KLAFYVEL

Zeste de savoir

Multiplications avec Arduino : jetons-nous à la float

26 juin 2022

Table des matières

Introduction

Dans mon précédent billet , nous avons discuté de la représentation flottante des nombres. Dans cet article, on va voir comment on peut utiliser cette représentation pour effectuer des opérations. En particulier, on va s'intéresser à l'implémentation d'une multiplication de flottants sur le processeur de l'Arduino Uno : l'ATMega328P, un microcontrôleur 8 bits.

i

Ce billet a été initialement rédigé pour la newsleter FedeRez de Juin 2022 (non disponible publiquement à ma connaissance).

1. Comment utiliser les flottants ?

La première question à se poser avant d'utiliser des flottants sur un microcontrôleur est «est-ce que j'en ai vraiment besoin?». Cela peut sembler étrange, surtout si on a l'habitude de coder sur des plateformes plus classiques, telles que votre ordinateur portable. Il y a deux raisons pour laquelle se poser la question vaut la peine :

— Un flottant prend de la place: 4 octets, soit 32 bits par nombre . Cela peut sembler anecdotique, mais quand on dispose en tout et pour tout de 2048 octets et 32 registres de 8 bits pour faire tourner son programme, on a tôt fait d'avoir rempli sa RAM avec un tableau.
— Notre microcontrôleur ne dispose pas de *Floating Point Unit* (FPU), au contraire de votre ordinateur par exemple, qui dispose de circuits dédiés au calcul en virgule flottante (qui font que tout ce qu'on va faire dans cet article et le suivant y serait au mieux inutile, au pire contre-productif).

Intéressons-nous aux conséquences concrètes du point numéro deux. Prenons par exemple ce petit programme Arduino très simple

```
1  float x;
2
3  void setup() {
4    x = 1.0;
5    x *= 2.45;
6  }
7
8  void loop() {
9  }
```

Après l'avoir compilé, on peut s'amuser à utiliser l'outil `avr-objdump` pour le décompiler et obtenir le fichier disponible ici [1]. Pour lire ce programme, vous pouvez vous référer à l'AVR Instruction Set manual. On peut commencer par ignorer le début du programme, qui sert principalement à initialiser le microcontrôleur, pour se concentrer sur l'action. En effet, la multiplication de x par $2,45$ s'effectue à la ligne 179.

Comme vous pouvez le constater, il s'agit bien d'un appel de fonction assez classique que notre compilateur a ajouté au programme pour permettre la multiplication de flottants. En continuant votre lecture du programme, vous pourrez vérifier que les lignes 189 à 391 sont consacrées à la gestion des flottants. On y retrouve la gestion de plein de petites particularités des nombre flottants comme les arrondis, *NaN* et autres infinis.

En pratique, cela se traduit par un petit surcoût de temps de calcul quand on multiplie des flottants plutôt que des entiers. Ainsi, quand on mesure le temps nécessaire pour chaque opération, on trouve qu'il faut 92 coups d'horloge (5,75 µs[2]) pour multiplier deux entiers `long` (32 bits sur cette plateforme), alors qu'il en faut 115 (7,1875 µs) pour multiplier deux flottants. Par curiosité, vous pouvez également mesurer le temps nécessaire pour ajouter deux nombres. Vous trouverez qu'ajouter deux `long` prend 22 coups d'horloge (1,375 µs), alors qu'il en faut 131 (8,1875 µs) pour ajouter deux `float`! Cette différence vient probablement du temps nécessaire pour renormaliser le résultat, c'est à dire s'assurer que la mantisse est comprise entre 1 inclus et 2 exclus.

Dans ces conditions, pourquoi donc utiliser le type `float`? On pourrait très bien se contenter d'utiliser des `long` et, par exemple, représenter un nombre x entre -1 et 1 par $x \times 2^{32}$. Il s'agit en réalité d'une très bonne idée, qui s'appelle le calcul en virgule fixe, et que nous creuserons plus en détail dans le prochain article. Retenez que dans beaucoup de situations, quand on travaille sur un processeur tel que le nôtre, la bonne solution est de **ne pas utiliser le type `float`**.

