TP 1 : Introduction à l'environnement de développement

Introduction

Pour cette première séance, vous allez vous familiariser avec l'environnement dans lequel vous travaillerez pendant 8 séances. Cet environnement est constitué :

– D'une carte-cible (HCS12TB), sur laquelle est implantée un microcontrôleur Freescale 9s12DP512, associé à divers connecteurs et périphériques.
– D'un PC "hôte", à partir duquel vous allez développer des programmes qui seront ensuite exécutés sur la carte cible, via l'utilisation de plusieurs logiciels.

La carte cible est connectée par une liaison-série asynchrone sur le port COM du PC. Vous verrez comment écrire un programme en assembleur, comment le transformer en un programme exécutable par la cible, comment le télécharger dans la cible, et comment l'exécuter.

En complément de ce sujet, pensez à consulter le cours et le poly de référence distribué.

1 Concepts fondamentaux

1.1 Présentation microcontrôleur 9s12DP512

Ce microcontrôleur 16 bits, de noyau identique au 68hc12, présente les caractéristiques suivantes :

– Boîtier LQFP112 (CMS), *operating voltage* : 5V
– Fréquence d'horloge jusqu'à 25 MHz
– 112 broches, jusqu'à 89 broches d'E/S
– Espace mémoire adressable de 65,5ko (bus d'adresse de 16 bits)
– mémoire : 4 ko EEPROM, 14 ko RAM, 512 ko mémoire Flash (paginée)
– interfaces séries synchrone et asynchrone
– Timer 16 bits
– Convertisseur A/D 10 bits 16 canaux

1.2 Programmation "traditionnelle" & programmation "embarquée" ?

– En programmation traditionnelle, on travaille sur un ordinateur, qui, est utilisable grâce à un système d'exploitation (*Operating System*, O.S. en anglais). Celui-ci s'occupe d'allouer les ressources de l'ordinateur, matérielles (périphériques) et logicielles (mémoire vive, services divers) à votre programme, de façon transparente pour vous :
  – Lorsque vous faîtes une saisie clavier, vous n'avez pas à vous soucier de la façon dont le clavier est géré électroniquement : vous utilisez directement la fonction fournie par le système d'exploitation.
  – Lorsque vous chargez/exécutez un programme, vous ne vous souciez pas de l'endroit auquel il est chargé en mémoire.
– En programmation embarquée (*embedded* en anglais), il y des limitations matérielles : pas de disque dur, taille et consommation limitées : on travaille parfois sans O.S. Il faut alors directement gérer les périphériques et la mémoire dans son programme. De plus, les périphériques n'étant pas ceux d'un ordinateur traditionnel, les fonctions classiques d'E/S n'ont plus de raisons d'être : on peut tout de suite oublier la fonction `printf()` quand l'affichage se résume à des DELs. . .

Note : les performances des systèmes embarqués contemporains permettent dans certains cas l'utilisation d'un système d'exploitation dédié (par ex. Android, basé sur GNU/Linux).

1.3 Le deboggeur NoIce

L'objectif d'un un logiciel débogueur est de permettre le téléchargement du programme utilisateur dans la mémoire RAM ou FLASH du composant, et de permettre la vérification de son bon fonctionnement : exécution pas à pas, examen et édition de la mémoire, etc, via une interface-utilisateur ergonomique. Il doit arriver à contrôler le processeur cible. Ceci est possible via un petit programme implanté en mémoire Flash dans le processeur cible (appelé "moniteur"), et qui intercepte les commandes transmises par le logiciel débogueur : il les exécute en renvoyant éventuellement les données demandées, via une liaison série (câble de connexion entre le PC et la carte-cible).

