

# Programmation assembleur (ASM)

## *Introduction*

Prof. Daniel Rossier  
Version 1.3 (2015-2016)

## Contact & Infos diverses

- Contact

- ✉ [daniel.rossier@heig-vd.ch](mailto:daniel.rossier@heig-vd.ch)

- 💬 [skype: rossierd](#)

- Institut REDS

- *Reconfigurable Embedded Digital Systems*

- <http://www.reds.ch>



- Bureau A21

- <http://www.linkedin.com>

## Cours ASM - Compétences-métier

- **Appliquer** la programmation assembleur sur une plateforme 32/64 bits
- **Etre capable** de mettre au point une application sur un processeur ARM
- **Maîtriser** la programmation mixte C/ASM

## Déroulement du cours ASM (1/3)

- Matériel du cours et laboratoires
  - <http://www.reds.ch>
  - <http://www.reds.ch/fr/Formations/Master/SEEDoc.aspx>
- **Contrôles continus & laboratoires** (50 %)
  - 2 travaux écrits (TEs) portant sur les cours et les laboratoires
  - 6 laboratoires
- **Examen final** (50 %)

## Déroulement du cours ASM (2/3)

- Les laboratoires se feront soit **individuellement**, soit par **groupe**.
- L'évaluation des laboratoires est décrite sur une **feuille annexe**.
- Les laboratoires portent sur les différentes parties du cours.
  - Environnements de développement croisés
  - Déploiement sur cibles émulés
  - Mise au point des applications
  - Exercices en langage assembleur

## Déroulement du cours ASM (3/3)

- Introduction
- Environnements croisés et *debugging*
- Assembleurs *x86* et *ARM* (Introduction)
- Instructions de traitement
- Instructions de transfert
- Instructions de branchement
- Interactions C/assembleur

## Plan (Introduction)

- Objectifs
- Architectures système
- Programmation système en C

## Objectifs (1/2)

- Quelle est l'importance du langage assembleur aujourd'hui ?

- Proche du **code binaire** contenant les instructions machine

- Possibilité de **désassembler** le code binaire

- **Sécurisation** logicielle

- **Optimisation** et performance



```
0011940 0010 fff1 005d 0000 0000 0000 0000
0011950 0010 fff1 0060 0000 0000 0001 0000 0000
0011960 0010 fff1 0077 0000 0000 0001 0000 0000
0011970 0010 fff1 0083 0000 0000 0001 0000 0000
0011980 0010 fff1 0080 0000 0000 0001 0000 0000
0011990 0010 fff1 0082 0000 0000 0001 0000 0000
00119a0 0010 fff1 0097 0000 0000 0000 0000 0000
00119b0 0012 0001 009d 0000 0000 0001 0000 0000
00119c0 0010 0001 7300 6174 6063 6c5f 776f 2400
00119d0 0064 7473 6361 576a 6968 6867 2400 0061
00119e0 616d 6a69 5700 6c65 6f63 656d 6c00 6f6f
00119f0 3170 6500 6978 0074 6174 0062 5f5f 7865
```

```
0208: 1affffff 0000 0204 <1600>
0000204 <end_strcat>:
0204: e0d00007  ldmia sp!, {r0, r1, r2}
0208: e59f2478  ldr r2, [pc, #1144] ; 8768 <tab+0x5b>
020c: e9200003  stmdb sp!, {r0, r1}
02f0: e1a01002  mov r1, r2
02f4: e4010001  ldrb r0, [r1], #1
02f8: e3500000  cmp r0, #0 ; 0x0
02fc: 8a000011  beq 8348 <end_strcat+0x64>
0300: e92047ff  stmdb sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, sl}
0304: e1a00000  nop (mov r0, r0)
```

8

Cours ASM - Institut REDS/HEIG-VD - Introduction

De nos jours, le développement d'un programme complet en assembleur est assez rare et concerne des applications très particulières (ressources matérielles limitées, code critique ultra-rapide, etc.). Néanmoins, il ne faut jamais oublier qu'un code source développé dans un langage de haut niveau sera toujours compilé et pourra tourner sur un processeur qu'avec des **instructions machines** dépendant directement de celui-ci (c-à-d des instructions appartenant au jeu d'instructions du processeur). Typiquement, si la traduction d'un code source vers un code machine peut s'effectuer sans problème, il en va tout autrement pour une traduction inverse (par exemple d'un code binaire vers le code C original). En revanche, la traduction de code binaire vers un code assembleur est beaucoup plus aisée. C'est pourquoi, le **désassemblage** d'un code binaire permettra au développeur de récupérer un code en langage clair et fournira des informations précieuses pour autant que l'on comprenne le langage assembleur.