Cependant, il est également des situations où l'utilisation de flottants peut s'avérer utile. Par exemple, si votre processeur est amené à analyser de la musique à l'aide de transformées de Fourier. Partant d'un signal bien quantifié par votre échantillonneur, utiliser des flottants permettra de garder une précision acceptable sur l'ensemble du spectre, bien que certaines fréquences auront des amplitudes plus grandes que d'autres par plusieurs ordres de grandeur.

1. Afin de ménager notre président dans son très bon travail de mise en page de la newsletter, je vous demande chers lecteurices de bien vouloir m'indulger cet usage impromptu mais néanmoins anecdotique des services de nos amis de chez Microsoft. Ceci ne m'empêche pas d'utiliser des notes de bas de page qui ne manqueront pas de lui faire perdre quelques cheveux à la compilation. *Même pas peur! (ndlr.)*
2. J'utilise un ATMega328P monté sur une carte Arduino, l'horloge pédale donc à 16 MHz

2. Cuisinons notre multiplication.

Comme on l'a vu, il y a des situations dans lesquelles on a besoin de `float`. Mais on peut quand même décider de ne pas se contenter des performances de la multiplication de base. Avant de commencer, il nous faut une recette maison pour notre multiplication.

Pour rappel, on a des flottants qui sont représentés en mémoire comme ceci:

$$s \quad e_7e_6e_5e_4e_3e_2e_1e_0 \quad m_1m_2...m_{23}$$

avec $e_7 \cdots e_0$ la représentation de l'exposant stockée avec un biais (on ajoute une constante à l'exposant pour pour ne stocker que des nombres positifs) pour permettre les exposants négatifs. Dans la suite on notera E l'exposant et e sa représentation pour le stockage, tels que $e = E + 127$. Les $m_1 \cdots m_{23}$ sont les 23 bits après la virgule de la mantisse. En utilisant toujours la même notation pour les nombres binaires, on lie donc un nombre x à sa représentation flottante ainsi:

$$x = (-1)^s \times (1, m_1 \cdots m_{23})_2 \times 2^{e-127}$$

On va implémenter tout ça directement en assembleur, donc il peut être utile d'avoir la liste des instructions à portée de main.

Et alors, comment faire notre multiplication? On dispose de deux nombres a et b en représentation flottante, et on souhaite calculer la représentation flottante de $c = a \times b$. Si on développe, on obtient:

$$beginalignedc = (-1)^{color0072b2s^a} times (1, color009e73m_1^a \cdots m_{23}^a)_2 \times 2^{e^b-127} s^b \times (1, m_1^b \cdots m_{23}^b)_2 \times 2^{e^b-127} = (-$$

On a notre algorithme de multiplication:

- On s'occupe du signe du résultat: un *OU exclusif* des signes de a et b suffit;
- On additionne les exposants. Attention, puisqu'on travaille avec leurs représentations biaisées, il est nécessaire de soustraire une fois le biais pour obtenir l'exposant correct. Autrement dit, on a besoin d'avoir $e^c = e^a + e^b - 127$ pour avoir $E^c = E^a + E^b$.
- On multiplie les mantisses. Pour ce faire, on va utiliser l'instruction `fmul`, qui permet de multiplier deux nombres en virgule fixe avec une représentation «1.7»\footnote{Un bit virgule sept bits}, c'est à dire pour lesquels le bit de poids fort d'un octet est associé à 2^0, le suivant à 2^{-1}, jusqu'au bit de poids faible associé à 2^{-7}.
- Il reste à s'assurer de la normalisation du résultat. Étant donné que a et b sont déjà normalisés, les mantisses tronquées (auxquelles on a enlevé les 16 derniers bits) que nous utiliserons sont comprises entre 1 et $2 - 2^{-7}$. Cela signifie que le résultat de la multiplication des mantisses sera compris entre 1 et $(2-2^{-7})^2 < 4$. Puisque notre résultat est sur l'intervalle $[1; 4[$, il est représentable en utilisant deux bits avant la virgule. De plus, puiqu'on a une résolution de 2^{-7}, une résolution de 14 bits après la virgule est nécessaire pour le résultat. En pratique, l'instruction `fmul` renvoie un résultat sur deux

