

Langage C : écriture d'un simulateur

M. Billaud

4 avril 2013 - V2

Résumé

- Comment programmer en C un simulateur pour un processeur. On veut écrire un simulateur qui
- lit dans un fichier un "programme" écrit en hexadécimal, destiné à un processeur fictif,
 - exécute ce programme,
 - affiche le contenu de la mémoire à l'issue de ce programme.

Table des matières

1	Décomposition	2
1.1	Le programme principal	2
1.2	Structures de données : la machine	2
1.3	Fonctions	3
1.4	Makefile	3
2	Affichage de l'état de la machine	3
2.1	Objectif	3
2.2	Exemple	3
2.3	Code	4
3	Lecture des données	4
3.1	Objectifs	4
3.2	Préliminaires : lecture dans un fichier	5
3.2.1	Lecture sur le terminal	5
3.2.2	Lecture d'un fichier	5
3.3	Lecture dans le simulateur	6
3.4	Code	7
4	L'exécution	7
4.1	Objectifs	7
4.2	Rappel : les instructions	7
4.3	Technique : Décomposer une instruction	8
4.4	Technique : extension de signe	9
4.5	Le code	9

1 Décomposition

1.1 Le programme principal

Le programme principal est très court :

```
1 //      Simulateur
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 #include "machine.h"
7 #include "entrees-sorties.h"
8 #include "execution.h"
9
10 int main(int argc, char **argv)
11 {
12     struct Machine m;
13     printf("*_Simulateur_v1.0\n");
14
15     if (argc != 2) {
16         fprintf(stderr, "Usage :_%s_fichier\n", argv[0]);
17         return EXIT_FAILURE;
18     }
19     charger_fichier(&m, argv[1]);
20     lancer_execution(&m);
21     afficher_etat(&m);
22
23     printf("*_Bye.\n");
24     return EXIT_SUCCESS;
25 }
```

On y voit différentes particularités de C

1. la déclaration de la variable `m`, qui est du type `struct Machine`, et qui représente l'état de la machine, c'est-à-dire le contenu de la mémoire et des différents registres. En C, il n'y a pas de classes, mais des *structures* qui regroupent simplement des champs.
2. l'écriture sur le terminal, par la sortie standard, se fait par la fonction `printf()` qui utilise comme premier paramètre une "chaîne de format", dans laquelle `\n` figure le caractère de saut de ligne.
3. on voit cette même notion de format dans l'écriture sur la sortie d'erreur (`stderr`), qui se fait par

`fprintf(stderr, format, expressions...);`

Dans le format, `%s` indique où et comment il faut afficher la variable `chemin`. Ici c'est une chaîne : `s = string`.

4. `charger_fichier()`, comme son nom l'indique, a pour vocation de charger le contenu d'un fichier "exécutable" dans la mémoire de la machine `m`. En C++, on ferait un passage par référence, qui n'existe pas en C, on passe donc comme paramètre l'adresse de la structure `m`. La fonction `charger_fichier()` recevra donc un *pointeur*.

1.2 Structures de données : la machine

Une machine comporte 4 champs : un tableau de mots de 16 bits représentant la mémoire, l'accumulateur et le compteur de programme, et un booléen qui indique si elle est arrêtée.

Commentaires

1. nous utilisons les types `int16_t` et `uint16_t` qui codent des entiers, respectivement avec et sans signe, sur 16 bits. Ces types sont déclarés dans `stdint.h`.
2. de même les booléens sont définis dans `stdbool.h`
3. le langage C dispose par ailleurs d'une variété de types standard : `char`, `int`, `float`, `double`, etc. Mais pas de chaînes de caractères, qui sont simplement des tableaux de `char`, la fin de la chaîne étant marquée par le caractère nul `'\0'`.
4. comme en C, les `char` sont simplement de petits entiers, codés sur un octet.

1.3 Fonctions

Pour l'instant les fonctions de chargement du programme, d'exécution et d'affichage de l'état de la machine se réduisent à des "*stubs*" qui sont présents pour que la compilation se passe bien, mais ne font rien d'utile.