Ce cours de programmation assembleur donne les fondements de ce type de programmation afin que le développeur puisse profiter au maximum de ce langage pour exploiter au mieux les ressources du processeur et faciliter la mise au point des logiciels lorsque cela s'avère nécessaire. De plus, il est évident que les *hackers* vont exploiter les failles des programmes (voire du matériel) en utilisant des techniques de très bas niveau qui reposent directement sur une compréhension avancée du système (structures de bas niveau liées au binaire, manipulations de la pile, corruption au niveau du processeur, etc.).



## Objectifs (2/2)

- Quels sont les contextes d'utilisation?
  - Amorçage d'un système (*bootstrap*)
  - Mise au point (*debugging*)
  - Accélération (*performance*)
  - Accès à des **instructions particulières**
  - *Reverse Engineering*

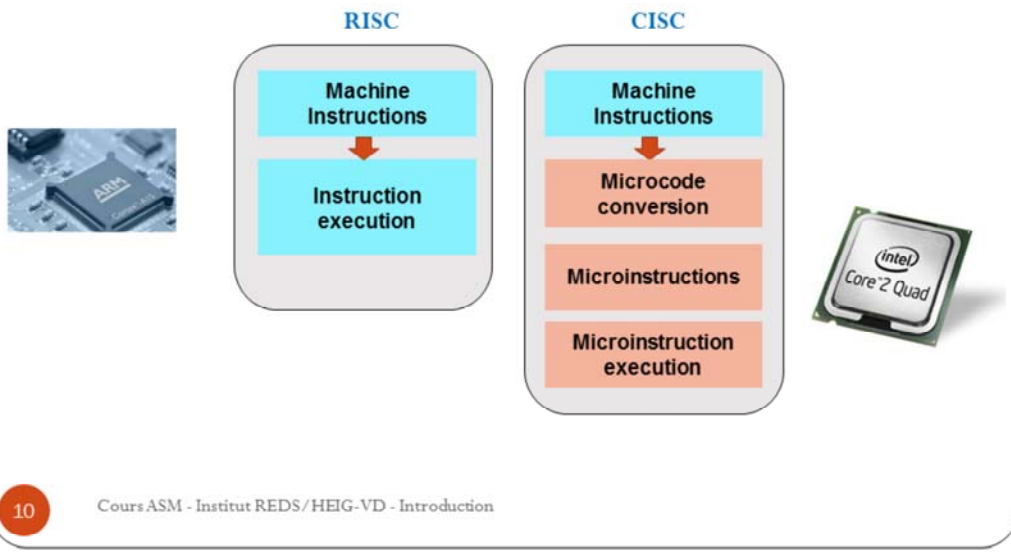


Parmi les utilisations les plus fréquentes de l'assembleur, on peut citer les contextes d'utilisation suivants:

- L'amorçage d'un système: lorsque l'on met un système sous tension, le processeur commence généralement à exécuter des instructions particulières de configuration/initialisation qui ne correspondent pas à une instruction d'un langage de haut niveau. C'est pourquoi, les premières instructions d'un code de démarrage (d'amorçage ou *bootstrap*) sont écrites en assembleur.
- La mise au point d'un programme (*debugging*) peut nécessiter l'utilisation d'un *debugger* connecté à l'application pendant que celle-ci s'exécute. Il est possible alors de récupérer le code désassemblé en temps-réel et d'obtenir de précieuses informations sur le comportement du programme (il se peut que le *debugger* ne puisse pas toujours associer le code source original écrit dans un langage de haut niveau avec l'instruction en cours d'exécution (à méditer!)).
- Dans certain cas, l'accélération d'un traitement peut nécessiter l'utilisation de l'assembleur, en particulier si le processeur dispose d'instructions performantes qui ne sont pas utilisées par le compilateur (*code DSP, coprocesseur arithmétique, etc.*).
- D'une manière générale, dès que l'on cherche à utiliser des instructions particulières du processeur, il faut utiliser du code assembleur. Comme nous le verrons dans ce cours, il est aussi possible de *mixer* du code C avec du code assembleur par exemple.
- Finalement, la compréhension de l'assembleur permet d'analyser du code binaire.

## Architectures des processeurs (1/7)

- Architecture RISC vs CISC



L'apprentissage de l'assembleur nécessite une bonne compréhension de l'architecture matérielle. Nous nous intéressons ici aux caractéristiques matérielles pouvant influencer le modèle de programmation.

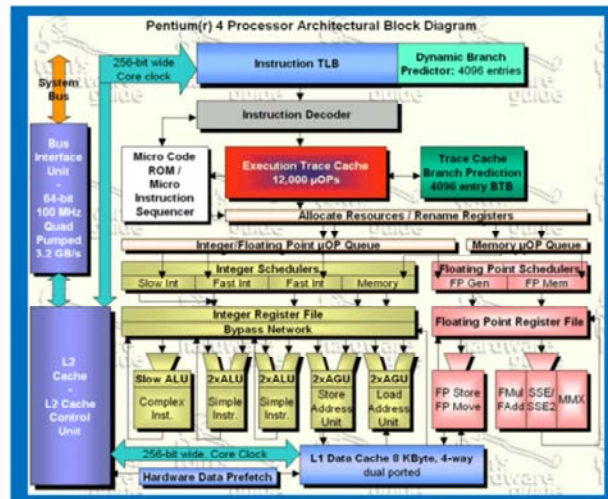
Il existe deux familles principales de processeurs: les processeurs basées sur une architecture **RISC** (*Reduced Instruction-Set Computer*) qui possèdent un jeu d'instruction réduit, et ceux basés sur une architecture **CISC** (*Complex Instruction-Set Computer*). La plupart des **microcontrôleurs** possèdent une architecture RISC.

