EXECUTER UN PROGRAMME EN ASSEMBLEUR

Problématique

- Lorsqu’on lance un programme, l’exécutable se trouve sur un disque.
- Il est tranférer en RAM par le SE
- Comment fait le processeur pour exécuter ce programme

Exemple

```
L1:
.word a
.word b
.word c
.comm a,4,4
.comm b,4,4
.comm c,4,4
1) mov r0,#10
2) ldr r1,L1
3) str r0,[r1]
4) mov r0,#20
5) ldr r1,L1+4
6) str r0,[r1]
7) ldr r0,L1
8) ldr r1,[r0]
9) ldr r0,L1+4
10) ldr r2,[r0]
11) add r1,r1,r2
12) ldr r0,L1+8
13) str r1,[r0]
```

Programme C correspondant

```
int a,b,c;

main()

{a=10;
b=20;
c=a+b;
}
```

Directives

- Les directives ne sont pas codées en RAM
- Elles sont exécutéees par le SE lors du transfert du programme en RAM

4 zones mémoires

- Zone 1 (12 octets) : adresse L1
  adresse de a
  adresse de b
  adresse de C
- Zone 2 (4 octets): valeur de a
- Zone 3 (4 octets) : valeur de b
- Zone 4 (4 octets) : valeur de c

CODE_OP

- Chaque instruction possède un CODE_OP sur 32 bits qui contient :
- Le nom de l’instruction
- Les modes d’adressages
- Les numéros des registres

Mov r0,#10

- CODE_OP
  mov r0,#?
- Paramètre
  Constante 10 sur 32 bits en B2

ldr r0,L1

- code_op
  ldr r0,?
- Paramètre
  adresse L1 (32 bits)

Str r0,[r1]

- CODE_OP
  Str r0,[r1]
- Pas de paramètre

Modes d’adressage

- Constante : mov r0,#10
- Registre : mov r0,r1
- Adresse : ldr r0,L1
- Indirect : ldr r0,[r1]
- Indirect avec décalage constant : ldr r0,[r1,#4]
- Indirect avec décalage par registre ldr r0,[r1,r2]
- Indirect avec décalage constant et barrel shifter
  ldr r0,[r1,r2, asl #2]

Paramètre

- Une instruction peut posséder un paramètre sur 32 bits qui contient :
  - Une adresse
  - Une constante

Codage du programme en RAM

0000 1000 → CODE_OP1 PARAM_10
0000 1008 → CODE_OP2 ADRESSE_L1
0000 1010 →CODE_OP3
0000 1014 → CODE_OP4 PARAM_20
0000 101C → CODE_OP5 ADRESSE_L1+4
0000 1024 → CODE_OP6
0000 1028 → CODE_OP7 ADRESSE_L1
0000 1030 → CODE_OP8
0000 1034 → CODE_OP9 ADRESSE_L1+4
0000 103C → CODE_OP10
0000 1040 → CODE_OP11
0000 1044 → CODE_OP12 ADRESSE_L1+8
0000 104C → CODE_OP13

Compteur Ordinal

- Un registre CO contient l’adresse du premier code_op
- Ici CO=0000 1000
- En anglais Program Counter = PC

Registre Instruction

- Il contient le code_op de l’instruction en cours

- Le processeur va lire la case mémoire CO et va mettre son contenu dans RI
- En anglais Instruction Register=IR

Exécution d'une instruction

- Chaque instruction est traduite en micro-instructions
- Exemple de micro-instruction
  - Registre1 ← Registre 2 : on copie registre2 dans registre 1
  - Registre1 ← [Registre2] : on met dans registre 1 le contenu de la case mémoire n° registre 2
  - Registre 1 → [Registre2] : on met la valeur de registre 1 dans la case mémoire n° registre 2
  - Registre 1 ← Registre 2+ registre3 : on addionne la valeur de registre2 et de registre 3 et on met le résultat dans Registre1
  - Registre 1 ← Registre1 +4 : on incrémente registre 1 de 4

Mov r0,#?