1.4 Makefile

Enfin, il n'est pas de projet décent sans un bon petit Makefile, qui contient essentiellement

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2
3 simulateur : simulateur.o
```

2 Affichage de l'état de la machine

Commençons par le plus facile, et nous en aurons besoin ensuite pour vérifier que les données sont chargées correctement : l'affichage de l'état de la machine.

2.1 Objectif

La fonction `afficher_etat` affiche

- le contenu de la mémoire, sous forme de 16 lignes de 16 mots en hexadécimal (4 caractères)
- les contenus des registres et indicateurs ACC, PC, HLT

2.2 Exemple

Voici un fragment de l'affichage produit

```
ADR    0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
+-----+
00 | 0160 43da 7fff 0000 34b0 43d7 7fff 0000 34b0 43d7 7fff 0000 34b0 43d7 7fff 0000
10 | aaa8 ec3b 7f62 0000 0003 0000 0000 0000 4e2e f63d 0000 0000 782c ec1a 7f62 0000
....
e0 | 0000 0000 0000 0000 04c3 0040 0000 0000 0000 0000 0000 0000 08c5 0040 0000 0000
f0 | bac0 ebe4 7f62 0000 0880 0040 0000 0000 0000 0000 0000 0000 0530 0040 0000 0000
Registres    ACC [hex=3710 dec= 14096]    PC [43d7]
```

Ce qui montre que l'adresse `0xe4` contient la valeur `0x04c3`.

Avec une machine non initialisée, c'est normal de voir un peu n'importe quoi.

2.3 Code

```
1 void afficher_etat(struct Machine *m)
2 {
3     /* en-tête du tableau */
4     printf("ADR_");
5     for (int colonne = 0; colonne < 16; colonne++)
6         printf("uuuu%x", colonne);
7     printf("\nuuuu+");
8     for (int colonne = 0; colonne < 16; colonne++)
9         printf("———");
10    printf("\n");
11
12    /* vue de la mémoire sous forme de
13       16 lignes de 16 nombres de 4 chiffres hexa,
14       précédées par l'adresse en hexa */
15
16    for (int ligne = 0; ligne < 16; ligne++) {
17        int adresse = 16 * ligne;
18        printf("%02x_|_", adresse);
19        for (int colonne = 0; colonne < 16; colonne++) {
20            printf("%04x_", m->M[adresse + colonne]);
21        }
22        printf("\n");
23    }
24
25    /* les registres */
26    printf(" Registres :uuACC_| hex=%04x_| dec=%6d ]uuPC_|[%04x]uuHALT_|[%x]\n" ,
27           m->ACC, m->ACC, m->PC, m->HALT );
28 }
```

Remarques

1. le paramètre `m` est ici un *pointeur* vers une structure, les champs sont donc désignés par `m->ACC`, `m->PC`, `m->M` qui sont des raccourcis pour `(*m).ACC`, etc.
2. la spécification de format `%x` fait apparaître un nombre en hexadécimal, sur autant de caractères que nécessaire. Ici le nombre à afficher est entre 0 et 15, donc il n'occupera qu'un seul caractère.
3. plus loin, `%02x` (et `%04x`) demande un affichage sur 2 (ou 4) caractères au moins, avec éventuellement des zéros en tête si c'est nécessaire.
4. enfin, la spécification `%6d` demande une représentation décimale sur 6 chiffres, des espaces occupant les emplacements des zéros non-significatifs

3 Lecture des données

3.1 Objectifs

- remplir la structure de données qui représente la mémoire avec le contenu d'un fichier texte
- le fichier contient une suite de "mots" de 4 chiffres hexadécimaux.

3.2 Préliminaires : lecture dans un fichier

Avant d'aborder la lecture des données dans un fichier, revoyons comment lire sur le terminal.

3.2.1 Lecture sur le terminal

Nous employons `scanf()`, qui joue un rôle quasi-symétrique à `printf()`

`scanf(format , adresses ...) ;`

