Programmation en assembleur 1 : boucles et «drapeaux»

On s'intéresse à un processeur dont voici les principales caractéristiques [1] :

Architecture 16 bits

- tous les registres font 16 bits, tous les mots mémoire font 16 bits ;
- toutes les opérations opèrent sur des données de 16 bits ;
- toutes les adresses font 16 bits ;
- toutes les instructions sont codées sur 16 bits.

Attention, la mémoire est adressable par octet : les adresses sont des adresses d'octets, pas de mots 16 bits.

16 registres internes

- R0 est le *pointeur de programme*, et on peut l'appeler PC *(program counter)*
- R1 est un *pointeur de pile*, et on peut l'appeler SP *(stack pointer)*
- R2 est le *registre de statut*, et on peut l'appeler SR *(status register)*. Il contient en particulier les drapeaux Z *(zero)*, C *(carry)*, V *(oVerflow)*,et N *(negative)*, qui sont mis à jour par chaque opération de calcul.
- R3 est spécial mais on n'y comprend rien alors on n'y touche pas
- R4 à R15 sont des registres d'usage général.

Opérations de calcul à deux opérandes qui s'écrivent `OP operand TO destination`

par exemple `ADD R4 TO R5`, qui réalise l'addition de R4 à R5 et range la somme dans R5.

Les opérations OP disponibles sont : `MOV`, `ADD`, `SUB` (qui fait `dest` $-$ `op` $\rightarrow$ `dest`), `ADDC` *(add with carry)*, `SUBC` *(subtract with carry)*, `AND`, `OR`, `XOR`. Elles mettent à jour les drapeaux lorsque cela a du sens. Il y a aussi une instruction `CMP` qui est identique à `SUB` mais qui ne modifie pas la destination.

Variante : `OP.B operand TO destination` par exemple `ADD.B R4 TO R5` qui n'opère que sur les 8 bits de poids faible (et annule les 8 bits de poids fort de R5).

Modes d'adressage Les champs *opérande* et *destination* peuvent être de l'une des formes suivantes :

- `Ri` : la donnée est dans le registre `Ri`
- `[Ri]` : la donnée est dans la case mémoire dont l'adresse est la valeur contenue dans `Ri`.
- `[Ri]+` : idem, mais `Ri` est post-incrémenté (de 1 pour une instruction `.B`, de 2 sinon).
- `[const]` où `const` est une constante 16 bits : la donnée est dans la case mémoire d'adresse `const`.
- `[const+Ri]` : la donnée est dans la case mémoire d'adresse `const+Ri`.
  Deux cas particuliers importants en pratique : lorsque `Ri=R0=PC`, et lorsque `Ri=R1=SP`. Lorsque `Ri=PC`, on parle de l'adresse du premier mot de l'instruction.
- `#const` : une valeur constante (ce mode d'adressage n'a pas de sens pour la destination).

Pour les 3 dernières formes, la (ou les) constante(s) 16 bits est (sont) stockée(s) dans le programme juste derrière l'instruction : ces instructions occuperont donc deux ou trois mot mémoire. *Dans le langage assembleur, on utilisera le préfixe 0x pour distinguer les constantes hexadécimales des constantes décimales. ($15 \neq 0x15 = 15_{16} = 21$)*

Sauts relatifs conditionnels Un saut relatif s'écrit `J const IF condition`

Cette instruction ajoute la constante `const` au PC, sous une certaine condition sur les drapeaux. La constante est ici un entier signé codé sur 10 bits, soit une portée de +/- 512 cases mémoires. Ce déplacement est relatif à l'adresse de l'instruction `J`.
Les conditions possibles sont :

- «toujours vrai» : le saut est toujours pris
- `Z` ou `EQ` : le saut est pris ssi le drapeau Z vaut 1, i.e. ssi la dernière opération/comparaison a donné zéro,
- `NZ` ou `NE` : le saut est pris ssi Z=0, i.e. ssi la dernière opération/comparaison a donné un résultat non-nul,
- `C` : le saut est pris ssi le drapeau *carry* vaut 1 ; `NC` ss'il vaut 0,
- `N` : le saut est pris ssi le drapeau *negative* vaut 1,
- et quelques autres.

Remarque : on peut faire des sauts absolus avec `MOV operand TO PC`.

Temps d'exécution Chaque instruction dure un nombre de cycles exactement égal au nombre d'accès mémoire qu'elle nécessite. Attention, cela inclut le chargement de l'instruction elle-même. Un saut relatif prend deux cycles.

