Insaïd Ze Ordi
Cours sportif N° 1 d'INF-445

Ronan KERYELL Gérald OUVRADOU

11 mai 2007
version Id : notice.tex,v 1.15 2007/03/30 23 :16 :02 keryell Exp

Résumé

Le but de ce cours associé à une bonne dose de travaux pratiques sportifs est de faire sentir aux sportifs les mécanismes qui sont derrière l'exécution de programmes dans les ordinateurs.

Il s'agira de découvrir des méthodes d'auto-organisation pour exécuter au mieux les programmes.

Accessoirement ce sera aussi l'occasion de (ré)viser le langage C.

Le tout sera basé sur une active participation des intervenants.

Les éléments de ce cours sont disponibles dans :
`http://enstb.org/~keryell/cours/ENSTBr/TC/arch_temps_reel/insaid_ze_ordi/`.

1 Introduction

Le but de cet exercice est de se mettre dans la peau d'un processeur qui exécute du code avec ses différentes unités.

Chaque joueur aura une tâche particulière et par recoupement chacun devra deviner ce que fait le programme exécuté.

À la fin on distribuera le code C et l'assembleur du programme pour comparer à ce que vous avez deviné.

Beaucoup de choses risquent d'être incompréhensibles à de nombreux élèves. Le but est d'intuiter, puis de demander à ses camarades voire aux encadrants de vous expliquer ce qui n'est pas compris.

1.1 Arithmétique binaire

On commencera par rappeler l'arithmétique binaire, ses subtilités et ses représentations (hexadécimal par exemple).

2 La structure de l'ordinateur

2.1 Le jeu d'instruction x86

Comme on a pris comme exemple du code de processeur de PC (style x86) c'est évidemment ce jeu d'instruction qui sera regardé.

Code binaire	Instruction en assembleur
01 44 24 0c	add %eax,0xc(%esp)
03 44 24 0c	add 0xc(%esp),%eax
0f af c2	imul %edx,%eax
40	inc %eax
51	push %ecx
59	pop %ecx
7e c9	jle 8048344 <main+0x20>
83 7c 24 08 02	cmpl $0x2,0x8(%esp)
83 c4 10	add $0x10,%esp
83 e4 f0	and $0xfffffff0,%esp
83 ec 10	sub $0x10,%esp
89 04 95 9c 95 04 08	mov %eax,0x804959c(,%edx,4)
8b 04 85 88 95 04 08	mov 0x8049588(,%eax,4),%eax
8b 14 85 7c 95 04 08	mov 0x804957c(,%eax,4),%edx
8b 44 24 08	mov 0x8(%esp),%eax
8b 44 24 0c	mov 0xc(%esp),%eax
8b 54 24 08	mov 0x8(%esp),%edx
8d 4c 24 04	lea 0x4(%esp),%ecx
8d 61 fc	lea 0xfffffffc(%ecx),%esp
90	nop
a1 98 95 04 08	mov 0x8049598,%eax
a1 9c 95 04 08	mov 0x804959c,%eax
a3 98 95 04 08	mov %eax,0x8049598
c3	ret
c7 44 24 08 00 00 00 00	movl $0x0,0x8(%esp)
c7 44 24 0c 00 00 00 00	movl $0x0,0xc(%esp)
eb 30	jmp 8048374 <main+0x50>
ff 44 24 08	incl 0x8(%esp)
ff 71 fc	pushl 0xfffffffc(%ecx)

TAB. 1 – Liste des instructions utilisées (attention aux instructions de branchement qui considèrent un adressage relatif !).

Dans la doc `info` de l'assembleur de GNU `gas` on peut trouver certaines choses et dans la FAQ à `http://www.faqs.org/faqs/assembly-language/x86/general` beaucoup plus de choses.

Les instructions utilisées dans notre exemple sont traduites du binaire en codes plus compréhensibles par un humain sur la table 1.

Le processeur contient 8 registres qui sont des cases mémoires de 32 bits accessibles de manière ultra rapide.

Les instructions x86 sont spéciales dans la mesure où elles ne sont pas à 3 adresses. On ne peut pas faire

```
op3 = op1 + op2
```

par exemple mais seulement

```
op2 += op1
```

ce qui complique les choses.