Les processeurs RISC disposent de beaucoup de registres *généraux* (entre 16 et 32), tous équivalents, pour faciliter leur allocation par le compilateur. Les instructions sont de taille fixe, souvent 32 bits. Les instructions arithmétiques ont généralement 2 registres servant d'opérandes et un registre de sortie. Enfin, les accès à la mémoire font l'objet d'instructions spécifiques, et une valeur correspondant à une adresse doit d'abord être chargée dans un registre pour être utilisée: on parle d'architecture *load-store* ou d'instructions *register-register*.

Les processeurs CISC ont quant à eux de très nombreuses instructions combinées avec des modes d'adressage relativement complexes. Les instructions peuvent être de taille différente et sont parfois difficiles à utiliser par un compilateur. Elles peuvent nécessiter plusieurs coups d'horloge; le processeur est globalement plus complexe. Le jeu d'instruction peut être modifié par *microprogrammation*. L'architecture CISC est présente dans la plupart des processeurs de type Intel/AMD.

## Architectures des processeurs (2/7)

- Microprocesseur vs Microcontrôleur
- Architecture d'un microprocesseur (*Pentium/x86*)



11

Cours ASM - Institut REDS/HEIG-VD - Introduction

La différence entre un microprocesseur et un microcontrôleur réside au niveau du contenu du composant électronique (*chipset*).

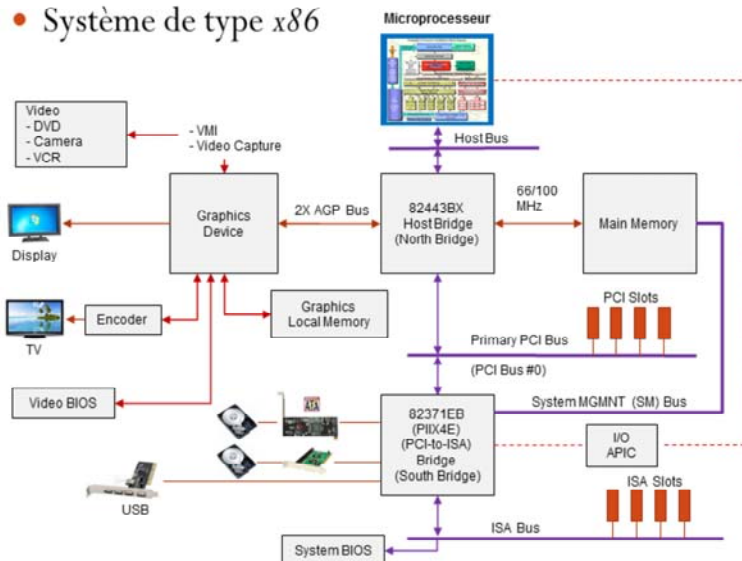
Dans le cas du microprocesseur, le *chipset* contient le-s cœur-s de calcul (*core*), c-à-d l'unité de **traitement** et de **calcul** basé sur un certain jeu d'instructions, les mémoires de type *cache*, les circuits permettant la gestion du *pipeline*, etc. Le *chipset* peut également contenir divers coprocesseurs utilisés pour le traitement arithmétique (opérations en virgule flottante), l'accélération de calcul, les opérations multimédia (accélération audio/vidéo), le chiffrement, etc.

On y trouve bien entendu un ensemble de registres programmables, dont notamment un registre d'état.

Un microprocesseur se destine en principe à un usage générale et constitue le composant central de tout ordinateur de type PC/laptop/serveur. Il est également souvent référé en tant que CPU (*Control Process Unit*) bien que ce terme devrait plutôt être lié au cœur de calcul intégré à l'intérieur du *chipset*. Bien entendu, le microprocesseur comporte des bus d'adresses et de données exportés vers l'extérieur afin de pouvoir y connecter des périphériques (typiquement un contrôleur de bus PCI). De plus, certaines lignes particulières (ports d'entrées-sorties I/O, lignes d'interruption) permettent de relier des composants plus critiques (horloges, contrôleur d'interruption, contrôleur mémoire, etc.) offrant ainsi une meilleure sécurisation des accès.