- CO vaut 0000 1000
- 1) RI ← [CO];CO ← CO+4; chargement
  2) décodage (examine RI et le registre d'état)
  3) r0 ← [CO];CO ← CO+4

mov r0,#10

1) RI ← [R0] ; CO ← CO+4
2) décodage
3) R0 ← [CO]; CO ← CO+4

Ldr r0, L1

4) RI ← [R0] ; CO ← CO+4
5) décodage
6) RP ← [CO]; CO ← CO+4
7) R0 ← [RP]

Str r0,[r1]

8) RI ← [R0] ; CO ← CO+4
9) décodage
10) R0 ← [R1];

Mov r0,#20

11) RI ← [R0] ; CO ← CO+4
12) décodage
13) R0 ← [CO]; CO ← CO+4

Ldr r1,L1+4

- 14) RI ← [R0] ; CO ← CO+4
  15) décodage
  16) RP ← [CO]; CO ← CO+4
  17) R1 ← [RP]

Str r0,[r1]

- 18) RI ← [R0] ; CO ← CO+4
  19) décodage
  20) r0 ← [r1]

Ldr r0,L1

- 21) RI ← [R0] ; CO ← CO+4
  22) décodage
  23) RP ← [CO]; CO ← CO+4
  24) R0 ← [RP]

Ldr r1,[r0]

25) RI ← [R0] ; CO ← CO+4
26) décodage
27) R1 ← [R0];

Ldr r0,L1+4

- 28) RI ← [R0] ; CO ← CO+4
  29)décodage
  30) RP ← [CO]; CO ← CO+4

  31)r0 ← [R0]

Ldr r2,[r1]

- 32) RI ← [R0] ; CO ← CO+4
  33) décodage
  34) R2 ← [R1];

Add r1,r1,r2

- 35) RI ← [R0] ; CO ← CO+4
  36) décodage
  37) R1 ← R1+R2

Ldr r0,L1+8

- 38) RI ← [R0] ; CO ← CO+4
  39)décodage
  40) RP ← [CO]; CO ← CO+4
  41) RO ← [RP]

Str r1,[r0]

- 42) RI ← [R0] ; CO ← CO+4
  43) décodage
  44) R1 ← [RO];

A quelle vitesse cadencer le processeur

- Il faut cadencer le processeur à la vitesse la plus lente des différentes opérations.
- En général, c’est la phase de décodage qui est la plus lente.

Le décodeur

- Le décodeur est un circuit qui à chaque top fournit les micro-instructions à exécuter.
- Il y a toujours une phase de chargement et une de décodage.
- Ensuite il y a une phase d’exécution qui dépend du code_op et du registre d’état.

Comment est réalisé le décodeur ?

- Si il y a peu d’instructions et peu de modes d’adressage : on peut envisager toutes les combinaisons lors de la création du processeur et mettre tout cela dans une ROM.
- Si ce nombre est plus importants, on réalisera un circuit qui réalisera cette opération.

Décodage sur un processeur complexe

- Si le processeur est complexe, il est parfois nécessaire d’avoir un processeur simple inclus dans le processeur principal uniquement destiné au décodage.
- Un programme réalisera alors le décodage.
- Cette solution est souple et permet simplement l’ajout de nouvelles instructions assembleur.
- Par contre la conception du processeur est complexe.

Une remarque intéressante

- On a introduit de nombreuses instructions assembleur et de modes d’adressage dans les processeurs.
- Les compilateurs ne connaisent pas la totalité des instructions assembleur et des modes d’adressage.
- Les compilateurs utilisent principalement les instructions les plus courantes : mov, ldr, str
- Conclusion : à quoi ça sert d’avoir créé une telle complexité ?

Processeur RISC et CISC

- Un processeur RISC est un processeur avec peu de modes d’adressage et peu d’instructions assembleur. (Power PC, ARM par exemple)
- Un processur CISC est un processeur avec beaucoup de modes d’adressage et d’instructions assembleur (X86,68000)

Comparaison

- Pour un même programme, il faut plus d’instructions assembleur sur un processeur RISC que sur un CISC.
- Par contre le décodage est plus simple sur un RISC donc le top horloge est plus petit.
- l’avantage va aujourd’hui aux processeurs RISC

Puces pour smartphone

- Apple Bionic A11 (Iphone X) : architecture arm 64 bits
- Exynos : également ARM 64 bits
- Le RISC s’impose

