Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

tp.02 - Introduction à l'assembleur

Classes T-2/I-2 // 2016-2017

Daniel Gachet | HEIA-FR/TIC

tp.02 | 04.10.2016

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Objectifs

- A la fin du laboratoire, les étudiant-e-s seront capables de
  - Utiliser différents modes d'adressage du µP ARM
  - Développer (concevoir, coder et tester) un algorithme élémentaire en assembleur permettant de générer et d'afficher les nombres premiers jusqu'à 100
  - Maîtriser le processus d'assemblage, d'édition de lien et de débogage
- Durée
  - 1 séance de laboratoire (4 heures)
- Rapport
  - Journal de laboratoire avec le code source

Travail à réaliser

- Développez un code assembleur capable de générer les nombres premiers jusqu'à 100 selon l'algorithme du crible d'Eratosthène

```
#define MAX 100
bool is_a_prime_number[MAX];
void prime_number_generator() {
        // 1st mark all numbers as prime number
        for (int i=0 ; i<MAX ; i++)
                is_a_prime_number[i] = true ;

        // 2nd mark all multiples as not a prime number
        for (int i=2 ; i<MAX ; i++)
                for (int j=i*2 ; j<MAX ; j+=i)
                        is_a_prime_number[j] = false ;

        // 3rd print all prime numbers
        for (int i=2 ; i<MAX ; i++)
                if (is_a_prime_number[i])
                        printf ("%d\n", i);
}
```

- Les conditions d'exécution sont les suivantes
  - Le squelette du projet se trouve sur le dépôt centralisé
  - Pour le télécharger, tapez les commandes suivantes

    ```
    $ cd ~/workspace/se12/tp
    $ git pull upstream master
    ```
  - L'algorithme sera intégré dans le fichier « main.S »
- Le code et le journal seront rendus au travers du dépôt Git centralisé
  - *sources : .../tp/tp.02*
  - *rapport : .../tp/tp.02/doc/report.pdf*
- Délai
  - Le journal et le code doivent être rendus le soir même du TP au plus tard à 24h00

Les questions...

- Citer les modes d'adressage courants supportés par le µP ARM ?
- Quel outil permet de transformer le code assembleur en code objet ?
- Quel outil permet de créer l'application à partir des codes objet précédemment générés ?
- Pourrait-on optimiser l'algorithme du crible d'Eratosthène ? Si oui, comment ?
- Pourrait-on réduire la taille de votre code en assembleur ? Si oui, comment ?

Indications de codage

- L'adresse d'une variable peut facilement être chargée dans un registre avec l'instruction assembleur suivante

  ```
  LDR Rn, =variable
  ```

- Une constante peut être chargée dans un registre avec l'instruction assembleur suivante

  ```
  LDR Rn, =<constante>
  ```

- Les instructions ci-dessous permettent de transférer une donnée entre un registre et la mémoire

  ```
  LDR<sz> Rx, [Ry]
  STR<sz> Rx, [Ry]
  ```

  `<sz>` spécifie la taille de la donnée
  - `B` pour un mot de 8 bits
  - `H` pour un mot de 16 bits
  - `blanc` pour un mot de 32 bits

Indications de codage (II)

- Le test de variables peut s'effectuer à l'aide de la combinaison des instructions suivantes

```
CMP Rn, Rm
B<cc> <etiquette>
```

- Si la valeur à tester est inférieure à 256, on peut utiliser l'instruction de test suivante

```
CMP Rn, #<val>
```

Condition<cc>	Description
EQ	equal
NE	not equal
HI	higher
HS	higher or same
LS	lower or same
LO	lower

Indications de codage (III)

- Le codage des valeurs hexadécimales se fait à l'aide d'un `0x<val>`
- L'affichage d'un string sur le terminal «minicom» peut être effectué avec le jeu d'instructions suivant

```
LDR R0, =<format> // printf formatting string
LDR R1, =<1st-arg> // printf 1st argument / optional
LDR R2, =<2nd-arg> // printf 2nd argument / optional
LDR R3, =<3rd-arg> // printf 3rd argument / optional
BL printf
```

- Attention : après l'appel de la fonction `printf`, les registres R0 à R3 seront très certainement altérés