## Architectures des processeurs (3/7)

- Système de type x86



12

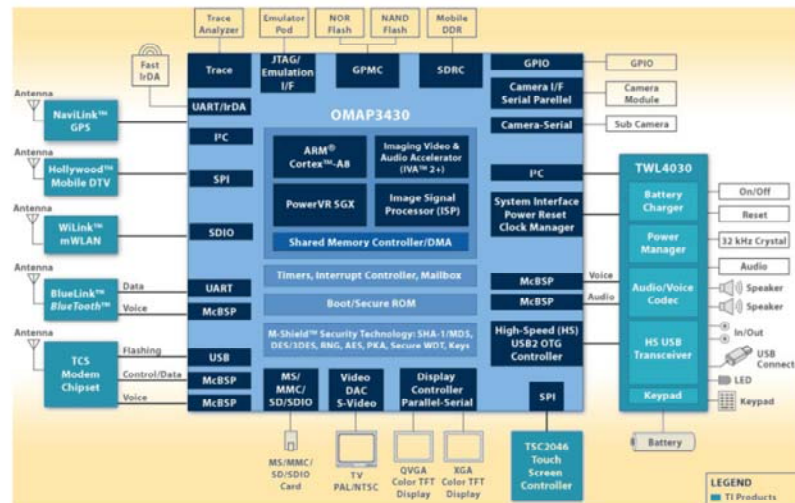
Cours ASM - Institut REDS/HEIG-VD - Introduction

Un exemple d'architecture de système basé sur un processeur de type x86 est présenté ci-dessus. On retrouve le *chipset* représentant le microprocesseur relié aux autres périphériques via les bus d'adresses/données. Aujourd'hui, la plupart des contrôleurs de périphériques peuvent facilement s'enficher dans des *slots* PCI, eux-mêmes disponibles sur la carte-mère du système sur laquelle se trouve le processeur.

Les périphériques (clavier, souris, écran, dispositifs USB, etc.) nécessitent *un contrôleur de périphérique* afin d'interagir correctement avec le processeur. Le contrôleur est constitué d'un (ou plusieurs) circuit électronique, généralement régulé d'une manière indépendante au processeur et disposant de leur propre cycle de vie. C'est pourquoi, un ensemble processeur-périphériques fait intervenir beaucoup d'interactions de type *asynchrones* qui sont pris en charge par le processeur au travers de lignes d'interruptions (IRQs). Le mécanisme d'interruption lui-même nécessite un contrôleur d'interruption permettant une gestion des priorités et des activations/désactivations (masquage/démasquage) de celles-ci. Le contrôleur d'interruption (ou **APIC** pour *Advanced Programmable Interrupt Controller*) est relié au processeur via des lignes dédiées.

## Architectures des processeurs (4/7)

- Architecture d'un microcontrôleur



13

Cours ASM - Institut REDS/HEIG-VD - Introduction

A la différence d'un microprocesseur, un microcontrôleur contient le processeur (cœur-s de calcul ainsi que coprocesseurs, mémoires cache, etc.), plus des contrôleurs de périphériques directement intégrés dans le *chipset*, ainsi que de la mémoire ROM/RAM et des entrées-sorties permettant de relier les composants externes vers les contrôleurs internes.

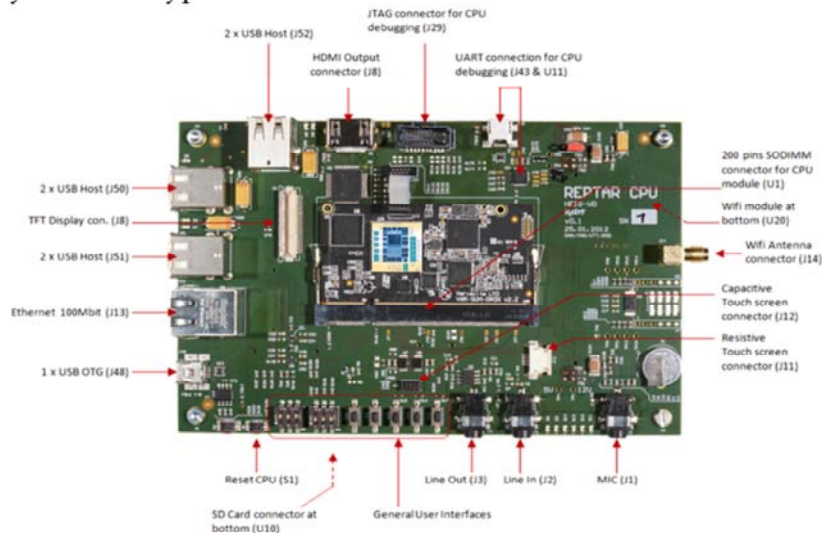
L'architecture du processeur et donc beaucoup plus "simple" que celui d'un microprocesseur et la consommation de courant est réduite. Ce type de circuit se retrouve majoritairement dans les systèmes embarqués, mais tend aujourd'hui à se répandre également dans les systèmes qui jusqu'ici ne comprenant que des microprocesseurs. C'est le cas par exemple des dernières génération de microcontrôleur de type ARM-Cortex A15, multicœur et supportant les instructions de virtualisation.

