Programmation C
Écriture d'un simulateur

ASR2 - Système

Semestre 2, année 2012-2013

Département informatique
IUT Bordeaux 1

Avril 2013

Fil conducteur

- Pour apprendre C, on écrit un simulateur pour le processeur fictif.
- Usage typique du C :
  - programmation à bas niveau
    - utilisation d'entiers 16 bits (signés / non signés)
    - manipulation de bits (extraction du code opération etc).
  - portabilité

Découpage du programme

- lire dans un fichier un “programme” écrit en hexadécimal,
- exécuter ce programme
- à la fin, afficher l'état de la machine

Allure générale du programme

```
struct Machine {
    .... M[256];
    .... ACC;
    .... PC;
    bool HALT;
};

int main(int argc, char **argv)
{
        struct Machine m;

        charger_fichier  (& m, argv[1]);
        lancer_execution (& m);
        afficher_etat    (& m);

        return EXIT_SUCCESS;
}
```

Structures

- Les structures regroupent plusieurs champs

```
struct Date {
  int jour;
  int mois;
  int annee;
};

struct Personne {
  char nom[30];
  char prenom[30];
  struct Date naissance;
};
```

Structures (suite)

- = classe C++, sans méthodes, et avec tous les champs publics.
- notations pointée "." et flèche "->"

```
struct Date bastille;

bastille.jour =   14;
bastille.mois =    7;
bastille.annee = 1789;

struct Personne * musicien;  // pointeur

strcpy (musicien -> prenom, "kevin");
strcpy (musicien -> nom   , "ayers");

(musicien -> naissance).jour   = 16;
(musicien -> naissance).mois   = 8;
(musicien -> naissance).annee  = 1944;
```

Les types

```c
#include <stdint.h>
#include <stdbool.h>

struct Machine {
  uint16_t M[256];
  int16_t  ACC;
  uint16_t PC;
  bool     HALT;
};
```

`HALT`	indique si la machine tourne	`bool`
accumulateur	un entier de 16 bits	`int16_t`
compteur de programme	entier de 16 bits, positif ou nul	`uint16_t`
mémoire	256 mots de 16 bits	

Types entiers

- les types de base
  - `int`, `char`
  - modificateurs : `short`, `long`
  - `signed`, `unsigned`

  dont la taille dépend de l'implémentation
- Les types standards, pour la portabilité :
  - de taille exacte : signés `int8_t`, `int16_t`, `int32_t`, `int64_t` et non signés `uint8_t` ...
  - de taille minimum : `int_least8_t`, ...
  - les représentations efficaces (en temps) : `int_fast8_t`, ...

La *représentation exacte* n'existe pas forcément (exemple microcontrôleur 18 bits).

Bilan provisoire

Vous connaissez

- les structures
  - définition
  - différence avec les classes
  - notations pointée et fléchée
- les types entiers
  - types de base : int / char / long / short / signed / unsigned
  - types standards, taille exacte, taille minimum, représentation efficace.

Le langage C permet de préciser la représentation des données : langage de bas niveau.

Retour à l'écriture du simulateur

Développement incrémental : on procède par étapes

1. ajout de "stubs" (fonctions bouchon) pour que le source soit accepté à la compilation
2. écriture et test de chaque fonction, dans l'ordre de facilité :
   1. affichage de l'état
   2. chargement du programme
   3. exécution

Fonctions bouchon : ne font rien (à part afficher un message !)

```
int foo (float bar)
{
    printf("Appel de %s\n", __PRETTY_FUNCTION__);
}
```

Travail : complétez le code par des stubs

```c
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

struct Machine {
   uint16_t M[256];
   int16_t  ACC;
   uint16_t PC;
   bool     HALT;
};
```

```c
// ....              ....
// .... STUBS ?      ....
///....              ....
```

```c
int main(int argc, char **argv)
{
   struct Machine m;

   charger_fichier   (&m,
                      argv[1]);
   lancer_execution  (&m);
   afficher_etat     (&m);

   return EXIT_SUCCESS;
}
```

Travail : programmer l'affichage de l'état de la machine

Exemple de présentation :

```
ADR      0     1     2     3     4     5     6     7     8     9     a     b
    +-------------------------------------------------------------------------
00  |  0160  43da  7fff  0000  34b0  43d7  7fff  0000  34b0  43d7  7fff  0000  3
10  |  aaa8  ec3b  7f62  0000  0003  0000  0000  0000  4e2e  f63d  0000  0000  7
.....
e0  |  0000  0000  0000  0000  04c3  0040  0000  0000  0000  0000  0000  0000  0
f0  |  bac0  ebe4  7f62  0000  0880  0040  0000  0000  0000  0000  0000  0000  0

Registres     ACC [hex=3710 dec= 14096]     PC [43d7]     HALT [0]
```