```
1  /*
2  *  lecture et écriture d'un nombre en hexadécimal
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main(int argc, char **argv)
9  {
10     unsigned int n;
11     do {
12         printf("hexa_u?_");
13         scanf("%x", &n);
14         printf("decimal_u=%d, _hexadecimal_u=%x\n", n, n );
15     } while (n!=0);
16     return EXIT_SUCCESS;
17 }
```

Remarques :

1. comme `printf()`, le premier paramètre est une spécification de format
2. par contre, les paramètres suivants sont les *adresses* des données à remplir. C'est une conséquence du seul mode de passage de paramètres disponible en C : le passage par valeur.
3. ici la lecture se fait en hexadécimal (format `%x`), ce qui implique que la donnée lue soit un entier non-signé, d'où la déclaration :

`unsigned int n;`

Pour cette démonstration, une variable de type `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` aurait aussi pu faire l'affaire.

3.2.2 Lecture d'un fichier

L'exemple suivant nous permettra de voir la lecture sur un fichier, à l'aide de

`fscanf(fichier, format , adresses ...) ;`

```
1  /*
2  *  lecture dans un fichier
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
```

```

7
8 /* lit et affiche des nombres */
9 void lire_nombres(char chemin[])
10 {
11     FILE *f;
12     f = fopen(chemin, "r");
13     if (f == NULL) {
14         fprintf(stderr, "Le fichier %s ne peut être ouvert\n", chemin);
15         return;
16     }
17     int nombre;
18     while ( fscanf(f, "%d", & nombre) == 1) {
19         printf ("%d ", nombre);
20     }
21     printf("\n");
22     fclose(f);
23 }
24
25 int main(int argc, char **argv)
26 {
27     if (argc != 2) {
28         fprintf(stderr, "usage : %s fichier\n", argv[0]);
29         return EXIT_FAILURE;
30     }
31     lire_nombres(argv[1]);
32     return EXIT_SUCCESS;
33 }

```

1. le premier argument de `fscanf()` est un fichier, plus précisément un "FILE *" qui sert à désigner un fichier ouvert.
2. la fonction `fopen()` ouvre un fichier, à partir de son *chemin d'accès* et d'un chaîne qui indique son *mode* d'ouverture. Ici "r" indique une ouverture en lecture (ce serait "w" pour une écriture). `fopen()` retourne le pointeur NULL si l'ouverture a échoué.
3. `scanf()` et `fscanf()` retournent un entier, le nombre d'éléments qu'elles ont réussi à lire. Dans ce programme, normalement c'est 1, sauf à la fin du fichier où c'est 0. Ce serait également 0 si le fichier contenait autre chose que des nombres. Ce qui explique la boucle de lecture "tant qu'on arrive à lire un nombre, faire ...".

En exécutant ce programme avec un fichier `donnees.txt` qui contient ceci,

```

12 34 5678
9 1011 121314
15

```

le déroulement produit, sans surprise :

```

$ ./lecture donnees.txt
12 34 5678 9 1011 121314 15

```

3.3 Lecture dans le simulateur

Pour en revenir notre simulateur, il y a une petite complication : d'une part nous voulons lire des `uint16_t`, et non des `unsigned int`, d'autre part toutes les machines ne sont pas semblables : certaines ont des entiers de 16 bits, d'autres de 32, ou de 64.

Nous voulons du code *portable* : c'est pour ça que nous avons précisé `uint16_t`, pour être sûrs de travailler avec des entiers codés sur 2 octets.

Pour la lecture, il faut en principe indiquer un format qui correspond à la nature des données. Sur une machine à mots de 32 bits, on lit un entier hexadécimal avec `%hx`, le `h` indiquant un demi-mot.

Mais cela dépend des machines : on se repose donc sur un fichier `inttypes.h` qui indique les bons formats pour les différents types disponibles sur cette machine.

En ce qui nous concerne, nous lirons nos mots de 16 bits avec la spécification

`"%" SCNx16`

qui concatène le caractère `"%"` et la chaîne qui va bien pour les entiers 16 bits en représentation hexadécimale. Ce qui nous donne :

3.4 Code