1. Ce jeu d'instructions est directement inspiré de celui du MSP430, que vous allez utiliser dans les TP du second semestre

Introduction

Dans ce TD vous allez programmer quelques algorithmes simples : multiplication, boucles. L'objectif est de pratiquer les notions de base de la programmation assembleur : représentation en binaire des instructions et des données (nombres, texte), jeu d'instructions (quelles sont les différentes opérations disponibles sur mon architecture pour calculer sur ces données), les registres du processeur (généralistes ou spécialisés).
En particulier, vous allez devoir vous intéresser au registre SR, dit «registre de statut», dans lequel le processeur maintient, à destination de votre programme, des informations sur son état actuel. C'est dans ce registre que les instructions de saut conditionnel consultent les «drapeaux» pour savoir si elles doivent sauter ou pas (voir l'illustration en page 2).

1 Mise en bouche : multiplication par trois

Question 1-1 Ecrivez le code assembleur qui réalise la multiplication par 3 de la valeur stockée à l'adresse 16, et range le résultat à l'adresse 18. Combien de mots mémoire ce code nécessite-t-il ? Combien de cycles dure son exécution ?

Question 1-2 On va reprendre le code précédent, mais on va en faire une *routine*[2] dont le *point d'entrée* est la première instruction, et dont les variables sont stockées à côté du code, juste avant le point d'entrée. En mémoire cela donnerait quelque chose comme ceci, pour une routine qui calcule Y=3*X :

0042	...
0044	variable X
0046	variable Y
0048	Début du code de mul3
004A	code de mul3
004C	code de mul3
004E	...

Ici le *point d'entrée* serait l'adresse 48.
Remarquez que les adresses des cases mémoires sont paires.

On souhaite que ce code soit *relogeable*, c'est-à-dire fonctionne quelque soit l'endroit en mémoire où il sera placé. Reprenez votre code de la question précédente, et remplacez ce qu'il faut pour qu'il réponde à ce cahier des charges. Indice : il faut utiliser un mode d'adressage relatif au PC pour attraper X et Y.

2 Multiplication d'entiers quelconques

Question 2-1 Écrivez une boucle qui multiplie par 256 un entier donné dans R15. (Vous pouvez écrire une version déroulée dans un premier temps. Si vous faites ainsi, comparez les tailles de code et les durées d'exécution de vos deux versions)

Étiquettes nommées Désormais on se permettra d'utiliser dans l'assembleur des *étiquettes*. Ce sont des noms qui désignent une adresse mémoire et qu'on peut mettre à gauche d'une instruction pour désigner son adresse, par exemple :

```
                    CMP R5 TO R14
                    J label2 IF N
label1 :            instr
                    ...
                    instr
                    J label3
label2 :            instr
                    ...
label3:             instr
```

L'assembleur, i.e. le programme qui traduit votre code assembleur en une séquence de mots de 16 bits, se chargera de remplacer `label3` dans `J label3` par la différence d'adresses correspondante.

Question 2-2 Le code ci-dessus est une construction classique de programmation, laquelle ? Remplacez `label1`, `label2` et `label3` par des noms plus intelligents, et que cela vous serve de leçon.

Question 2-3 Modifiez votre boucle pour qu'elle réalise la multiplication de deux entiers positifs 8 bits passés dans R14 et R15. Dans cette question, on se contentera d'ajouter R15 à lui même R14 fois.

Question 2-4 Posez la multiplication en binaire de 17 par 42 (par exemple). Vous avez réalisé seulement 8 additions, mais autant de décalages. Comment fait-on un décalage avec nos instructions ? Proposez du code assembleur réalisant la multiplication de deux entiers positifs 8 bits passés dans R14 et R15. Commentez chaque instruction !

Question 2-5 On a un peu galéré pour tester successivement les bits de notre multiplicande. Suggérez une instruction assembleur qui nous simplifierait la vie. Indice : dans le MSP430 elle s'appelle RRC pour *rotate right through carry*.

2. terme parfois utilisé pour un morceau réutilisable de code assembleur

3 Conversion de chaîne de caractère en majuscules

On rappelle que pour le code ASCII, les minuscules a à z ont les codes 0x61 à 0x7a, alors que les majuscules A à Z ont les codes 0x41 à 0x5a. On rappelle aussi qu'en C une chaine de caractère se termine par l'octet 0x00.

Question 3-1 Implémentez une routine qui reçoit dans R15 l'adresse d'une chaîne de caractères et convertit cette chaîne en majuscules, en ignorant bien sûr tout ce qui n'est pas une minuscule.