Sujet de contrôle – systèmes embarqués

J.-M Friedt, 6 janvier 2015

A society, most of whose members spend a great part of their time, not on the spot, not here and now and in the calculable future, but somewhere else, in the irrelevant other worlds of sport and soap opera, of mythology and metaphysical fantasy, will find it hard to resist the encroachments of those who would manipulate and control it.

A. Huxley, *Brave New World Revisited* (1958)

1 Opérations arithmétiques

Une grandeur physique x est mesurée au travers d'une instrumentation (par exemple une mesure de tension aux bornes d'une diode) qui fournit une mesure y tel que

$$y = \exp(-x/X_0)$$

avec $X_0 > 0$ une constante d'étalonnage fournie par le constructeur.

Nous désirons implémenter sur un système embarqué la mesure de y et informer l'utilisateur de la valeur de la grandeur physique x déduite de y. La gamme de fonctionnement de l'instrument est $x \in [1 : 100]$.

Nous cherchons à garantir l'exactitude du calcul et du résultat fourni. Dans un premier temps, nous considérons la mesure de y parfaite et cherchons à identifier le nombre de décimales sur X_0.

1. La mesure de y doit permettre de remonter à la grandeur physique x : quelle relation lie x à y ?
2. Exprimer l'incertitude sur X_0 en fonction de l'incertitude sur x recherchée. Application numérique : si X_0 vaut théoriquement 10π, alors combien de décimales de cette constantes sont nécessaires pour estimer x à 10^{-2} près ?

On s'intéresse maintenant à la résolution du convertisseur analogique-numérique permettant d'acquérir la grandeur que nous supposerons convenablement mise en forme pour exploiter pleinement la gamme de mesure de l'interface d'acquisition.

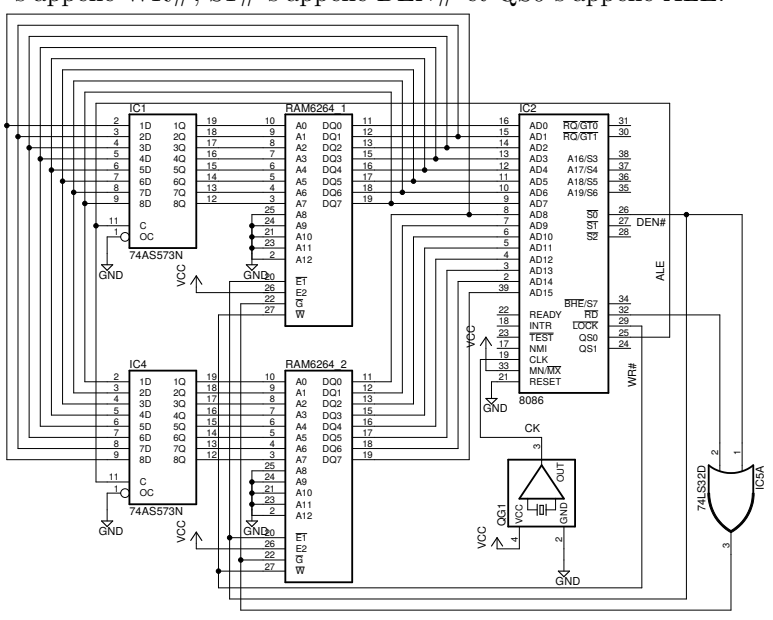
3. Reprendre le calcul précédent en tenant compte cette fois de l'incertitude sur X_0 – le coefficient d'étalonnage – *et* sur y, la mesure qui permettra de déduire la grandeur physique x.
4. En se rappelant du sens physique du logarithme en base N d'un nombre exprimé dans cette base, proposer un algorithme pour calculer le logarithme en base 2 de y (sachant que le logarithme en base N de y est $\ln(y)/\ln(N)$, le logarithme en n'importe quelle base nous permet de passer au logarithme dans une autre base en multipliant par une constante, ici égale à $\ln(2)/\ln(N)$). Ce calcul ne s'effectue que sur des entiers, et fournira donc en résultat l'entier par défaut le plus proche de $\ln_2(y)$.