> octets sous la forme «2.15»\footnote{Deux bits virgule quinze bits, donc}, le bit «en trop» (puisque le résultat est sur 17 bits) étant stocké dans le flag *carry* du registre de status. Pour pouvoir assurer la normalisation du résultat, il y aura donc au plus un décalage bit-à-bit sur le résultat de la multiplication des mantisses (quand le bit de *carry* est passé à un). Le cas échéant, il faudra incrémenter l'exposant.

Pour gagner un peu de temps sur la multiplication des mantisses, on peut se contenter de faire la multiplication uniquement sur les 7 premiers bits des mantisses, bien que l'on puisse s'en passer et travailler avec autant d'octets qu'on le souhaite. La documentation donne un exemple de comment implémenter une multiplication sur deux octets. Mais on y reviendra dans le prochain article.

Pour donner une idée de ce à quoi le résultat peut ressembler, voici un exemple d'implémentation . Il y a un peu de subtilité pour effectuer la soustraction du biais de l'exposant pour éviter un *overflow*. Et pour éviter d'utiliser trop de registres, on a besoin d'un peu de gymnastique sur les positionnements des différents morceaux de la représentation flottante. Pour pouvoir utiliser `fmul`, on a aussi besoin de positionner un 1 sur le premier octet de la mantisse. Cela donne un résultat assez rapide, puisque l'on effectue ainsi une multiplication en 102 coups d'horloge (6.375 µs), contre 148 (9.25 µs) pour un cas non pathologique sur les flottants natifs. Ceci bien-entendu au prix de la justesse du résultat: non seulement on ne gère pas les règles d'arrondis ou les infinis, mais les mantisses tronquées induisent des erreurs visibles par un être humain.

On peut s'amuser à optimiser encore plus notre code, en utilisant un patron maison pour nos flottants. En effet, on voit que l'on passe pas mal de temps à chaque multiplication pour avoir la représentation de l'exposant alignée sur un seul registre. On peut donc penser à travailler en permanence avec des exposants alignés sur un seul registre, autrement dit créer notre propre patron, qui pourrait ressembler à ceci:

$$e_7e_6e_5e_4e_3e_2e_1e_0 \quad s \quad m_1m_2...m_{23}$$

À condition que l'on ne passe pas son temps à faire des va-et-vient entre la représentation classique du type `float` et cette représentation, on gagnera ainsi un peu de temps. J'ai implémenté une multiplication avec ce patron pour l'exemple . En mesurant, on obtient une multiplication en 90 coups d'horloge (5,625 µs).

Conclusion

L'idée de cet article était de montrer le fonctionnement interne des flottants au travers d'une opération de base : la multiplication. J'espère qu'avec notre implémentation bricolée, vous avez une meilleure idée des étapes qu'impliquent une telle opération. Il ne s'agit là que d'une caricature d'implémentation, car il faudrait encore prendre soin des règles d'arrondis, et des valeurs spéciales définies par la norme IEEE754.

Mais si vous êtes prêts à mettre les mains dans le cambouis, et à quelques compromis sur la norme, il est possible de venir gratter quelques coups d'horloge sur chacune de vos multiplications. L'implémentation proposée possède d'autres propriétés rigolotes, par exemple elle a un temps d'exécution quasi-constant, contrairement à l'implémentation par défaut de la multiplication.

Ainsi, la multiplication de manière native de $3,40$ et $55,5$ (115 coups d'horloge, soit 7,1875 µs) est plus lente que celle de $0,2$ et $55,5$ (90 coups d'horloge, soit 5,625 µs, en moyenne). Enfin, on pourrait imaginer d'autres codes *ad hoc* de multiplication, par exemple pour les multiplications en place (c'est-à-dire en modifiant directement une des variables d'entrée).

Dans le troisième et dernier article de la série, on parlera d'un autre type de représentation des nombres à virgule que nous avons déjà effleuré : la représentation en virgule fixe !