Les microcontrôleurs sont dès lors plus spécifiques que les microprocesseurs. De part leur architecture interne et les contrôleurs de périphériques qu'ils intègrent, certains se destineront aux applications multimédia (*smartphones*, tablettes PC, etc.), alors que d'autres conviendront mieux pour des systèmes temps-réels critiques (navigations, machines-outils, etc.) , ou encore aux systèmes de télécommunication (routeurs, *gateways*, etc.) ou aux systèmes de traitement rapide de données (encryptage, monitoring, etc.).

C'est dire que la gamme de microcontrôleurs est infiniment plus riche que celle des microprocesseurs.

## Architectures des processeurs (5/7)

- Système de type ARM



14

Cours ASM - Institut REDS/HEIG-VD - Introduction

Un exemple de système embarqué équipé d'un microcontrôleur est la plate-forme REPTAR (*Reconfigurable Embedded Platform for Training And Research*) développée au sein de l'institut REDS et employé dans de nombreux laboratoires de Bachelor et Master.

La plate-forme est bâtie autour d'un microcontrôleur de type *ARM Cortex-A8* de *Texas Instruments* (TI); il s'agit du microcontrôleur DM-3730. Le microcontrôleur est monté sur une carte de développement disponible sur le marché (carte *Variscite*); ce petit module contient de la RAM ainsi qu'une connexion *Ethernet* et un module *Wifi*. Comme la plupart des microcontrôleurs, le DM-3730 offre la possibilité de relier des périphériques externes au travers de *GPIO* (*General Purpose Input/Output*) permettant l'envoi/réception de données ou d'utiliser une ligne comme canal d'interruption. Ces lignes sont reliées en interne - dans le microcontrôleur - grâce à un contrôleur spécifique appelé contrôleur *GPIO*.

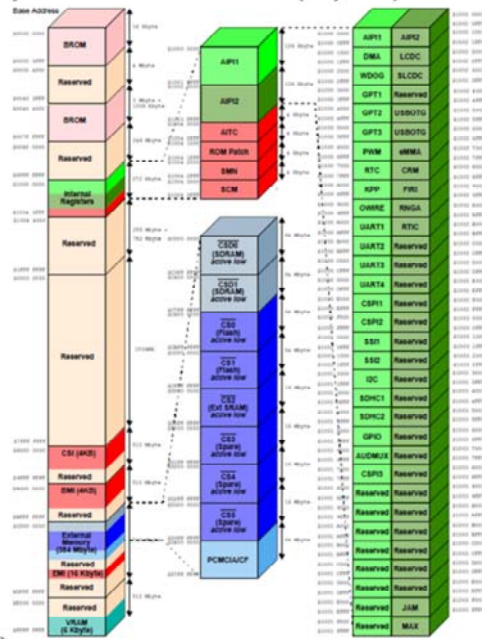
L'accès aux différents composants, et notamment aux contrôleurs de périphériques, s'effectue généralement au travers **d'accès mémoire** (lecture/écriture) à des adresses I/O bien définies (contrairement au microprocesseur qui peut lui piloter des composants externes via des ports I/O dédiés qui nécessitent des instructions d'accès réservées).

Les adresses I/Os sont déterminés grâce au **plan d'adressage** dicté - donc figé - par le microcontrôleur/microprocesseur.



## Architectures des processeurs (6/7)

- Plan d'adressage
  - Adresses 32/64 bits
  - Adressage **linéaire**
  - Mémoire **byte-adressable**
  - Accès via les instructions de lecture/écriture en mémoire



Le plan d'adressage d'un microprocesseur/microcontrôleur permet de déterminer les adresses physiques qui permettent d'accéder tout type de mémoire (mémoire ROM/RAM, mémoire I/O, registres, mémoires externes, etc.).

Il faut noter que le plan d'adressage est généralement linéaire et peut être *contigu* ou *discontigu* (il existe des plages d'adresses invalides qui ne sont pas utilisées). De plus, certaines plages d'adresses peuvent être dynamiquement reconfigurées par le processeur et correspondre à des périphériques différents (notamment au démarrage où certaines adresses peuvent référencer une ROM, puis changer de *mappage* pour référencer une RAM ou une mémoire de stockage secondaire comme de la mémoire *flash*).

En revanche, une adresse ne peut référencer plusieurs octets à la fois. Chaque adresse référence un et un seul octet (*byte*) de la mémoire; on dit que la mémoire est **byte-adressable**.

La capacité d'adressage d'un système est directement liée à la taille des registres du processeur; en effet, lors d'un transfert mémoire (même que d'un seul *byte*), le processeur utilise les registres pour stocker l'adresse intervenant dans l'accès mémoire. C'est pourquoi la taille d'une adresse ne peut dépasser la taille d'un registre: une architecture 32 bits comporte des registres 32 bits, c-à-d qu'une adresse peut être codée sur 32 bits au maximum, et peut donc aller de **0x0** à **0xffff'ffff** ( $2^{32}-1$ ), ce qui correspond à une zone de **4 Go**.