2 Interfaçage de périphériques numériques

Le 8086 [1, chap.9] avait la propriété, tout comme le slot GBA de la Nintendo DS aujourd'hui, de multiplexer les fonctionnalités de certaines broches. Il s'agit d'une méthode classique de gestion des périphériques visant à limiter le nombre de broches ou de signaux sur les bus, au détriment de la bande passante puisque chaque broche ne peut effectuer qu'une fonction à la fois. Un signal additionnel informe de la fonction active. Dans le cas du 8086, des broches servaient tantôt de bus d'adresse, tantôt de bus de donnée. L'interfaçage avec un périphérique tel qu'une RAM nécessite donc de démultiplexer les signaux afin de respecter les chronogrammes du périphérique.

Dans cet exemple, nous nous proposons d'analyser la connexion minimale entre un processeur et des mémoires, qui ne tient ni compte de la nécessité d'avoir des périphérique de communication, ni d'avoir une mémoire non-volatile pour contenir le programme à exécuter. Une version complète de mini-ordinateur à base de 8086 est par exemple disponible à `http://rti.etf.bg.ac.rs/rti/ri4ms/literatura/vezbe/html/minimal8086.htm`. Le câblage d'un tel circuit devient rapidement ... complexe : `https://www.youtube.com/watch?v=sHKN7zDKV0g`.

Un schéma de câblage du 8086 qui se limite aux fonctionnalités minimales pour accéder à de la mémoire vive est proposé ci-dessous. On y notera que la nomenclature des broches du processeur sont celles du Maximum mode (multi-processeurs), alors que dans une architecture minimaliste telle que celle proposée ici, le Minimum mode suffit (MN/MX#=0). En mode minimum, la broche LOCK# s'appelle WR#, S1# s'appelle DEN# et QS0 s'appelle ALE.



Un chronogramme de l'accès aux bus du 8086 est fourni dans la datasheet [1] et reproduit ci-dessous. On s'y intéressera en particulier aux signaux ALE (*Address Latch Enable*) et aux signaux du bus AD.

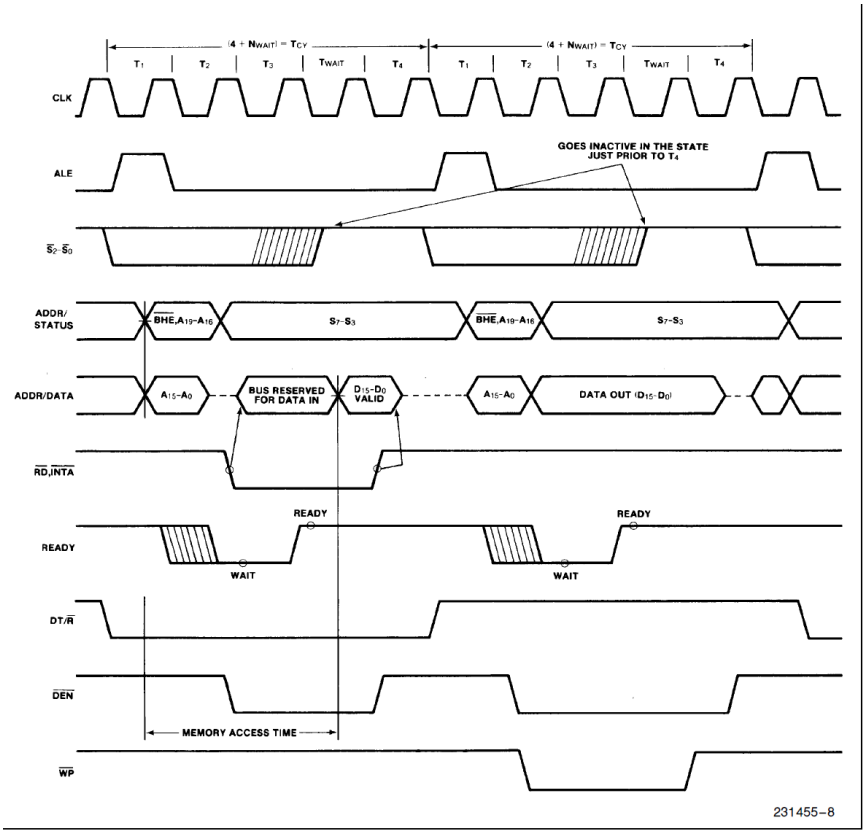


Figure 5. Basic System Timing

1. Que signifie le symbole # après un nom de signal ?
2. Identifier sur le schéma les bus de données et d'adresses en sortie du processeur (piège). Idem en entrée des RAMs.
3. En particulier, quelle est la taille du bus d'adresses en sortie du processeur ? Quelle plage d'adresses est ainsi accessible ?
4. Quelle est la taille du bus de données ?
5. Compte tenu du câblage des RAM, combien de bits du bus d'adresse sont effectivement utilisés ?
6. Combien d'octets sont alors accessibles sur cette implémentation ? sur quelle plage d'adresses ?