```
1 #include <inttypes.h>
2
3 void charger_fichier (struct Machine *m, char chemin [])
4 {
5     FILE *f;
6
7     f = fopen(chemin, "r");
8     if (f == NULL) {
9         fprintf (stderr, "Le_fichier_%s_ne_peut_tre_ouvert\n", chemin);
10        return;
11    }
12
13    int position = 0;
14    while ( fscanf(f, "%" SCNx16, & m->M[position]) == 1) {
15        position++;
16    }
17    printf("-_Fichier_%s_:_%d_mots.\n", chemin, position);
18
19    /* remplissage du reste avec des zeros */
20    while (position < 256) {
21        m->M[position++] = 0;
22    }
23    fclose(f);
24 }
```

4 L'exécution

4.1 Objectifs

La fonction d'exécution fait tourner le simulateur, en exécutant les instructions depuis la première (PC=0) jusqu'à ce que l'indicateur HLT passe à 1.

4.2 Rappel : les instructions

Rappelons que les instructions du processeur fictif sont sur 16 bits, dont les 4 premiers indiquent le code-opération. Nous n'allons implémenter que quelques instructions du processeur fictif :

Code	Mnémonique	Description
0	loadi <i>imm12</i>	chargement immédiat
1	load <i>adr12</i>	chargement direct
3	store <i>adr12</i>	rangement direct
5	add <i>adr12</i>	addition
C	halt 0	arrêt

Le programme suivant nous permettra de faire des tests. La traduction hexa est en commentaire.

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

4.3 Technique : Décomposer une instruction

Nous allons avoir besoin d'extraire le code instruction et l'opérande codés respectivement codés sur 4 et 12 bits dans une instruction représentée sur 16 bits.

Pour cela, on utilise les opérations C qui agissent sur des entiers considérés comme des vecteurs de bits (*bitwise operators*) :

- l'opération de *décalage à droite*, noté `>>`. Le code opération est dans les 4 premiers bits (sur 16), on le récupère en décalant le mot de l'instruction de 12 positions vers la droite
- l'opération *et "bit à bit"* : l'opérande s'obtient en faisant un "et" avec un masque qui contient 4 bits à 0 et 12 à 1, soit `000 1111 111 1111`, qui s'écrit `0x0FFF` en hexadécimal.

Voici un programme interactif qui met en oeuvre ces opérations

```
1  /*
2  * décomposition de mots
3  * lus sur l'entrée standard
4  */
5
6  #include <inttypes.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 int main(int argc, char **argv)
11 {
12     uint16_t mot;
13     while (scanf("%" SCNx16, & mot) == 1) {
14         unsigned int code = (mot >> 12);    /* décalage 12 bits à droite */
15         unsigned int operande = (mot & 0xfff); /* masquage */
16         printf("mot=%04x, code=%0x, operande=%0x\n",
17             mot, code, operande);
18     }
19     return EXIT_SUCCESS;
20 }
```

et son déroulement

```
$ ./bits < prog2.txt
mot = 0007, code = 0, operande = 7
```



```

mot = 5004, code = 5, operande = 4
mot = 3005, code = 3, operande = 5
mot = c000, code = c, operande = 0
mot = 0004, code = 0, operande = 4
mot = 0000, code = 0, operande = 0

```

4.4 Technique : extension de signe

Avec l'instruction `loadi` se pose un petit souci. En effet, on souhaite légitimement avoir la possibilité de charger dans l'accumulateur des constantes négatives

```
load -1
```

or la constante -1 se code 1111 1111 1111 1111 sur 16 bits et ne rentre pas sur les 12 bits d'opérande d'une instruction.

Le parti pris est donc de coder les constantes sur 12 bits seulement, et l'instruction ci-dessus sera représentée (en hexadécimal) 0x0fff.

Lors de l'exécution du `loadi`, il faudra mettre cette valeur -1 (codée) sur 12 bits, dans l'accumulateur de 16 bits. Pour cela on procède à une *extension de signe*, en profitant du fait que le décalage à droite d'un entier *signé* propage le bit de signe vers la droite.

De façon détaillée :

- la valeur non signée sur 16 bits est décalée de 4 bits vers la gauche et placée dans un entier signé de 16 bits,
- l'entier signé est décalé de 4 bits vers la droite.

Ceci tient en une ligne, en utilisant la conversion en entier signé :

```
m->ACC = ((int16_t)(operande << 4)) >> 4;
```

Pour convertir explicitement une variable (ou le résultat d'une expression) dans un autre type, on la précède par le nom du nouveau type entre parenthèses.