Mémoire : 16 lignes de 16 nombres de 4 chiffres en hexadécimal

```
pour ligne de 0 à 15 faire
  | pour colonne de 0 à 15 faire
  |   | afficher  M[ 16*ligne + colonne ]
  | sauter à la ligne
```

Formats à utiliser : `%x`, `%4x`

Chargement du programme

- Programme = suite de mots de 16 bits
- 1 mot = 4 chiffres hexadécimaux
- chargement à partir de l'adresse 0

Exemple :

```
0007   5004 3005
C000 0006
0000
```

Traduction de

```
  loadi 7    # 0007
  add   A    # 5004
  store B    # 3005
  halt  0    # C000
a word  6    # 0006
b word  0    # 0000
```

Lecture dans un fichier

Vous connaissez déjà `scanf( format, adr....)` pour lire sur le terminal.

- équivalent : `fscanf( fichier, format, adr....)`
- où `fichier` est un "FILE *".

```
FILE * f;
int nombre, somme = 0;

f = fopen("monfichier.txt", "r");
while ( fscanf(f, "%d", & nombre) == 1) {
    somme += nombre;
};
fclose(f);
printf("La somme vaut %d\n", somme);
```

Fichiers de haut niveau

- ouverture par `fopen(chemin, mode-d'accès)`
  - retourne un `FILE *`
  - modes : `"r"` en lecture, `"w"` en écriture, ...
- lecture par `fscanf(fichier, format, adr...)`
  - retourne le nombre d'éléments lus avec succès
- écriture par `fprintf(fichier, format, ....);`
- fermeture par `fclose(fichier);`

Travail : écrire la lecture d'un programme

Fichier "prog1.hex"

```
0007  5004 3005  C000 0006 0000
```

Portabilité pour lire des mots de 16 bits en hexadécimal, utiliser le format portable `"%" SCNx16`, défini dans `inttypes.h`

Exemple de programme de lecture

```
FILE * f = fopen("monfichier.txt", "r");
if ( f == NULL ) {
    ... // fichier absent ?
}
uint16_t nombre;
while ( fscanf(f, "%" SCNx16, & nombre) == 1) {
    ....
};
fclose(f);
```

Bilan

Vous savez maintenant

- ouvrir un fichier
- y lire des données
- utiliser les formats de lecture adaptés aux types standards

Ecrire : c'est pareil.

Exécution du programme

Pour lancer l'exécution :

```
initialiser les registres :  HALT = faux, PC = 0 ...

tant que non HALT
   | aller chercher l'instruction en M[PC]
   | la décomposer en  code opération + opérande
   |
   | selon le code opération
   | si 1 (load)  =>   Acc = M[opérande], PC=PC+1
   | si 3 (store) =>   M[opérande] = ACC, PC=PC+1
   | ....
   | si 12 (halt) =>   HALT = vrai
```

Énumérations

Avec une enumération

```
enum Code {
   LOADI, LOAD,  LOADX,
   STORE, STOREX,
   ADD,   SUB,   JMP,
   JNEG,  JZERO,
   JMPX,   CALL, HALT
};
```

on pourra utiliser des constantes

```
enum Code code;
...
switch (code} {
case LOADI:
      ....
case LOAD:
      ....
default:
      ...
};
```

Décomposition d'une instruction : opérande

- une instruction = 16 bits, non signé

0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0

- pour obtenir l'opérande (12 bits de droite) : masquer

```
uint_16   avant =  0x3005;
uint_16   apres =  avant & 0x0FF;
```

avant	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1
et 0x0FF	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
après	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

- Opération : *expression entière* & *expression entière*
- Bitwise operations :
  - binaires : et (&), ou (|), ou-exclusif (^),
  - unaire : complément (~).