3 Modifier le flux d'exécution d'un logiciel

Soit l'exemple de programme trivial suivant (proposé sur deux colonnes) :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {int i;
6  initialise_i(&i);
7  if (i==1) printf("bloque'\n");
8     else printf("debloque'\n");
9 }
```

Ce programme nous autorise à afficher le message "débloque" si la variable `i` est à 0, et affiche par défaut "bloqué" sinon. Ce comportement fort désagréable est imposé par l'initialisation de la variable `i` dans la fonction `initialise_i()` au début du programme, que l'utilisateur n'a *a priori* pas le droit de modifier. Dans notre cas, il s'agit simplement de

```
1 void initialise_i(int* i) {*i=1;}
```

Notre objectif est de débloquer le programme, sans toucher au code source du logiciel (qui, dans un environnement propriétaire, ne nous est de toute façon pas fourni). Il faudra donc modifier le contenu de `i` en cours d'excécution.

Un programme, sur PC comme sur microcontrôleur, n'est qu'une succession d'instructions exécutées séquentiellement. Pour rappel, le jeu d'instructions du 8086, dont est issue la grande majorité des ordinateurs personnels actuels (PC dits compatibles IBM), est disponible à `http://matthieu.benoit.free.fr/cross/data_sheets/8086_family_Users_Manual.pdf`. Si nous pouvons interrompre l'exécution, ou modifier l'initialisation de `i`, nous serons en mesure d'afficher le message recherché (`débloque`). Supposons que le vendeur de ce logiciel ne nous fournisse que le binaire exécutable, et pas le code source : nous avons donc simplement en notre possession le binaire résultant de la compilation `gcc -g -o test test.c` (l'option de déverminage `-g` n'est pas nécessaire mais simplifie la démonstration), et éventuellement la bibliothèque dynamique que nous nommerons `bloque_debloque.so` issue de la compilation `gcc -shared -fPIC pas_bloque_debloque.c -o`

1. `http://www.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:8086-datasheet.pdf`

`pas_bloque_debloque.so` si la fonction `initialise_i()` n'a pas été liée de façon statique au binaire (vérifier par l'instruction `ldd` qui indique la liste des bibliothèques binaires nécessaires à l'exécution d'un programme). Trois méthodes de travail s'offrent à nous :

- Modification du binaire :
  - nous avons à notre disposition deux outils : `objdump` convertit le binaire en série d'instructions assembleur et fournit les opcodes correspondant,
  - `hexedit` [2] permet de modifier le fichier binaire `test`
- Exécution dans un *debugger* :
  - le GNU-debugger `gdb` permet d'exécuter un programme jusqu'à un point d'arrêt,
  - l'instruction `set variable` permet de modifier la valeur d'une variable au cours de l'exécution
- Modification de la bibliothèque dynamique :
  l'appel aux fonctions d'une bibliothèque dynamique (`.so` sous GNU/Linux, `.dll` sous MS-Windows) peut être court-circuitée, notamment au moyen de `LD_PRELOAD=bibliotheque.so` sous GNU/Linux.

On s'intéresse au premier cas :

1. Rappeler comment compiler le code source en un binaire exécutable sur PC.
2. Une fois ce binaire testé en l'exécutant, nous affichons le code assembleur correspondant par `objdump -dSt binaire` qui donne (noter l'utilisation de l'option `-g` lors de la compilation pour obtenir les symboles de déverminage)

