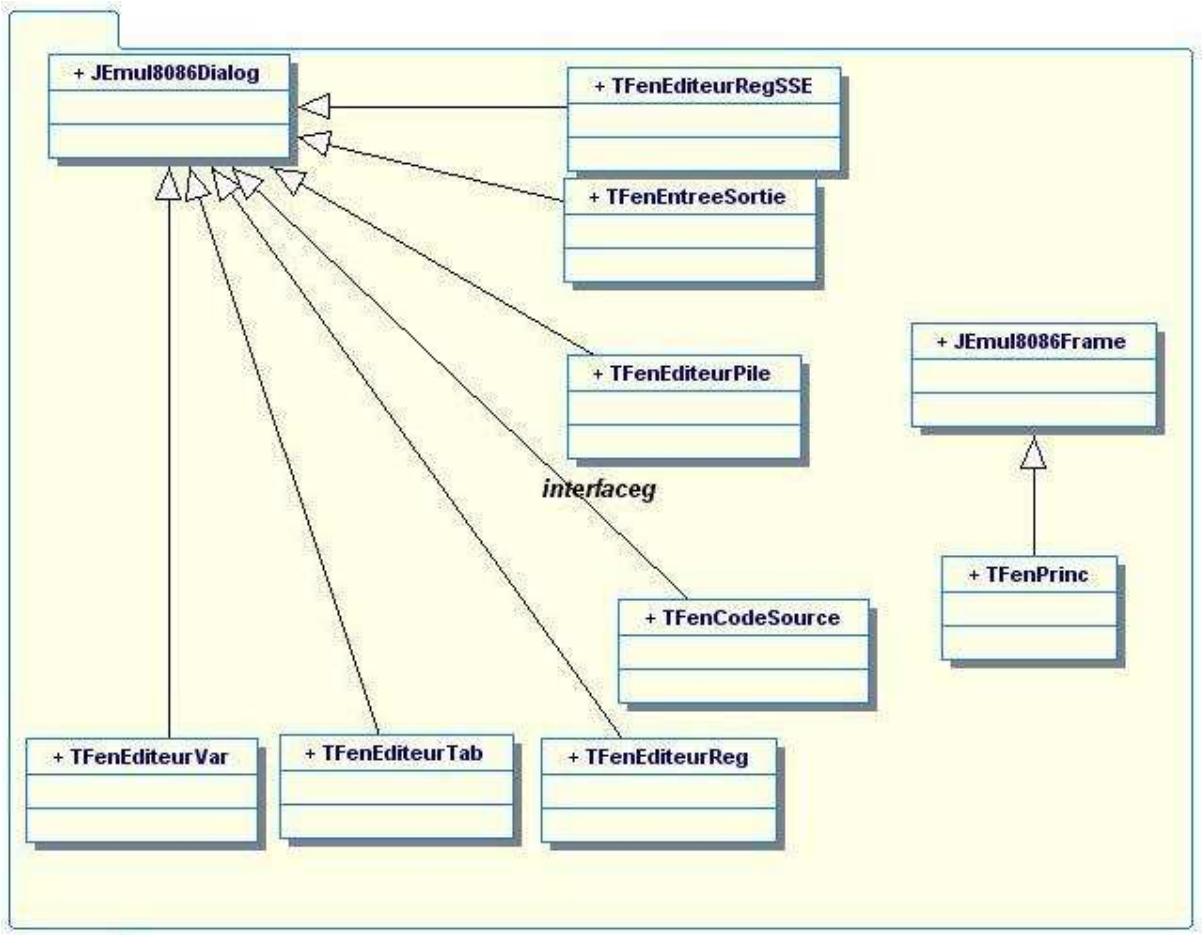
Le programme affiche donc dans une fenêtre d'avertissement la première erreur qu'il rencontre. Le créateur du programme peut donc être amené à effectuer ses corrections.