Avec l'assembleur utilisé ici (syntaxe style AT&T et non Intel) l'ordre des opérandes est de type

```
opération op1,op2
```

par rapport à l'équivalent en C précédent.

Le mode d'adressage dans un opérande est de type

```
section:disp(base, index, scale)
```

qui représente dans la `section` (sans importance ici) l'adresse `disp`+`base`+`index`×`scale`.

Concrètement `disp(,index,scale)` cela peut servir à représenter

```
&disp[index]
```

du C.

Il y a un mécanisme de pile pour empiler (`push`) et dépiler (`pop`) des valeurs sur la pile dont le sommet est pointé par le registre de pile `%esp` (*extended stack pointer)*. Le fonctionnement équivalent en pseudo-C serait :

- `push` *a* équivalent à `esp -= sizeof(`*a*`) ; *esp = `*a*
- `pop` *a* équivalent à *a* `= *esp ; esp += sizeof(`*a*`)`

L'instruction `leave` est équivalente à :

```
mov %ebp,%esp
pop %ebp
```

L'instruction `lea` (*load effective address)* sert à faire des calculs d'adresses (style arithmétique de pointeur,...) ou des calculs simples qui rentrent dans le cadre `disp`+`base`+`index`×`scale`.

L'instruction `ret` est un retour de fonction ou procédure et récupère sur la pile l'adresse de retour d'exécution dans le PC (ou `%eip` sur $x86$) et le registre de status, c'est à dire fait un

```
pop %eip
pop %cs
```

3 Les rôles des joueurs

Toutes les sections suivantes doivent être réalisées par des élèves différents pour exécuter globalement le programme donné.

3.1 La mémoire centrale

Elle contient les instructions du programme à exécuter et éventuellement du code de service (boot, vecteur IT, ...). Elle sert aussi de tampon pour les données manipulées par le programme entre l'Unité Centrale (UC) et l'environnement de l'ordinateur.

3.2 Le bus mémoire

C'est par lui que passent tous les échanges entre l'UC et son environnement immédiat, i.e. la mémoire centrale, Unités d'entrée/sortie.

Il est en général constitué de 3 sections : adresses, données, contrôle et associé à un système de gestion qui implémente les protocoles d'utilisation du bus.