```
int main()
{int i;
 80483fb:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80483ff:       83 e4 f0                and    $0xfffffff0,%esp
 8048402:       ff 71 fc                pushl  -0x4(%ecx)
 8048405:       55                      push   %ebp
 8048406:       89 e5                   mov    %esp,%ebp
 8048408:       51                      push   %ecx
 8048409:       83 ec 14                sub    $0x14,%esp
 initialise_i(&i);
 804840c:       83 ec 0c                sub    $0xc,%esp
 804840f:       8d 45 f4                lea    -0xc(%ebp),%eax
 8048412:       50                      push   %eax
 8048413:       e8 35 00 00 00          call   804844d <initialise_i>
 8048418:       83 c4 10                add    $0x10,%esp
 if (i==1) printf("bloque'\n");
 804841b:       8b 45 f4                mov    -0xc(%ebp),%eax
 804841e:       83 f8 01                cmp    $0x1,%eax
 8048421:       75 12                   jne    8048435 <main+0x3a>
 8048423:       83 ec 0c                sub    $0xc,%esp
 8048426:       68 f0 84 04 08          push   $0x80484f0
 804842b:       e8 a0 fe ff ff          call   80482d0 <puts@plt>
 8048430:       83 c4 10                add    $0x10,%esp
 8048433:       eb 10                   jmp    8048445 <main+0x4a>
```

   Identifier l'adresse de l'instruction et l'opcode qui induit le saut à la fonction qui affiche `bloqué`.
3. Comment modifier le code assembleur proposé dans ces lignes afin de sauter à la fonction qui affiche `débloqué` ?
4. Pour la seconde méthode proposée, sur quelle fonction pouvons nous placer un point d'arrêt lors de l'exécution du programme pour interrompre son exécution avant d'afficher `bloqué` ?

4 Questions de cours

1. Exprimer 0xdd en décimal.
2. Exprimer 254 en hexadécimal.
3. Quel est le pas de quantification (en tension) d'un convertisseur analogique-numérique de 8 bits de résolution de tension de référence 5,12 V ?
4. Comment afficher le contenu du sous répertoire `a` sous unix ?
5. Comment définir une variable non-signée codée sur 8 bits en C ?
6. Comment créer un tableau `tab` de 12 éléments entiers non-signés en C ?
7. Comment accéder au dernier élément de ce tableau ?
8. Quelle est la fonction principale appelée lors de l'exécution d'un programme C ?
9. Qu'affiche le programme

   ```
   int main() {char c[10]={1,2,3,4,72,'e','l','l','o',0};printf("%s\n",&c[4]);}
   ```

   ?
10. Proposer un chronogramme de la transmission de l'octet 0x55 en format 8N1 (8 bits de données, pas de bit de parité, 1 bit de d'arrêt) par liaison RS232. En quoi cet octet est-il spécial ?
11. Comment effectuer efficacement l'opération $N \times 15$ sans faire de multiplication ?

2. `https://packages.debian.org/search?keywords=hexedit` – CTRL-S pour rechercher un motif, F2 pour sauver le fichier modifié, CTRL-X pour quitter

Solutions

J.-M Friedt, 6 janvier 2015

1 Opérations arithmétiques

1. Calcul d'incertitude sur $y = \exp(-x/X_0)$: $dy = \left|\frac{\partial y}{dx}\right| + \left|\frac{\partial y}{dX_0}\right| = \frac{y \cdot dx}{X_0} + \frac{xy \cdot dX_0}{X_0^2}$

   Si la mesure est parfaite, $dy = 0$ donc il reste $dx = x\frac{dX_0}{X_0}$ ou $dX_0 = X_0\frac{dx}{x}$, avec $x \in [1 : 100]$ et $dx = 10^{-2}$ donc $dX_0 < X_0/10000$: X_0 s'exprime avec 4 décimales.

2. Application numérique

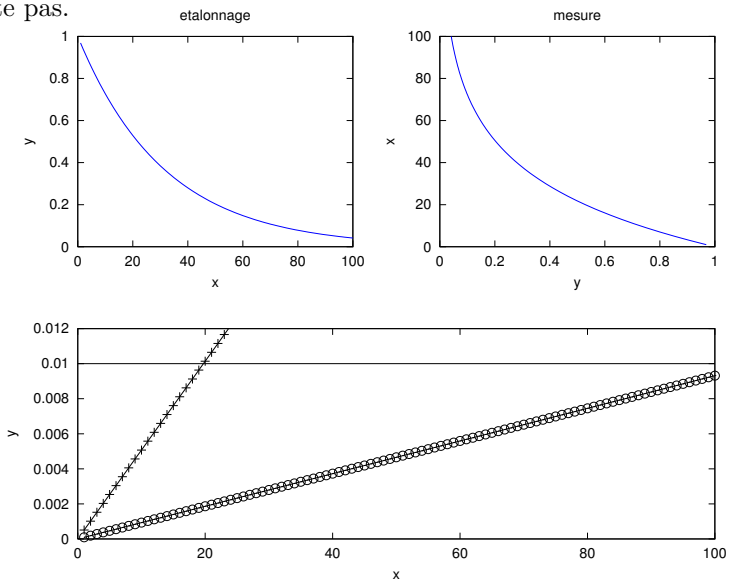
   La courbe avec les ronds indique le respect de la condition sur dX_0, la courbe avec les croix indique le résultat avec une décimale de moins que nécessaire. Le premier cas respecte le critère de résolution indiqué par la droite horizontale dans la gamme de mesure, le second ne le respecte pas.