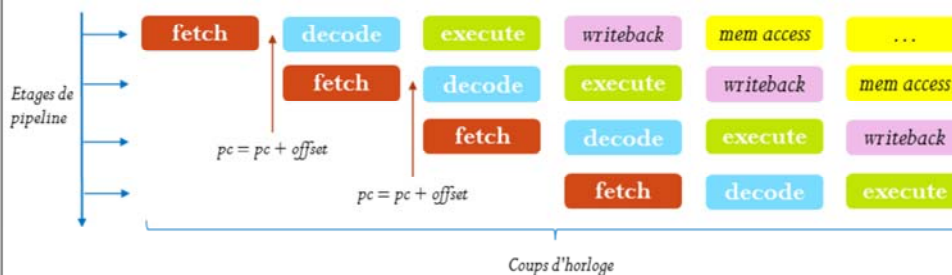
## Architectures des processeurs (7/7)

- Cycle de traitement d'une instruction (*fetch-decode-execute*)



- Exécution en *pipeline*

- Le processeur exécute plusieurs cycles en parallèle (plusieurs étages)



16

Cours ASM - Institut REDS/HEIG-VD - Introduction

Un aspect important des processeurs qui aura un impact sur le développement d'un programme en assembleur concerne la parallélisation en étage de plusieurs cycles de traitement; chaque cycle est associé à un étage. En effet, pour chaque instruction d'un programme, le processeur doit:

- 1) récupérer l'instruction en mémoire (*fetch*). Pour cela, il doit accéder l'instruction stockée à une adresse placée dans un registre de type PC (*Program Counter*)
- 2) décoder l'instruction à partir de la valeur stockée en mémoire.
- 3) exécuter l'instruction
- 4) effectuer divers traitements internes (réécriture dans un registre, mise à jour de la mémoire, etc.)

A chaque coup d'horloge, chaque étage exécute une étape du cycle en parallèle; il s'agit de la technique de *pipeline*.

Les mécanismes de *pipeline* ont un impact non négligeable sur le développement d'un code assembleur. En particulier, l'utilisation du registre PC nécessite une attention particulière. Il faut comprendre que sa valeur durant l'exécution d'une instruction a été incrémentée plusieurs fois (2 fois selon l'exemple ci-dessus) depuis l'opération *fetch*, autrement dit le PC est en avance de 8 *bytes* lors de l'exécution de l'instruction, si les instructions sont codées sur 32 bits.

L'utilisation d'une instruction de type *nop* (*no operation*) peut s'avérer utile pour vider le *pipeline* et anticiper une mauvaise lecture d'instruction (fin de code par exemple).



## Programmation système en C (1/7)

- Synthèse des types de données en C

Type de donnée	Signification	Taille	Plage de valeurs standardisée
char	Caractère	8 bits / 1 byte	-128 à 127
unsigned char	Caractère non signé	8 bits / 1 byte	0 à 255
short int	Entier court	16 bits / 2 bytes	-32 768 à 32 767
unsigned short int	Entier court non signé	16 bits / 2 bytes	0 à 65 535
int	Entier	32 bits / 4 bytes	-2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	32 bits / 4 bytes	0 à 4 294 967 295
long int	Entier long	32 bits / 4 bytes	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	32 bits / 4 bytes	0 à 4 294 967 295
long long	Entier long 64 bits	64 bits / 8 bytes	-9 223 372 036 854 775 807 à +9 223 372 036 854 775 807
unsigned long long	Entier long 64 bits non signé	64 bits / 8 bytes	0 à 18 446 744 073 709 551 615 (0xFFFFFFFFFFFFFFFF)
float	Flottant (réel)	32 bits / 4 bytes	$3.4 \cdot 10^{-38}$ à $3.4 \cdot 10^{38}$
double	Flottant double	64 bits / 8 bytes	$1.7 \cdot 10^{-308}$ à $1.7 \cdot 10^{308}$

17

Cours ASM - Institut REDS/HEIG-VD - Introduction

Le langage C comprend de nombreux types de nombres entiers, occupant plus ou moins de bits. La taille des types n'est que partiellement standardisée: le standard fixe uniquement une taille minimale et une magnitude minimale. Les magnitudes minimales sont compatibles avec d'autres représentations binaires que le complément à deux, bien que cette représentation soit presque toujours utilisée en pratique. Cette souplesse permet au langage d'être efficacement adapté à des processeurs très variés, mais elle complique la portabilité des programmes écrits en C.

Chaque type entier a une forme "signée" (*signed*) pouvant représenter des nombres négatifs et positifs, et une forme "non signée" (*unsigned*) ne pouvant représenter que des nombres naturels. Le type char, généralement utilisé pour représenter un caractère, est un type entier comme les autres, si ce n'est que, selon l'implémentation, il équivaut à *signed char* ou à *unsigned char*.

