Architecture des système

Fiche Programmation assembleur x86

1 Introduction

Assembleur x86 = langage de génération 1
Avantage : exécution très rapide
Inconvénients : très bas niveau (très peu user-friendly); pas portable; beaucoup de contraintes de programmation (langage très technique)
Un assembleur par architecture de processeur : famille x86
– famille de processeurs incluant Intel Pentium, Intel i7, AMD Phenom, AMD Athlon, etc. –ex processeur 8086 : 1978
manipule 8 bits, 16 bits ou 32 bits ; syntaxe Intel (et non pas syntaxe ATetT) ; environnement : système d'exploitation Linux, logiciel nasm
Exectution Exemple:

nasm -f elf hello-world.asm	ld -m elf_i386 -o hello-world hello-world.o ./hello-world	Désassemblage : objdump -d hello-world -M intel	Déboguage : gdb hello-world

2 Registres et mémoire

Registres : variables internes au processeur , accès très rapide, mais faible nombre de registres , les registres sont spécialisés
Registres de données :
registres 32 bits : EAX, EBX, ECX, EDX : très utilisés (manipulation des données, appel de fonctions, etc.)
chacun comprend un registre 16 bits (AX, BX, CX, DX)
chaque registre 16 bits comprend deux registres 8 bits (AH, AL, BH, BL, CH, CL, DH, DL)

Mémoire : décomposée en segments (blocs) et en offsets (index)
variables globales, variables locales, allocation dynamique
pile d'appels de fonction (schéma)
Registres de segments : code : CS ; données : DS ; supplémentaires : ES, FS, GS ; pile : SS

Registre d'index : code : IP ; données : DI, SI; pile : SP, BP
Registre des indicateurs : indicateur de zéro : ZF ; indicateur de signe : SF ; indicateur d'interruptions : IF ; etc

Adressage :

adressage direct : le contenu de la case 0x100 se dit [0x100] (ou DS:[0x100])	adressage indirect : le contenu de la case dont l'adresse est la valeur de DI (ou de SI) : [DI] (très utilisé)
adressage basé : le contenu de la case dont l'adresse est la valeur de BX : [BX] (ou avec BP, mais utilise le segment SS dans ce cas) (peu utilisé)	adressage indirect basé avec déplacement : [DI+BX+4] (très peu utilisé

3 Instructions assembleur

•Affectation : MOV destination, valeur
Remarques : 'valeur' peut être une constante, un registre, une zone mémoire et une zone mémoire peut être désignée par : [constante], [registre], [registre+constante], [registre+BP+constante]
on peut préciser le segment en utilisant segment: [offset], ou [segment:offset]

•Contraintes
on ne peut pas désigner deux zones mémoire dans un MOV, il faut passer par un registre intermédiaire
on ne peut pas modifier avec un MOV certains registres (IP, registres de segments)
implicitement, DS est utilisé, sauf avec B

Instructions

Type	Nom	Remarque	contrainte
Logiques	AND destination, valeur		
	OR destination, valeur		
	NOT destination		
	XOR destination, valeur		
arithmétiques	ADD destination, valeur		
	SUB destination, valeur		
	IMUL registre1, opérande2	multiplie r1 et o2, et stocke résultat dans r1	
	IDIV opérande	divise le contenu de EDX:EAX par l'opérande, place le quotient dans EAX et le reste dans EDX	
Comparaisons	CMP a, b		
Branchements	JMP destination	(inconditionnels)	
Branchements	JE/JNE/JG/JGE/JL/JLE dest	(conditionnels)si égalité ZF=0etCF=0/si non-égalité ZF=0/si supérieur ZF=0etSF=OF/si sup et eg/ si inf /si inférieur ou égal ZF=1ouSF!=OF	
Remarques/autres	JMP segment:[offset]		
Boucle	LOOP label	décrémente CX et effectue un saut au label désigné	
Fonctions	call label	pour appeler la fonction	
Fonctions	ret	pour sortir de la fonction	
Pile		utilisée pour les arguments des fonctions, et pour sauvegarder le contexte lors d'interruption. utilise SS:ESP	
Pile	push valeur	empilement (ESP diminue)	
Pile	pop valeur	dépilement (ESP augmente)	
Interruption	int numéro	appeler une interruption	
Interruption	iret	sortir d'une interruption	

Nom	Utilisation
0x0 à 0x7	processeur
0x8 à 0xf	périphériques
0x10	vidéo
0x13	accès aux disques
0x16	clavier