```
X0=10*pi;dx=1e-2;x=[1:100];
y=exp(-x/X0);

subplot(221);plot(x,y);
 xlabel('x');ylabel('y');title('etalonnage')
subplot(222);plot(y,x);
 xlabel('y');ylabel('x');title('mesure')

dX0=X0*dx/max(abs(x))
X0e=floor(X0/0.003)*0.003
xe=-X0e*log(y);
subplot(212);plot(x,abs(x-xe),'ko-');hold  on
X0e=floor(X0e*10)/10
xe=-X0e*log(y);
subplot(212);plot(x,abs(x-xe),'k+-')

line([0 100],[dx dx]);axis([0 100 0 dx*1.2])
xlabel('x');ylabel('y')
```



3. $dy = \frac{y \cdot dx}{X_0} + \frac{xy \cdot dX_0}{X_0^2} \Rightarrow dX_0 = \frac{X_0^2}{xy}dy + \frac{X_0}{x}dx$. Si par exemple $dy/y = 10^{-3}$ (convertisseur 10 bits), alors l'application numérique donne $dX_0 = 10^{-3}\frac{X_0^2}{x} + 10^{-2}\frac{X_0}{x}$ avec $x \leq 100$ et $X_0^2 \simeq 10^3 \Rightarrow dX_0 \simeq 10^{-2} + 10^{-4}X_0 = 10^{-2} + 3 \cdot 10^{-3}$.

4. Le logarithme d'un nombre est le nombre de bits qu'il contient. On a donc simplement (seule la ligne 5 était demandée)

```
1 int main(int argc, char **argv)
2 {unsigned int v=129; // mot dont on veut ln2
3  unsigned int r = 0; // r=ln2(v)
4  if (argc>1) v=atoi(argv[1]);printf ("ln2(%d)=",v);
5  while (v >>= 1) {r++;}
6  printf("%d\n",r);
7 }
```

2 Périphériques du 8086

1. # signifie que le signal est actif au niveau bas : son état de repos est Vcc, et lorsque le signal se déclenche, il passe à la masse,
2. le 8086 a la propriété de multiplexer sur les *mêmes broches* données et adresses. Les broches AD0-15 font office tantôt de 16 bits de données si DEN# est bas, tantôt de bus d'adresses si DEN# est haut. Les 4 bits additionnels d'adresse se trouvent sur A16-A19.
3. le bus d'adresses est de largeur 20 bits : il peut adresser 2^20=1 MB.
4. Les données sont transférées sur 16 bits ou deux octets
5. Seuls 8 bits d'adresse sont exploités, puisque A8-A12 des RAM sont à la masse.
6. 256 adresses de 2 octets chacune soit 512 octets, accessibles sur la plage 0x00-0xFF

3 Flux d'exécution

1. `gcc -o executable source.c`
2. aux adresses 804841e et 8048421 : `cmp $0x1,%eax` compare un registre avec 1 et saute, `jne 8048435 <main+0x3a>`, s'il n'y a pas égalité.
3. il suffit de modifier l'argument de la comparaison `cmp` de 1 à 0 : avec l'éditeur de binaire `hexedit`, rechercher la chaîne `7512` qui indique l'instruction `jne`, et modifier la valeur 01 précédent cette séquence par 00. La conséquence de cette modification est telle que attendue :

```
$ ./test
 debloque'
```

4. la seule fonction appelée avant le test est `main` : nous plaçons donc le point d'arrêt (`b` comme *breakpoint* dans `gdb`) dès le début du programme. La séquence complète de commandes `gdb` (qui n'était pas demandée) devient donc