Décomposition d'une instruction : code opération

- Pour obtenir le code opération : décaler de 12 bits vers la droite

```
uint_16   avant =  0x3005;
uint_16   apres =  avant >> 12;
```

avant	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1
après	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

- Décalages :
  - *expression entière* >> *nombre de positions vers la droite*
  - *expression entière* << *nombre de positions vers la gauche*
- Propagation de signe pour le décalage à droite des entiers signés,
  exemple : `-1 >> 1` donne -1 111...1111

Travail : décomposition

Programme interactif qui lit des instructions (nombres 16 bits hexadécimaux), et les décompose en code opération + opérande :

```
int main( ...) {
  for(;;) {
    uint16_t instruction;
    printf("Instruction : ");     // lecture
    if (scanf( ..... instruction) != 1)
      break;
    uint16_t code    = .....      // décomposition
    uint16_t operande = .....
                                  // affichage
    printf("code = ..... operande = .....\n",
           code, operande);
  }
  return 0;
}
```

travail : petit supplément

Ajoutez l'affichage des mnémoniques, en utilisant un tableau

```
char * mnemoniques[16] = {
    "loadi", "load", "loadx",
    ...
    "halt", "illegal13", "illegal14", "illegal15"
};
```

Exécution du programme : écriture en C

```
void lancer_execution(struct Machine * m)
{
  // initialiser les registres
  while ( ! m -> HALT ) {
     uint16_t instruction = m->M [ m->PC ];
     enum Code code       = ...
     uint16_t operande    = ...
                                  // tracer instruction
     switch(code) {
     case LOAD :
                 ...
                 m->PC ++;
                 break;
     case HALT :
                 m -> HALT = true;
                 break;
    }

}
```

Implémentez `load`, `store`, `add`, et `halt`.

Cas de l'instruction `loadi`

Rôle : charger la valeur de l'opérande dans l'accumulateur.
Mais

- code opération sur 4 bits
- l'opérande est codé sur 12 bits
- l'accumulateur un mot de 16 bits.

Exemples

- `loadi 42` = 0x002A, charge 0x002A
- `loadi -1` = 0x0FFF, charge 0xFFFF

Extension de signe

Pour les nombres négatifs

- -1 codé $\boxed{1111\ 1111\ 1111}$ en binaire
- loadi -1 : $\boxed{0000\ 1111\ 1111\ 1111}$ en binaire
- pour amener -1 dans l'accumulateur, il faut donc étendre le nombre signé 12 bits sur 16 bits.
- propager le bit de signe (5 ième) dans les 4 premiers.

avant	0000	Syyy	zzzz	tttt
après	SSSS	Syyy	zzzz	tttt

Propagation de signe 12 à 16 bits : comment faire

On utilise

- un décalage à gauche pour amener le bit de signe à gauche
- un changement de type pour transformer en nombre signé
- un décalage à droite du nombre signé

expression	type	contenu en binaire
`opérande`	uint16_t	`0000 Syyy zzzz tttt`
`opérande << 4`	uint16_t	`Syyy zzzz tttt 0000`
`(int16_t) (opérande << 4)`	int16_t	`Syyy zzzz tttt 0000`
`((int16_t) (opérande << 4)) >> 4`	int16_t	`SSSS Syyy zzzz tttt`

Conversions de type (typecast)

- `float quotient = (float) num / den;`

Bilan

Vous connaissez maintenant

- les opérations “bit-à-bit”
- les décalages
- les conversions de type

qui permettent à travailler “à bas niveau” sur les données en mémoire

Travail : complétez le simulateur

1. `loadi`
2. `sub`
3. `jmp`, `jzero`, `jneg`,
4. ...

Allocation dynamique

Objectif

- simulateur avec taille de mémoire variable

```
struct Machine {

  uint16_t *M;
  int      memsize;

  int16    ACC;
  uint16_t PC;
  bool HALT;
};
```

- l'allocation mémoire se fait par `malloc(nombre d'octets)`,
- la libération, par `free`

Allocation dynamique (suite)

Utilisation

- l'allocation mémoire par `malloc(nombre d'octets)`,

```
#include <stdlib.h>

m->memsize = 256;
m->M       = malloc ( m->memsize * sizeof(uint16_t) );
```

sizeof() indique la taille d'un type / d'une variable

- la libération par `free(pointeur)`

```
free( m->M );
```

- il est possible redimensionner (en recopiant)

```
void *realloc(void *ptr, size_t size);
```

Bilan général

A travers le développement d'un exemple typique :

- types, instructions
- chaines
- utilisation des pointeurs
- structures, énumérations
- programmation à bas niveau (manipulation de bits)
- allocation dynamique

Il reste quelques autres aspects à voir

- unions
- utilisation du préprocesseur
- fichiers à bas niveau
- bibliothèque Unix
- ...