Le standard impose les relations suivantes:

$$\begin{array}{cccccc}
 \text{sizeof(char)} & \leq & \text{sizeof(short)} & \leq & \text{sizeof(int)} & \leq & \text{sizeof(long)} & \leq & \text{sizeof(long long)} \\
 \geq 8 & & \geq 16 & & \geq 16 & & \geq 32 & & \geq 32
 \end{array}$$

(source: Wikipedia, 2012)

## Programmation système en C (2/7)

- Variables & pointeurs
  - Stockage en mémoire
  - Accès mémoire

```
int var;          /* déclaration d'une variable (réservation mémoire) */  
int *pvar;        /* déclaration d'un pointeur (réservation mémoire) */  
pvar = &var;      /* écrit l'adresse de la variable dans le pointeur */  
*pvar = 3;        /* écrit la valeur 3 dans la variable (déréférencement) */
```

- Encodage des entiers (32 bits)

- Représentation *little-endian*
- Représentation *big-endian*

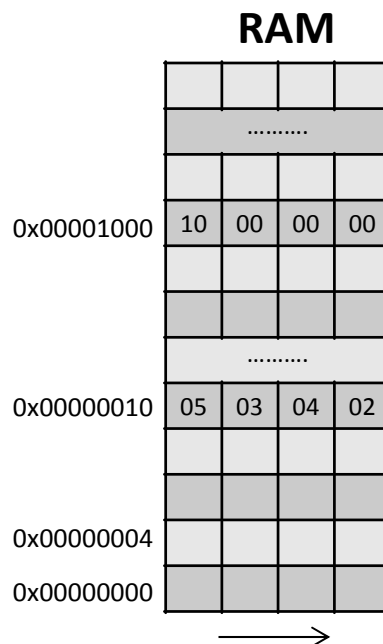
Encodage de 0x01020304

04	03	02	01
01	02	03	04

18

Cours ASM - Institut REDS/HEIG-VD - Introduction

La représentation d'une variable en mémoire est illustrée sur le schéma ci-dessous. On suppose que la variable *var* est plantée à l'adresse **0x00000010** et la variable *pvar* est plantée à l'adresse **0x00001000**.



## Programmation système en C (3/7)

- Exemples de manipulation des pointeurs

```
int main(int argc, char **argv) {

    unsigned int var;
    unsigned int *ptr_var;
    unsigned char *ptr_c; /* Accès à la variable par valeur */

    var = 0x10ab;
    printf("valeur %d\n", var); /* valeur 0x10ab en décimal */
    printf(" (hex) %x\n", var); /* valeur 0x10ab en hexadécimal */

    /* Accès à la variable par pointeur/référence */
    ptr_var = &var;
    ptr_c = ptr_var;

    printf("valeur %x\n", *ptr_c) ; /* affichage ? */
    ptr_c++;
    printf("valeur %x\n", *ptr_c) ; /* affichage ? */

    printf("taille ptr_var: %d et ptr_c: %d\n", sizeof(ptr_var), sizeof(ptr_c));

}
```

19

Cours ASM - Institut REDS/HEIG-VD - Introduction

Le code ci-dessus montre l'utilisation des opérateurs "&" et "\*" pour accéder à l'adresse d'une variable et modifier son contenu à partir de son adresse en mémoire.

L'utilisation du symbole "\*" dans la partie déclarative sert à déclarer un pointeur; il s'agit d'une variable qui pourra contenir une adresse mémoire. A noter que le mot-clé *null* est utilisé en C pour une adresse égale à 0 (*#define null (void \*) 0*).

Il ne faut pas oublier que la déclaration d'un pointeur (comme toute autre variable) **ne suffit pas à initialiser** la variable et que celle-ci peut contenir à priori n'importe quelle valeur.

Le symbole "\*" utilisé dans le corps d'une fonction permet d'accéder au contenu se trouvant à l'adresse stockée dans la variable (pointeur). On parle aussi de **déréférencement**. C'est à ce moment que le type du pointeur intervient afin que le compilateur sache quel quantité de données doit être transférée (1, 2, ou 4 *bytes* typiquement).

Le symbole "&" permettra de récupérer l'adresse d'une variable. On notera qu'un pointeur est traité en C comme tout autre variable, à savoir qu'il est bien sûr possible de récupérer l'adresse d'une variable contenant déjà une adresse! (adresse d'un pointeur).

## Programmation système en C (4/7)

- Utilisation du mot-clé *volatile*
  - 3 types de variable concernée
    - **Registres** de périphériques
    - Variables **globales** modifiées par une **routine d'interruption**.
    - Variables **globales** dans une applications **multitâches**.

```
volatile unsigned int *ptr;

int main(int argc, char **argv) {
    ptr = (unsigned int *) 0xc0001345;

    /* Le contenu à l'adresse 0xc0001345 peut éventuellement changer !... */
    printf("%x\n", *ptr);
}
```

Le compilateur C cherchera en principe à optimiser le code et fera tout pour minimiser les accès mémoire; cela se traduit par l'utilisation de variable en mémoire *cache* ou encore en modifiant l'ordre d'exécution des instructions.