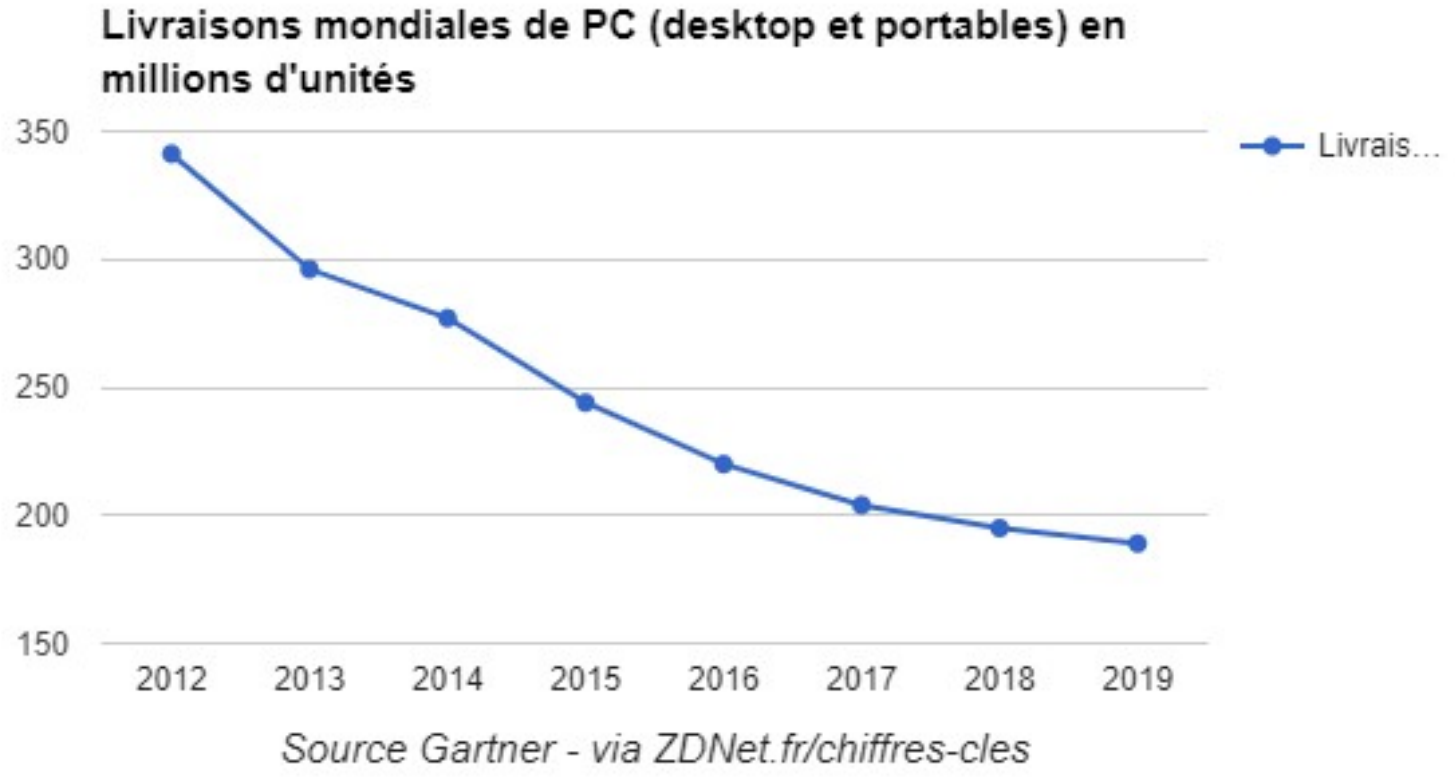
Puissance des processeurs smartphone vs X86 stockfish

- Core i7-4700mq (@2.4 Ghz) 5.492.000
- Iphone 8+ (A11, 6 cores) 4.818.800

L’exception X86

- L’architecture X86 est CISC
- Intel a créé une famille de processeurs CISC très perfermante mais horrible à faire évoluer.
- Un bijou technologique mais une “usine à gaz” indescriptible.
- Un succès commercial avec un rapport prix performance intéressant.

Avenir du X86 et du CISC ?



Le nouveau monde

- l’essor du deep learning change radicalement la donne sur le marché des processeurs
- La performance pure d’une machine n’est peut être plus le principal critère
- Sa capacité d’apprentissage intelligent devient le critère déterminant.

Deep mind

- Mars 2016 : AlphaGo bat lee sedol (meilleur joueur du monde entre 2000 et 2010) : apprend en examinant des parties jouées entre des très forts joueurs.
- Décembre 2017 : AlphaZero bat la version AlphaGo Zero (3 jours d’apprentissage et 21 millions de parties jouées contre lui-même)
- Janvier 2019 : alphaStar bat 2 professionnels à star craft. Joue contre lui-même sur une version 1000 fois plus rapide de star craft.
- On ne connait pas les limites de ces technologies

IA : une nouvelle donne

- Iphone XS est équipé d’une puce A12 Bionic comportant une partie “neural engine” dédié au deep learning.
- Ce Neural Engine est doté de 8 coeurs.
- Capacité d’apprentissage “intelligent” énorme ?
- Premiers usages : comprendre rapidement la scène qu’il photographie.

Puces IA

- TPU google : Puces spécialisées dans la bibliothèque de deep learning tensorflow de google : cloud TPU (pas de vente)
- Nvidia : utilisation des GPU pour la bibliothèque tensorflow