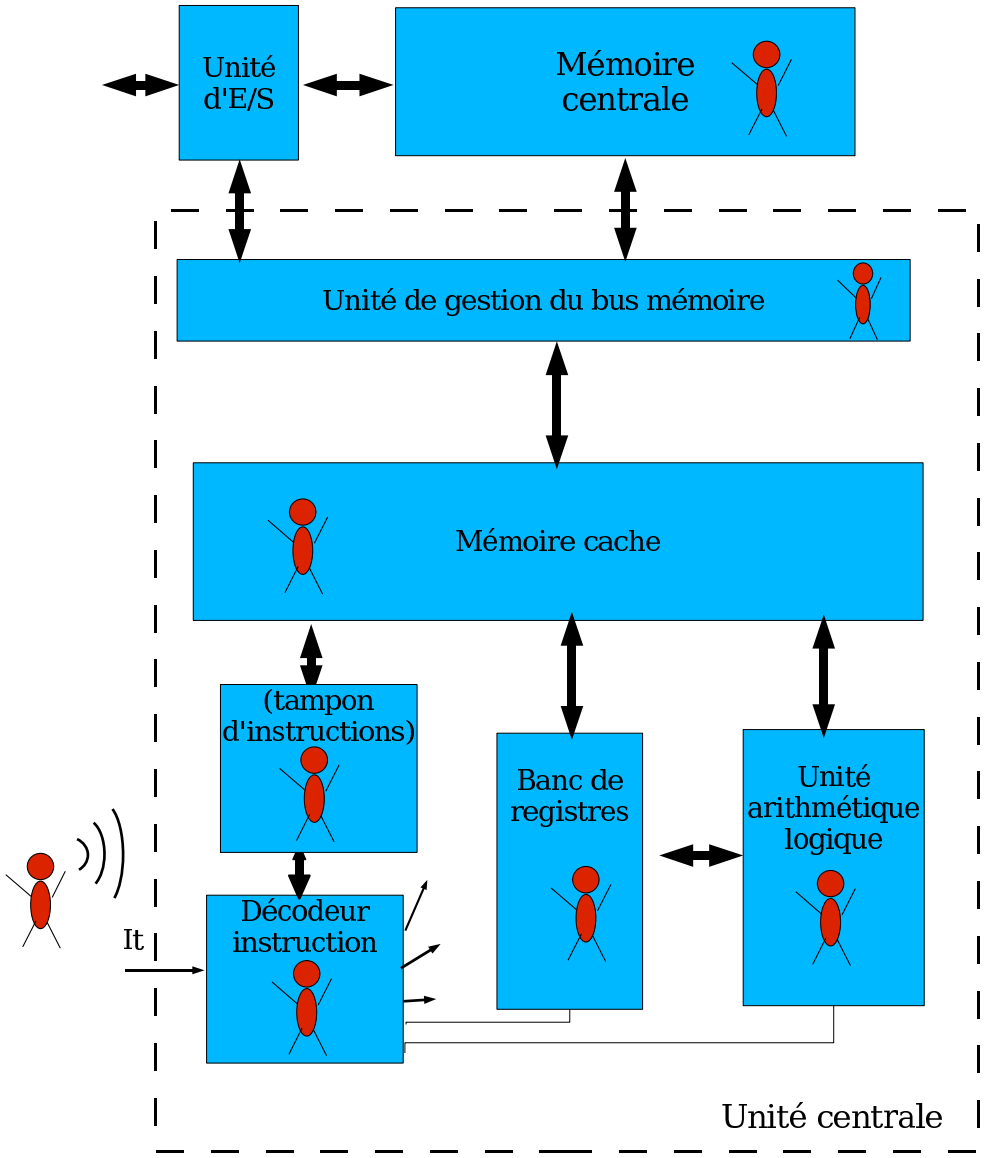

FIG. 1 – Vue générale de l'architecture mise en *live*.

3.3 Au sein du processeur ou Unité Centrale (UC)

3.3.1 Le tampon d'instruction

Son rôle est d'aller lire des paquets d'instructions dans la mémoire et d'alimenter, à la demande, le décodeur d'instructions (voir suite).

3.3.2 Le décodeur d'instruction

Le décodeur d'instruction extrait une par une les instructions du tampon et les exécute l'une après l'autre. Il maintient un registre spécialisé appelé «

compteur ordinal » (CO) ou « *program counter* » (PC en anglais) qui contient l'adresse mémoire de l'instruction à extraire du tampon.

Chaque instruction fait l'objet d'un même traitement :

1. décodage de l'instruction, c'est-à-dire identification de l'instruction au sein du jeu d'instruction du processeur ;
2. recherche opérande (0, 1 ou 2 selon les cas) selon modalités spécifiées dans l'instruction ;
3. exécution du traitement spécifié dans instruction ;
4. rangement du résultat selon modalités spécifiées dans l'instruction ;
5. mise à jour CO (alias PC) ;
6. test de présence d'interruption ;
7. recherche de la prochaine instruction, puis retour au pas 1.

3.3.3 Les interruptions

Il s'agit de la possibilité donnée au programmeur de dérouter le programme qui s'exécute lors de la survenue d'un évènement externe à l'ordinateur ou interne (anomalie, exemple : division par 0). En général, une adresse prédeterminée de la mémoire contient la nouvelle valeur à placer dans le CO quand le décodeur d'instruction prendra en compte l'interruption. La nécessité de reprendre le programme là où il avait été interrompu implique d'effectuer une sauvegarde de l'état de certains éléments de l'UC avant de servir l'interruption (lesquels ?).

Dans ce TP ce sera un encadrant qui provoquera des interruptions.

3.3.4 Le banc de registres

Il contient des registres spécialisés (ex : registre d'état) et des registres à usage général (i.e. que le programmeur peut utiliser comme il l'entend). Voir les interconnexions du banc de registres avec son environnement sur la figure 1.