Si une variable ou un emplacement mémoire est sujette à être modifié entre deux accès, l'utilisation du mot-clé ***volatile*** dans la déclaration de la variable ou du pointeur indiquera au compilateur d'effectuer inconditionnellement l'accès mémoire, et de ne pas avoir recours à une mémoire *cache* à des fins d'optimisation.

Typiquement, les pointeurs associés à des adresses I/O doivent être déclarés avec *volatile* afin d'éviter des mauvaises surprises lors de l'exécution.

## Programmation système en C (5/7)

- Accès aux registres de périphériques et mémoires externes
  - Accès **16 ou 32 bits**
  - Valeur **non signée** (positive ou nulle)
  - Attention aux **mémoires caches** entre le processeur et le périphérique

```
#define GPIO_R1      0x20af630

#define GPIO_R1_VAL  *((volatile unsigned int *) GPIO_R1)

int main(int argc, char **argv) {

    volatile unsigned int *gpio_r1;
    unsigned int temp;

    *gpio_r1 = 0x11b67;

    temp = GPIO_R1_VAL;

    *gpio_r1 = 0x22;

    GPIO_R1_VAL = 0xffffffff;

}
```

21

Cours ASM - Institut REDS/HEIG-VD - Introduction

L'utilisation de termes prédéfinis avec la clause *#define* est vivement encouragée lorsque l'on doit accéder la mémoire à des adresses fixes. C'est le cas typiquement pour les registres de périphériques.

Une bonne pratique consiste à définir l'adresse à l'aide des *#define* mais de procéder au déréférencement à l'aide d'une variable de type pointeur. Si l'on souhaite toutefois effectuer directement le déréférencement dans le *#define*, il est important de bien nommer les constantes, à savoir de faire la différence entre *adresse* et *valeur*.

Par ailleurs, il ne faut pas oublier non plus de **caster les valeurs numériques en adresse**, c-à-d en *(unsigned int \*)* par exemple. Par défaut, le compilateur considère toute valeur entière comme des valeurs de type *int*.

## Programmation système en C (6/7)

- Arithmétique binaire (opérateurs &, |, ^, ~, <<, >>)

```
#define UART_REG_ADDR 0x020af630 /* registre de l'interface série */
#define UART_M1      0x00000002 /* masque pour le bit no 1 du registre */
#define UART_M2      0x00000038 /* masque pour les bits no 3-4-5 du registre */

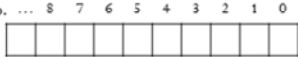
int main(int argc, char **argv) {
    volatile unsigned int *uart_reg;

    uart_reg = (unsigned int *) UART_REG_ADDR;

    *uart_reg |= UART_M1;          /* Met à 1 le bit no 1 (set) */
    *uart_reg &= ~UART_M1;        /* Met à 0 le bit no 1 (clear) */
    *uart_reg ^= UART_M1;          /* Inversion du bit 1 (toggle) */

    if (*uart_reg & UART_M1 != 0) { ... } /* Test le bit 1 */

    /* Ecrit la valeur 6 dans le champ de bits spécifiés dans UART_M2 */
    *uart_reg = (*uart_reg & ~UART_M2) | (0x6 << 3);
}
```



22

Cours ASM - Institut REDS/HEIG-VD - Introduction

Dans le langage C, il est possible de manipuler les valeurs au bit près grâce aux opérateurs arithmétiques et logiques. Ces opérateurs possèdent généralement une équivalence directe avec des instructions en assembleur réalisant les mêmes opérations.

Il s'agit notamment du ET logique (&), OU logique (|), XOR logique (^), et NOT (~). On rajoutera également les opérateurs de décalages binaires vers la droite (>>) et vers la gauche (<<).

Tous ces opérateurs agissent sur des valeurs correspondant aux types de base (*scalaires*) du langage C.

## Programmation système en C (7/7)



- Un registre (32 bits) de configuration d'une interface UART se trouve à l'adresse `0xF0000004`.

⇒ Ecrivez un code C permettant d'effectuer les opérations suivantes:

- Mettre à 0 le bit 16 du registre de configuration.
- Mettre à 1 le bit 19 du registre de configuration.
- Inverser l'état des bits 17 et 18 du registre de configuration.

## Références

- Pierre Fichoux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)
- ARM Architecture Reference Manual,  
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html>
- Intel® 64 and IA-32 Architectures Software Developer's Manual,  
<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>
- Documentation Intel, <http://developer.intel.com/design/Pentium4/documentation.htm>
- Assembleur Linux, <http://www.linuxassembly.org>
- The Art of Assembly, <http://webster.cs.ucr.edu>