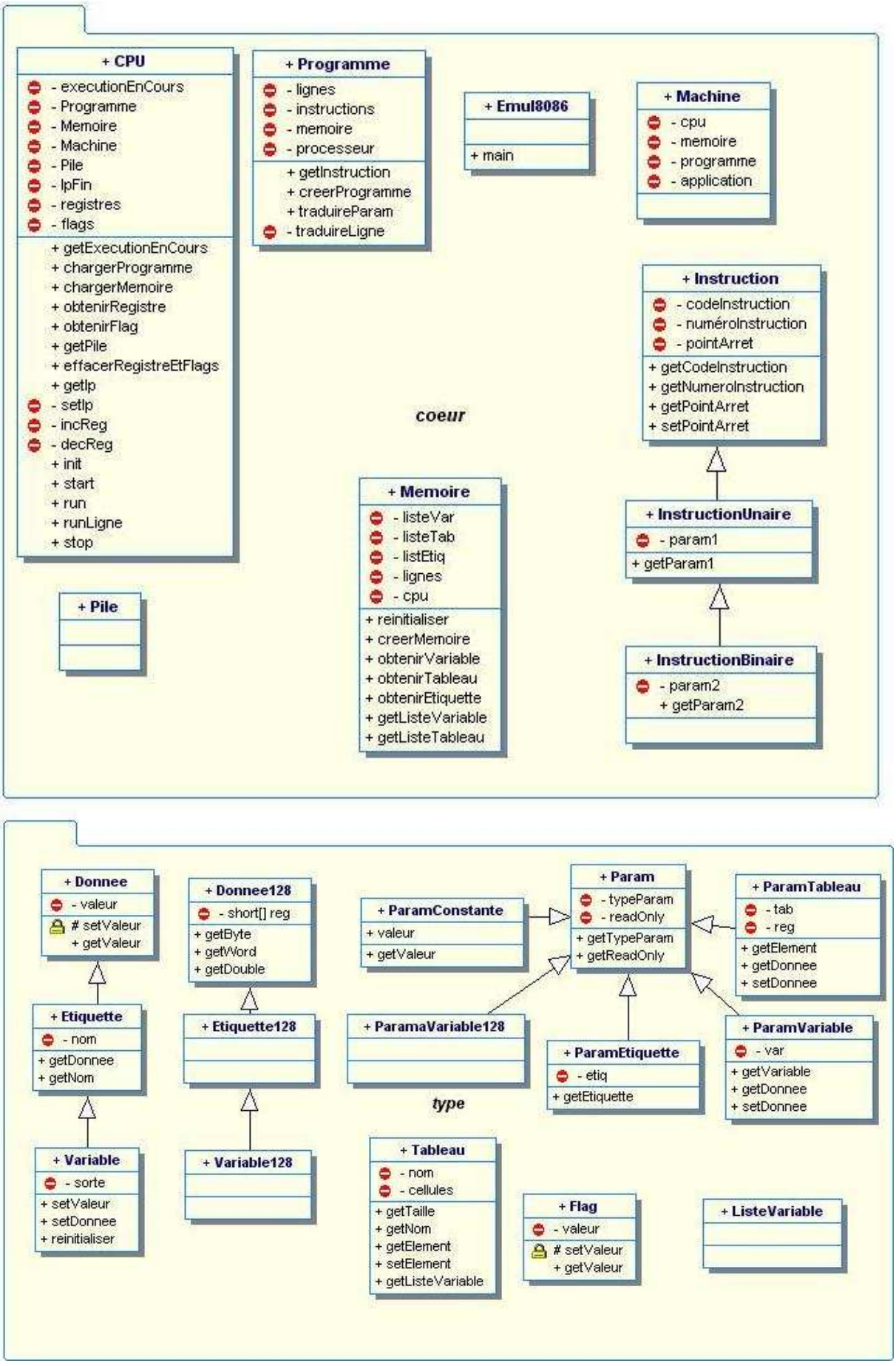
En revanche, lors de la création du programme, on ne peut pas déterminer si le programme va accéder à une case hors limites du tableau, ce qui est cohérent avec le fonctionnement d'un débuggeur normal.

5 – Le nouveau diagramme de classes

Nous avons commenté le code et nous avons également tenu à le réorganiser avec les possibilités qu'offre le langage. Nous avons donc créé 3 paquetages et réparti les classes. Ces modifications font suite à la difficulté que nous avons eue à bien comprendre le code original, très peu commenté.

Voici le nouveau diagramme de classes:




+ CPU
- executionEnCours
- Programme
- Memoire
- Machine
- Pile
- Ip Fin
- registres
- flags
+ getExecutionEnCours
+ chargerProgramme
+ chargerMemoire
+ obtenirRegistre
+ obtenirFlag
+ getPile
+ effacerRegistreEtFlags
+ getIp
- setIp
- incReg
- decReg
+ init
+ start
+ run
+ runLigne
+ stop
+ Programme
- lignes
- instructions
- memoire
- processeur
+ getInstruction
+ creerProgramme
+ traduireParam
- traduireLigne
+ Emul8086
+ main
+ Machine
- cpu
- memoire
- programme
- application
+ Instruction
- codeInstruction
- numéroInstruction
- pointArret
+ getCodeInstruction
+ getNumeroInstruction
+ getPointArret
+ setPointArret
coeur
+ Memoire
- listeVar
- listeTab
- listEtiq
- lignes
- cpu
+ reinitialiser
+ creerMemoire
+ obtenirVariable
+ obtenirTableau
+ obtenirEtiquette
+ getListeVariable
+ getListeTableau
+ InstructionUnaire
- param1
+ getParam1
+ InstructionBinaire
- param2
+ getParam2
+ Pile

