Les choix de programmation

- Représentation des mémoires
  1. Mémoire de commande
  2. Mémoire principale
- Simulation des registres
- Simulation des circuits
- Programme principal

Lorsque on utilise la programmation pour résoudre le problème de simulation de Mic-I on remarque trois types de choix selon la représentation de la micro-instruction :

- Par un tableau de 13 élément.
- Par deux entiers de 16 bits (bit à bit).
- Par une structure de données qui enveloppe 13 champs.

Bien qu'on utilise dans le TP le 2^{ième} choix ; on ne s'intéresse que par la représentation bit à bit.

Représentation des mémoires

Mémoire centrale (principale) : On la représente par un tableau des entiers ; La mémoire principale sera déclarée (*int mp [48] ;*), partagée en trois parties : segment code (0-15), segment donnée (16-31) et segment pile (32-47) ;

Mémoire de command : on la représente par un tableau de deux entiers : La mémoire de command sera déclarée (*int mcom [79] [2] ;*)

Simulation des registres

Les registres sont déclarés comme étant des entiers : les registres généraux (*int rg[16] ;*) et tous les autres registres sont déclarées dans une procédure (*mic*) qui est incluse dans l'unité <*k_mic.h*>.

Simulation des circuits

operation: qui fait les opérations de UAL et de décalage <*k_fct.h*>
*void operation(int *ual,int *n,int *z,int ek,int b,int i,int j)*

ck1: permet de charger la micro-instruction de MCOM à RMI <*k_ck.h*>
void ck1(int mco,int rmi[],int mcom[79][2])

ck2: permet de charger le contenue des bus A et B <*k_ck.h*>
*void ck2(int *a,int *b,int rmi[],int rg[])*

ck3: permet de charger le RAD <*k_ck.h*>
*void ck3(int *rad,int b,int rmi[])*

ck4: permet de manipuler UAL,DECAL,RDO,Registres,MCO <*k_ck.h*>
*void ck4(int *mco,int *rdo,int rad,int rg[],int rmi[],int mp[],int a,int b)*

Programme principal

Le vrai programme principal qui gère le fonctionnement de (MIC - I) est dans l'unité <*k_mic.h*> et le programme ci-après c'est le plus important dans le déroulement de l'exécution d'un ensemble des instructions.

```
while ((compteur<16)&&(mp[compteur]!=-1)) //-1=0xffff;
{
ck1(mco,rmi,mcom);
ck2(&a,&b,rmi,rg);
ck3(&rad,b,rmi);
ck4(&mco,&rdo,rad,rg,rmi,mp,a,b);
cycle++;
if(mco==0)
{
affich_etat(rg,mp,cycle);
compteur=rg[0];
}
}
```

Il est préférable qu'on mette l'instruction (*fin*) qui a le code (*ffff*) qui est (-1) en décimal pour sortir de la boucle dans un programme incluse des sous programmes.

Et dans un programme on peut négliger cette étape puisque la mémoire est initialisée par le code de sortir, et aussi on peut la négliger lorsqu'on utilise tout le segment de code dans ce cas le compteur ordinal va vers l'adresse 16.

Les fonctionnalités implémentés

- L'unité « K_FCT.H »
- L'unité « K_CK.H »
- L'unité « K_MIC.H »
- L'unité « K_CODE.H »
- L'unité « K_GRAPH.H »
- L'unité « K_SOURIS.H »

Dans ce petit simulateur de (MIC-I) on utilise plusieurs fonctionnalités graphiques, fonctions de vérification et le plus importants de tous ça les fonctions de la machine étudiée. Et ici nous allons étudier les unités qu'on a utilisées dans ce petit simulateur.

L'unité « K_FCT.H »

Cette unité contient les quatre fonctions suivantes :

operation: qui fait les opérations de UAL et de décalage.
*void operation(int *ual,int *n,int *z,int ek,int b,int i,int j)*

init_rg : qui fait les initialisations des registres.
void init_rg(int rg[],int n)

init_mem : qui initialise la mémoire de commande
void init_mem(int mcom[79][2])

init_mp : qui initialise la mémoire principale
void init_mp(int mp[])

L'unité « K_CK.H »

Cette unité contient les fonctions suivantes *ck1*, *ck2*, *ck3*, *ck4* qu'on a déjà démontré dans la simulation des circuits au chapitre précédent. Elle utilise la fonction (*operation*) de l'unité précédente au niveau de *ck4*.