```
gcc -g -o test test.c bloque_debloque.o
gdb test
b main
run
list
s
1       void initialise_i(int* i) {*i=1;}
s
7        if (i==1) printf("bloque'\n");
s
8           else printf("debloque'\n");
set variable i=0
continue
```

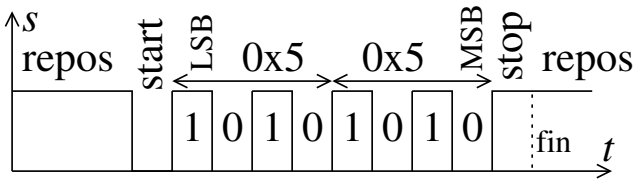
5. la dernière méthode, de surcharger une fonction fournie par une bibliothèque dynamique, n'est pas abordée dans les questions de l'examen mais fonctionne de la façon suivante :

```
$ gcc -g -o test test.c bloque_debloque.o                  # la fonction fournie dans bloque_debloque est statique
$ ldd test
        linux-gate.so.1 (0xb7707000)
        libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb752e000)
        /lib/ld-linux.so.2 (0xb7708000)
$ gcc -shared -fPIC bloque_debloque.c -o bloque_debloque.so # la fonction fournie dans bloque_debloque est dynamique
$ gcc -g -o test test.c bloque_debloque.so                 # et pourra donc etre modifiee a l'execution
$ ldd test
        linux-gate.so.1 (0xb7798000)
        bloque_debloque.so (0xb7793000)
        libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb75bc000)
        /lib/ld-linux.so.2 (0xb7799000)
$ LD_PRELOAD=./pas_bloque_debloque.so ./test               # pas_bloque_debloque.c contient void initialise_i(int* i) {*i=0;}
 debloque'
$ ./test                                                   # tandis qu'initialement on avait void initialise_i(int* i) {*i=0;}
 bloque'
```

Aussi antiques qui puissent paraître ces attaques, elles restent citées dans des document récents tels que `https://www.youtube.com/watch?v=hABj_mrP-no`.

4 Questions de cours

1. 0xdd=221
2. 254=0xfe
3. $5,12/2^8 = 0,02$ V=20 mV
4. `ls a`
5. `unsigned char`
6. `unsigned` *type* `tab[12];`, *type* ∈ {`char, short, int, long, long long`}
7. `tab[11];`
8. `main ()`
9. `Hello` : l'affichage commence au 4ème élément, c'est à dire le caractère de code ASCII 72 (ou 'H'), et affiche la chaîne de caractères jusqu'à atteindre la valeur 0 indiquant sa fin (format `%s` de `printf()`).
10. la transmission se fait par le bit de poids le plus faible (LSB) en premier, donc la transmission commence par le passage du niveau haut à bas pour le START bit, puis alterne entre haut et bas pour les 8 bits qui suivent avant de repasser au niveau haut pour le STOP bit. Cette transmission est spéciale car elle alterne à la fréquence de transmission de chaque bit l'état du bus de communication et permet donc de simplement identifier le débit de communication s'il est inconnu.



11. Une opération aussi simple que la multiplication par 15 donne lieu à de subtiles optimisations. Afin de ne *pas* utiliser la multiplication, toujours plus lente que les opérations faites au cœur de l'ALU, le programme `int main() {volatile n=4; printf("%d\n",n*15);}` se compile (`gcc` 4.9.1) sur architecture x86 en

```
8048416: 89 d0      mov    %edx,%eax
8048418: c1 e0 04   shl    $0x4,%eax
804841b: 29 d0      sub    %edx,%eax
```

qui est une rotation de 4 bits vers la gauche (multiplication par 16) et la soustraction de l'élément initial : $N \times 16 - N = N \times 15$, et ce même en faisant appel à `gcc` sans optimisation (`-O0`).

Plus subtil, sur une petite architecture telle que le MSP430, la multiplication par 15 devient (la valeur initiale est sur l'élément 4 de la pile) :

```
a:  1d 44 fc ff mov  -4(r4),r13
e:  0e 4d       mov  r13,   r14
10: 0f 4e       mov  r14,   r15
12: 0f 5f       rla  r15
14: 0e 4f       mov  r15,   r14
16: 0e 5d       add  r13,   r14
18: 0f 4e       mov  r14,   r15
1a: 0f 5f       rla  r15
1c: 0f 5f       rla  r15
1e: 0f 5e       add  r14,   r15
```

Vous pouvez tester, ça marche ! Cette solution un peu tordue est certainement liée au jeu d'instructions du MSP430 (rotation d'un coup et absence de soustraction).

Références

[1] M. Rafiquzzaman, *Fundamentals of Digital Logic and Microcomputer Design – 5th Ed.*, Wiley Interscience (2005), disponible à `gen.lib.rus.ec`