Nom	Utilisation
0x1C	horloge
0x20	DOS : terminer un programme
0x21	DOS : API
0x28	DOS : boucle d'attente du shel
80h	DOS : Unix : API, util pour les appels systèmes Unix

Appels systèmes sous Linux
unistd.h contient les numéros de chaque appel système
le numéro de l'appel système est dans EAX
les paramètres sont passés via EBX, ECX, EDX, ESI, EDI, EBP (dans l'ordre)

4 Exemples

... exemple calcule, boucle, algos,...

```
        //xor ax,ax
i=0             mov ax,0
i=0             mov bx,100
j=100 debut:
        cmp ax,20
        jl fin
        cmp bx,50
        jge fin
        mov dx,ax
        imul dx,2
        sub bx,dx
        cmp bx,10
        jge else
        add bx,120
        jmp endif
        else:...
        endif:
        inc ax
        jmp debut
fin:
OU Inverse
debut :
        . . .
        inc ax;
        cmp ax,10
        jg debut
```

```
section .data
table: db 'Hello', 3
size: equ $ - table
char: db 0
section .text
global _start
_start:
        mov al, [table]; on instancie le minimum
avec le premier élément de la table
        mov ebx, 0 ;offset
find_minimum:
.loop:
        cmp ebx, size
        jae subtract
        cmp al, [table+ebx]
        jb .no_update ; Si le caractère n'est pas ¡
minimum on ne fait rien
        mov al, [table+ebx]
        .no_update:
        inc ebx
        jmp .loop
subtract:
        mov ebx, 0
.loop:
        cmp ebx, size
        jae end_loop
        sub [table+ebx], al ; on soustrait la
valeur
        ; et on affiche pour vérifier que tout
fonctionne
        push eax
        mov dl, [table+ebx]
        push ebx
        mov eax, 4
        mov ebx, 1
        mov [char], dl
        mov ecx, char
        mov edx, 1
        int 0x80
        pop ebx
        pop eax
        inc ebx
        jmp .loop
end_loop:
        mov eax, 1
        mov ebx, 0
        int 0x80
```

```
tri par bulle, pour un tableau table de taille n
section .data
table: db 'teststring'
n: equ $ - table
section .text
global _start
_start:
mov al, 0
.loop_i: ; i est implémenté avec al
        cmp al, n
        jae .end_loop_i
        mov ebx, 1 ; j est stocké dans ebx (taille
32 bits pour les offsets)
        .loop_j:
        mov ecx, n
        sub ecx, eax
        cmp ebx, ecx ; j n'a pas besoin d'aller au
dela de n-i-1
        jae .end_loop_j
                ;on place [table + ebx] dans cl
pour pouvoir utiliser cmp
                mov cl, [table + ebx]
                cmp cl, [table + ebx - 1]
                jae .no_swap ; si t[j] ¿ t[j-1] il n'y
a rien à faire
                 ; on met t[j-1] dans t[j]
                 mov dl, [table + ebx-1]
                 mov [table + ebx], dl
                 mov [table + ebx - 1], cl ; on
met t[j] dans t[j-1]
                .no_swap:
                inc ebx
        jmp .loop_j
        .end_loop_j:
        inc al
jmp .loop_i
.end_loop_i:
mov eax, 0
.loop_print:
        cmp eax, n
        jae end_print
        mov ecx, table
        add ecx, eax
        push eax
        mov eax, 4
         mov ebx, 1
        mov edx, 1
        int 0x80
        pop eax
        inc eax
        jmp .loop_print
end_print:
        mov eax, 1
        mov ebx, 0
        int 0x80
```

...

accès à la libc en assembleur:
utilisation de "extern" et utilisation de "main" au lieu de "_start" (car "_start" est redéfini par le linker) : la fonction doit respecter l'interface imposée (pour ne pas avoir d'effet de bord)
(exemple avec extern puts, exit)

```
int main() {
        int value = 100;
        asm( "mov eax, %0;"
                "inc eax;"
                "mov %0, eax;"
                :"=r"(value):"r"(value)
        );
        printf("Hello world");
        printf("value is %d (should not be ",
value);
        printf("equal to 100 anymore)");
        return 0;
}
```

compiler avec "gcc -masm=intel -o methode2 methode2.c"
Pour accéder aux variables en écriture, on utilise "=r" (pour registre)
Pour accéder aux variables en lecture, on utilise "r"
Dans la partie assembleur, on utilise %0, %1, etc

optimiser le code assembleur produit par du C
gcc -S -masm=intel methode3.c
modifier le fichier methode3.s
gcc -o methode3 methode3.s

5 Autres

• •