L'unité « K_MIC.H »

Elle contient seulement une fonction *mic* qu'on a déjà démontré dans le chapitre précédent sous titre de « programme principal ». « *void mic(int mp[])* »

L'unité « K_CODE.H »

Cette unité sert pour le décodage d'instruction et pour détecter les erreurs de dépassement d'adresse de segment. Elle contient deux fonctions :

Convert : sert pour convertir les codes mnémoniques et les données à un entier mais lorsque il est acceptable c à d il n'y a aucun défaut d'adressage, et pour cette raison nous utilisons un variable de type caractère renvois **'o'** si le code est accepté.
'n' : si la données est assez grande (0-1096) ;elle sert avec LOCO.
'd' : si l'adresse de donnée incorrecte est hors segment (16-31) sert avec les instructions qu'ils utilisent des adresses données comme LODD.
'x' : si l'adresse hors mémoire sert avec les instructions qu'ils adressent la mémoire comme ceci (m[pp+x]) donc il faut empiler (x+1) fois avant de les utilisées.
'1' : lorsque on veut dépiler avant qu'on empile ou on veut dépiler par un nombre des instructions supérieur à celle des instructions d'empilement.
'i' : sert avec les instructions de branchement lorsque les adresses hors segment d'instruction(0-15).
'm' : s'il y a un défaut dans le code mnémonique.
'f' : pour dire qu'on a tapé sur (*fin*) alors le code est (*ffff*).
*void convert(char scode[],int *code,char *oui,int *pp)*

L'unité « K_GRAPH.H »

C'est l'unité qui gère les fonctions graphiques et d'affichage.

lire1 : c'est la fonction qui lit une seule instruction sous forme (*xxxx 9999*) ; où les *x* sont des alphabets et les *9* sont des chiffres. Elle lit caractère par caractère et affiche au même temps sur l'écran.
void lire1(char kaka[],int y0)

lire2 : pour lire les données sous forme hexadécimal.
void lire2(char kaka[],int y0)

affich_nb : sert pour afficher les nombres en hexadécimal ses nombres sont les valeurs de segment de donnée et les valeurs des registres modifiables.
void affich_nb(int k,int x0,int y0)

affich_cycle: sert pour afficher le nombre des cycles processeur.
void affich_cycle(int k)

affich_msg : sert pour afficher les messages d'erreur de lecture des codes.
*void affich_msg(char *h)*

init_graph : sert pour initialiser le mode graphique.
void init_graph()

init_screen : pour initialiser l'affichage.
void init_screen()

lecture : lecture de tous les instructions et les données; elle utilise les deux fonctions de lecture précédents.
void lecture(int mp[])

affich_etat : sert pour afficher les résultats des registres et segment donnée. Elle utilise *affich_nb* pour afficher les nombres.
void affich_etat(int rg[],int mp[],int cycle)

init_boite : sert pour initialiser la boite de dialoge.
void init_boite()

bouton : sert pour manipuler les deux boutons de dialogue.
*void bouton(int *e)*

fin_bout : gérer la boite de dialogue de fin programme.
void fin_bout()

about : c'est l'écran d'information (à propos de …).
void about()

L'unité « K_SOURIS.H »

Elle utilise l'interruption **33h.**

init_souris : fonction qui détermine la position initiale de la souris :la limite horizontale (fonction **7h**) et verticale (fonction **8h**).
void init_souris(int x,int y)

affich_souris : sert pour afficher la souris (fonction **1h**).
void affich_souris()

masq_souris : sert pour masquer la souris (fonction **2h**).
void masq_souris()

etat_souris : sert pour donner l'état de la souris (fonction **3h**).
*void etat_souris(int *x,int *y,int *b)*

Le mode d'emplois

- Lecture des instructions
- Lecture des données
- Les étapes de l'exécution

Lorsque nous utilisons ce simulateur on doit d'abord remplir le code après on remplis les données dans le segment données, et l'exécutions va dérouler pas à pas ; puisque on ne peut pas interrompre l'exécution où il y a une boucle infinie.