+ Donnee
- valeur
setValeur
+ getValeur
+ Donnee128
- short[] reg
+ getByte
+ getWord
+ getDouble
+ ParamConstante
+ valeur
+ getValeur
+ Param
- typeParam
- readOnly
+ getTypeParam
+ getReadOnly
+ ParamTableau
- tab
- reg
+ getElement
+ getDonnee
+ setDonnee
+ Etiquette
- nom
+ getDonnee
+ getNom
+ Etiquette128
+ ParamaVariable128
+ ParamEtiquette
- etiq
+ getEtiquette
+ ParamVariable
- var
+ getVariable
+ getDonnee
+ setDonnee
type
+ Variable
- sorte
+ setValeur
+ setDonnee
+ reinitialiser
+ Variable128
+ Tableau
- nom
- cellules
+ getTaille
+ getNom
+ getElement
+ setElement
+ getListeVariable
+ Flag
- valeur
setValeur
+ getValeur
+ ListeVariable

IV – Conclusion

Pour nous, ce projet a été l'occasion de retravailler un projet existant, déjà fonctionnel et plutôt bien construit. C'est la première fois dans le cursus licence que nous avons travaillé de cette manière. Reprendre un code existant comporte des avantages et des inconvénients :

D'un côté, nous nous trouvons face à des difficultés nouvelles. Il faut en effet comprendre la manière dont le code a été pensé alors que nous ne disposions que de très peu de documentation. L'absence totale de commentaires, l'utilisation du seul paquetage par défaut nous ont montré qu'un projet pouvait être très clair dans l'esprit de ses auteurs mais que ceux qui reprennent le code peuvent se retrouver face à un véritable obstacle.

Mais d'un autre côté, quand la barrière de la compréhension a été franchie, nous pouvons aller plus loin puisque le programme de base est déjà avancé et nous n'avons donc pas besoin de reprendre le sujet au début.

Finalement, ce stage a été très bénéfique pour nous. En effet il mêlait ensemble plusieurs disciplines et nous a permis de mettre à profit les études des semestres précédents et de parfaire nos connaissances dans des matières déjà étudiées au premier semestre :
En effet, les acquis du cours de programmation objet étaient sans cesse sollicités et ce nouveau développement de projet en java nous a encore permis d'aller plus loin dans les possibilités du langage.

En outre, notre sujet était parfaitement dans la continuité du cours d'architecture des ordinateurs et d'initiation à l'assembleur. Il nous a donc permis de bien mieux comprendre le fonctionnement des instructions, des flags ou des registres. Pour nous, il est certain que ce projet de « longue durée » a été bénéfique : nous n'avons pas souvent eu l'occasion de nous plonger dans un projet de cette façon. Et passer du temps sur un sujet forme notre esprit et notre intelligence à raisonner un peu plus comme des informaticiens !

Enfin, un des avantages majeurs de ce projet a été le travail en binôme. Ce n'est bien sûr pas un travail en grande équipe mais nous apprend déjà à bien répartir le travail, à se concerter régulièrement pour organiser le travail et à s'enrichir mutuellement en partageant des idées. Cela a été très formateur tout au long de ce mois de travail.

Nous voyons bien sûr qu'il reste des améliorations à apporter et des fonctionnalités à ajouter. D'un point de vue esthétique, la coloration syntaxique du code aurait sans doute été utile. De plus, pour être vraiment exhaustif, notre programme doit pouvoir accepter encore de nouvelles instructions pour se rapprocher au mieux du langage assembleur des processeurs actuels. Mais la vitesse d'évolution des caractéristiques des processeurs promet d'occuper de nombreuses générations d'étudiants en informatique !

Bibliographie / Sitographie

- Programmer en JAVA : 5e édition java 5 et 6 / Claude Delannoy. – Editions EYROLLES, 2007 – 800 pages.
  ISBN : 978-2-212-12232-9

- Développons en Java version 0.95 / Jean-Michel DOUDOUX - 18/11/2007 -
  Disponible sur son site: http://www.jmdoudoux.fr/accueil_java.htm

- http://benoit-m.developpez.com/assembleur/tutoriel/CoursASM_Fichiers/sommaire.php
  Club d'entraide des développeurs francophones
  Benoit-M., 03 janvier 2003
  Tutorial sur l'assembleur

- http://java.sun.com/javase/6/docs/api/
  Sun – Documentation java 6

- http://en.wikipedia.org/wiki/X86_instruction_listings
  Article wikipédia X86

- http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
  Article wikipédia SSE

- http://www.ews.uiuc.edu/~cjiang/reference/
  Documents de référence sur les instructions SSE