3.3.5 L'unité de calcul arithmétique et logique

Comme son nom l'indique, c'est elle qui effectue les traitements des opérandes spécifiés par les instructions, mais aussi certains calculs inhérents à la mise en œuvre des instructions (calcul d'adresse, calcul sur le CO ou PC).

3.4 Le système de mesure

Inventez-le !

On peut mesurer par exemple le nombre d'instructions exécutées par secondes... euh, par heure plutôt. ☺ On pourrait mesurer le temps d'exécution de chaque instruction et voir si cela s'améliore petit à petit.

Mettre en place un statisticien.

4 Déroulement de la séance

Tout d'abord, chaque élève effectuera une lecture attentive de ce document. Ensuite, les équipes seront constituées, à raison d'un acteur par unité (cf. schéma de la figure 1). Il faudra éventuellement assigner plusieurs rôle à un même acteur.

Chaque équipe organise alors son espace (voir section 4.1). On répartit les rôles sur une base tournante. Idéalement, chaque acteur devrait passer aux différents postes, mais le temps ne le permettra probablement pas.

Chaque acteur explique au reste de l'équipe ce qu'il a compris de son premier rôle. On explicite notamment les documents associés à chaque unité. On fixe enfin les modalités de mesure de performance pour chaque unité et les traces d'exécution qu'il faudra consigner.

4.1 Placement des joueurs

De même qu'un architecte organise les différents constituants d'un ordinateur, les joueurs devront se placer de manière cohérente avec la réalité.

Comme la principale limitation d'un architecte est la vitesse de la lumière et le fait que les cartes des ordinateurs sont bassement bidimensionnelles il faut organiser les choses pour que la distance entre chaque module interagissant soit la plus faible possible.

Les joueurs devront se concerter pour analyser leur interdépendance afin de se placer au mieux.

Le cas de la mémoire est particulier car elle est en dehors du processeur pour des raisons de technologie[1] et loin du processeur sur le circuit imprimé de l'ordinateur dans des barrettes séparées.

À l'échelle du processeur en terme de vitesse d'exécution la mémoire est *très* loin. Pour être représentatif on placera donc le joueur mémoire le plus loin possible des autres joueurs, rajoutant au côté sportif. Par exemple, l'élève qui gère la mémoire pourrait être placé au centre-vie ou au foyer[2].

4.2 C'est parti !

Chacun à son poste, chaque unité travaille en interagissant avec une ou plusieurs autres unités. Chaque acteur joue et tient un journal de son activité.

Au top qui sera donné par l'enseignant, on effectuera une permutation des rôles.

4.3 Débriefing

L'exercice est stoppé au bout d'un certain temps. Chaque équipe devra faire un petit bilan de l'expérience et se verra confier la tâche d'expliciter le fonctionnement d'une ou plusieurs unités sous forme d'un petit exposé informel. Concertation et mise au point préalable au sein de l'équipe.

5 Augmentation des performances

Essayer de voir comment réorganiser les choses ou rajouter du matériel pour augmenter les performances d'exécution. On utilisera les mesures de performance pour voir ce qu'on gagne.

[1] Le monde est mal fait : on ne peut pas avoir à la fois des transistors ultra-rapides et de la mémoire ultra-dense... ☹

[2] Accompagnée d'une boisson chaude pour compenser l'isolement ! ☺

On pourra réallouer les ressources humaines, travailler en anticipation, améliorer la disposition dans la salle, optimiser la chorégraphie des acteurs...

Discussion au sein de chaque équipe, puis exposé des propositions.

6 Dis ! Comment ça marche ?

Les éléments du jeu sont générés automatiquement par un `Makefile`.

On compile normalement un programme C en assembleur et au format exécutable.

Ensuite la sortie de `objdump` sur le programme binaire est analysée par un programme en Python qui place les instructions en mémoire et génère une représentation en LaTeX du tout, la notice que vous lisez et les pièces du plateau de jeu. C'est bien l'informatique ! ☺

7 Conclusion

Le processeur n'a pas non plus conscience de ce qu'il exécute. ☺