Lecture des instructions

La lecture se fait par le tapage des caractères sous forme (xxxx 9999) ; les quatre premiers caractères sont des alphabétiques même si on frappe les autres touches, ils ne fonctionnent pas et il fait un saut sur un blanc et les quatre derniers caractères sont des numéros.

Il n'y a pas de curseur donc on ne peut pas savoir notre position sauf si nous taperons un caractère.

Les boutons SHIFT et ENTER sont admis.

Le code de fin de programme est « fin » et pour sortir de l'insertion de code il suffit de taper « EXIT » et on part vers l'insertion des données.

On peut utiliser le majuscule ou le minuscule ou les deux.

Lecture des données

La lecture se fait par hexadécimal ; et pour sortir de l'insertion il suffit de frapper sur ENTER sans insertion.

EXEMPLE :

```
05 ENTER   ; on peut utiliser la touche ENTER.
FFAD       ; ici si on remplis toute la case on va sauter directement vers la suivante.
ENTER      ; ici on sort de l'insertion puisque on met un vide.
```

Les étapes de l'exécution

Cliquer sur le bouton « **continue** » pour continuer l'exécution jusqu'à le dialogue de fin ou interrompre l'exécution avec « **quitter** »

Les performances

- Lecture plus protégée
- Mémoire segmentée
- Affichage très clair
- Simulation des composantes

Comme étant un logiciel, ce simulateur à des performances. De ces derniers on remarque des performances où moment de la lecture, des performances de la simulation de segmentation mémoire, des performances de l'affichage qui est très clair pour l'utilisateur et plus de ça on a la performance de la simulation de MIC-I.

Lecture plus protégée

A cause de la fonction de gestion des codes « **Convert** » on ne peut pas dépasser les adresses des segments, puisqu'il y a des contraintes telles que :

- Le segment de code est entre 16 et 31.
- Le segment de données est entre 0 et 15.
- On ne peut pas dépiler qu'on a déjà empilé.

Mémoire segmentée

La mémoire est partagée en trois parties comme il est noté au dessus.

Affichage très clair

Ce simulateur muni d'un affichage très clair pour l'utilisateur qui sait les instructions concernant cette machine.

Simulation des composantes

Simulation des composantes de MIC-I telles que les « clocks », l'UAL…etc. Et simulation des fonctionnalités qu'ils sont en réalité des composants électroniques ; malgré ça la simulation reste insuffisante, mais on peut la considérer comme étant un test de cette machine.

Les défauts

- Perd de temps
- Perd de l'espace mémoire
- Complexité de l'utilisation
- La limitation de l'utilisateur

On essaie avec les instructions 'LODD, STOD, ADDD, SUBD, LOCO, ADDL, CALL, PSHI, POPI, PUSH, POP, RETN' et tout va bien avec ces instruction, mais comme on a dit il n'y a pas une chose idéal, donc on peut trouver plusieurs défauts dans ce simulateur. Le problème ici est si on a détecté des erreurs durant la programmation on a obligé de les corriger. Même où il y a des fautes, on essaie de les minimiser. Et pour cette raison les défauts décrits ici sont des défauts généralement sur le temps et l'espace mémoire perdue dans ce simulateur.

Perd de temps

Puisque ce simulateur est riche des fonctions graphiques donc on a un perdre de temps durant les fonctions de dessin, et plus de ça les interruptions par les deux fonctions d'affichage de souris et de masquage, puisque si on dessin sur la souris la place de cette dernière n'aura pas de couleur, et pour ça on masque la souris, on dessine et on l'afficher.

Perd de l'espace mémoire

Le perd de l'espace mémoire est clair, puisque on a utilisé plusieurs fonctionnalités et la preuve de ça est que l'exécutable prend 50,9 Ko (52 158 octets).

Complexité de l'utilisation

Il y a une difficulté de l'utiliser sans lire cette documentation même avec l'affichage qui est plus clair.

La limitation de l'utilisateur

Il limite l'utilisateur puisque il ne le permet pas plusieurs choses comme le dépilement sans empilement.