Architecture des ordinateurs

Telecom Lille 1 QROC A23 Jeudi 17 février 2011
Durée 2 heures, épreuve sans documents.

NB : dans toute l'épreuve, on considère la plate-forme 8086 et on suppose que tous les pointeurs et adresses sont intra-segments (pointeur NEAR).

Aspects matériels

Résumer par une portion du schéma interne du processeur 8086 à architecture segmentée, le calcul de l'adresse physique à 20 bits mise en oeuvre par une instruction accédant à la mémoire grâce au mode d'adressage `[BX+SI+2]`. On donne les informations suivantes relatives aux valeurs des registres au moment où le calcul d'adresses a lieu.

```
AX=0010 BX=0020 CX=0030 DX=0010 SP=FFEE BP=FFEE SI=2000 DI=4000
DS=CD20 ES=CD20 SS=CD20 CS=CD20 IP=0100
```

Annoter le schéma avec les valeurs calculées à partir de ces valeurs de registres.

On considère l'instruction `MOV [BX+SI+2],AL`. Dessiner le chronogramme représentant l'activité des bus et signaux de contrôle du processeur 8 bits 8088 compatible 8086 (mêmes signaux que le 8086 à l'exception du bus de données) lors de l'exécution proprement dite de l'instruction.

On considère l'instruction `LEA SI,[BX+SI+2]` (`LEA` = Load Effective Address). Quelle valeur hexadécimale se trouve dans `SI` après exécution de cette instruction ?

Quel est l'état des signaux RD# (RD barre), WR# et M/IO# lors de l'exécution d'une sortie (instruction OUT) ?

Modes d'adressage

On a les déclarations suivantes :

```
#define a 5
int v = 0x4A2B; // adresse choisie par compilateur : 0300
```

et l'état initial des registres :

```
ax=4C3A, bx=0300, si = 300, di = 200
```

Chaque ligne qui suit est indépendante des précédentes, on ne tient compte que de l'état initial des registres et variables. Remplir les colonnes suivantes :

	Taille opérande (en bits)	Mode(s) d'adressage	Valeur après exécution	Segment mis en oeuvre
`mov ah,0`			`ax=`	
`mov ax,[bx]`			`ax=`	
`mov ax,offset v`			`ax=`	
`mov bh,[0300]`			`bx=`	
`mov byte ptr[si],a`			`v=`	
`mov al,v`			`ax=`	
`mov bx,0300`			`bx=`	
`lea si,[bx+di]`			`si=`	

Accès à des structures de données simples

On considère les déclarations suivantes :

```
#define TRUE 1
#define FALSE 0
typedef struct {
 int ch1;
 char ch2, ch3;
} rec_t;
rec_t e ;
int tab[4];
int variable1, variable2;
int *p;
```

Citez les types utilisés ou déclarés ici.

Citez les variables déclarées ici.

Représentez à l'aide de schémas de mémoire l'état des variables après exécution des instructions :

```
e.ch1 = 0;
e.ch2 = 'A';
e.ch3 = 'B';
for (variable1=0 ; variable1<4 ; variable1++)
     tab[variable1] = 0;
variable2 = TRUE;
p = &variable1;
*p = FALSE;
```

Quel est le nombre d'octets utilisés en mémoire ?

Traduire en code assembleur les instructions suivantes en utilisant les modes d'adressage appropriés.

```
// e.ch1 = 0;
```

```
// e.ch2 = 'A';
```

```
// e.ch3 = 'B';
```

```
// variable1=0;
```

```
// tab[variable1] = 0;
```

```
// variable2 = TRUE;
```

```
// p = &variable1;
```

```
// *p = FALSE;
```

Accès à des structures de données composées

On considère les structures de données suivantes :

```
#define NULL 0
int i;
typedef struct {
 char name[8];
 char *tel;
} rec_t;
rec_t *p;
rec_t tab[2];
```

On procède aux initialisations suivantes :

```
i = ...; // valeur non montrée intentionnellement mais comprise entre 0 et 1
p = &tab[i] ;
strcpy(p->name, "DURAND" ); // copie de chaîne
p->tel = (char *) malloc(11); // allocation mémoire
p->tel[0] = 0;
```

Représentez à l'aide de schémas de mémoire l'état des variables après ces initialisations.

Coder en langage d'assemblage les instructions suivantes :

```
p = &tab[i] ;
```

```
p->tel[0] = 0;
```

Structuration du code

On considère l'algorithme suivant. Coder chacune de ses étapes en langage d'assemblage.

```
// char *tab[10];
// int i;
// ...
// i = 0;
```

```
// while (i < 10) {
```

```
//   if (tab[i] != NULL) {
```

```
//        printf("%s\n", tab[i]) ;
//        tab[i] = NULL;
```

```
//   }
```

```
//   i++;
```

```
//}
```

Procédures et fonctions

On considère la déclaration d'une variable globale `i` de type `int`. Ecrire en langage C une procédure `setI` qui initialise `i` avec la valeur passée en paramètre :

```
#include <stdio.h>
int i;
```

Ecrire une fonction `getI` qui retourne la valeur de `i`.

Compléter la fonction `main` suivante de façon à :

a) utiliser la procécure `setI` pour initialiser `i` avec la valeur 0

b) afficher avec `printf` le résultat retourné par l'appel à la fonction `getI`.

```
int main() {

}
```

Passage d'arguments

On considère le programme suivant (dans ce code `sizeof(int)` vaut 2)

```
int *p;
int r;

void assign(int *p1, int i, int *p2) {
     *p2 = p1[i];
}
int main() {
     p = (int *) malloc(10 * sizeof(int));
     assign(p, 3, &r);
     return 0;
}
```

Représenter à l’aide d’un schéma mémoire l’état des variables juste avant le `return 0`.

Coder en langage assembleur, l'appel à la fonction `malloc` :

```
// p = malloc(20); (version simplifiée / la ligne réelle)
```

Coder en langage assembleur, l'appel à la fonction `assign` :

```
// assign(tab, 3, &r);
```

Accès aux paramètres d’une procédure

On se place maintenant d'un point de vue du sous-programme appelé.

```
void assign(int *p1, int i, int *p2) {
     *p2 = p1[i];
}
```

Au cours de l'exécution de la procédure, que contiennent les paramètres `p1` , `i` et `p2` exactement ?

`p1` contient …

`i` contient …

`p2` contient …

NB : en langage C, l'opérateur […] peut être appliqué indifféremment à un tableau ou à un pointeur sur un tableau avec le même effet. Dans ces conditions, que désigne les notations suivantes (parmi les variables connues du programme appelant)

`p1[i]` désigne …

`*p2` désigne …

Dessiner le contexte de pile visible par ce sous-programme. Dessiner aussi les flèches montrant les relations entre les paramètres et les autres variables juste avant l'épilogue de la procédure.

Comment peut-on désigner `p1`, `i` et `p2` à l'aide d'un adressage basé sur BP ?

```
p1:

i :

p2 :
```

Coder en assembleur la ligne suivante :

```
// *p2 = p1[i];
```