```

int num, den;
...
float r;
r = ((float) num) / den.

```

Si on écrivait simplement `r = num / den`, il y aurait d'abord une division entière. L'affectation déclencherait alors une conversion implicite de ce quotient en flottant.

En procédant comme indiqué, `num` est d'abord converti, ce qui provoque une division flottante.

4.5 Le code

Pendant l'exécution, nous allons faire "tracer" le déroulement de la simulation, et pour cela nous utilisons une table des mnémoniques.

Il convient également de tester la validité des accès à la mémoire : une fonction s'en charge, et en cas d'erreur imprime un message circonstancié, et met la machine à l'arrêt.

```

1 char * mnemoniques[] = {
2     "loadi",      "load",      "loadx",
3     "store",      "storex",
4     "add",        "sub",
5     "jmp",        "jneg",      "jzero",
6     "jmpx",       "call",      "halt",
7     "illegal13", "illegal14", "illegal15"

```

```

8  };
9
10
11 bool valider_adresse(struct Machine *m,
12                     unsigned int adresse, char message[])
13 {
14     if (adresse < 256)
15         return true;
16     else {
17         printf("—PC=%x*ERREUR_ACCES_MEMOIRE_ADRESSE=%x—%s\n",
18             m->PC, adresse, message);
19         printf("—Machine_arretee.\n");
20         m->HALT=true;
21         return false;
22     }
23 }
24
25 void lancer_execution(struct Machine * m)
26 {
27     m->ACC = 0;
28     m->HALT = false;
29     m->PC = 0;
30
31     /* ready to go ! */
32
33     while ( ! m->HALT ) {
34         if (! valider_adresse(m, m->PC, "lecture_instruction"))
35             break;
36
37         uint16_t mot = m -> M[ m -> PC ];
38         unsigned int code = (mot >> 12);
39         unsigned int operande = (mot & 0xfff);
40
41         printf("—pc=%d_execution_de_%s_%d\n",
42             m->PC, mnemoniques[code], operande);
43
44         switch (code) {
45             case 0 : /* loadi */
46                 /* extension du signe de l'opérande 12->16 bits */
47                 m->ACC = ((int16_t)(operande << 4)) >> 4;
48                 m->PC ++;
49                 break;
50             case 1 : /* load */
51                 if (valider_adresse(m, operande, "operande_load")) {
52                     m->ACC = (int16_t)(m->M [operande]);
53                     m->PC ++;
54                 }
55                 break;
56             case 5 : /* add */
57                 if (valider_adresse(m, operande, "operande_add")) {
58                     m->ACC += (int16_t)(m->M [operande]);

```

```

59         m->PC ++;
60     }
61     break;
62     case 3 : /* store */
63         if (valider_adresse(m, operande, "operande_store")) {
64             m->M [operande] = (int16_t) m->ACC;
65             m->PC ++;
66         }
67         break;
68     case 0xC : /* halt */
69         m->HALT = true;
70         break;
71     default :
72         printf("ERREUR : operation illegale '%x'\n", code);
73         break;
74     }
75     printf("ACC=%d\n", m->ACC);
76 }
77 }

```

et voici le compte rendu d'exécution du programme ci-dessus :

* Simulateur v1.0

– Fichier prog2.txt : 6 mots.

– pc=0 execution de loadi 7

ACC=7

– pc=1 execution de add 4

ACC=11

– pc=2 execution de store 5

ACC=11

– pc=3 execution de halt 0

ACC=11

ADR	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	0007	5004	3005	c000	0004	000b	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
10	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
20	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
30	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
40	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
50	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
60	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
70	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
80	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
90	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Registres : ACC [hex=000b dec= 11] PC [0003] HALT [1]

* Bye.