2 Travail proposé

2.1 Utilisation de NoIce

1. Lancer le programme "NoIce". Vous obtenez l'écran de la fig. 1, découpé en plusieurs parties :
   – La barre d'outil, qui est dotée d'"info-bulles" (faîtes-y attention !).
   – La fenêtre "Registres", qui montre leur valeur courante.
   – La fenêtre "code", qui montre sur 3 colonnes l'**adresse**, le **code opératoire** qui s'y trouve, et le **mnémonique** correspondant, avec les opérandes.
   – La fenêtre du bas, qui peut afficher plusieurs choses, selon l'onglet choisi (*Data*, *Output*, *View*, *Watch*, *Memory*).

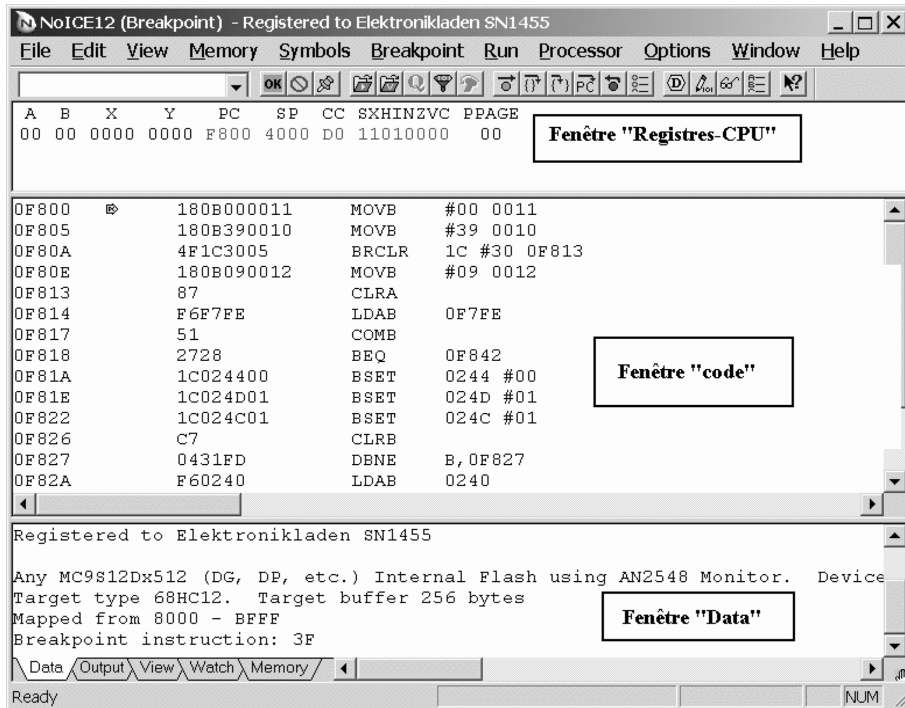

FIGURE 1 – Copie d'écran du logiciel **NoIce**.

En cas de problème de communication avec la carte : appuyer **d'abord** sur le bouton de reset de la carte, puis sur le bouton *Reset the target* dans la barre d'outil de NoIce.

L'utilisation peut se faire de 3 façons, à votre convenance :
- En cliquant sur le bouton de la barre d'outil (s'il existe),
- En utilisant le menu correspondant,
- En tapant la commande désirée dans la ligne de saisie. (voir la référence des commandes dans le poly.)

Une aide en ligne est disponible en appuyant sur la touche F1.

2. A partir du poly. de référence, étudier le plan mémoire : quelle est la première adresse de la RAM : ________________
3. Avec la commande "Dump", afficher le contenu des 512 premiers octets de la RAM.
4. Désassembler le code qui se trouve à l'adresse $f800
   - Quelle est l'instruction (le mnémonique) qui se trouve en $f800 : ________________
   - Combien d'octets occupe cette instruction : ________________
   - Quelle est la valeur de l'octet en $f800 : ________________
5. Dans la fenêtre du bas, sélectionner l'onglet "Memory", et afficher le contenu de la mémoire en $2000
6. Cliquer directement sur un octet : vous pouvez le modifier directement au clavier (Il devient rouge). Vérifiez que la modification a été prise en compte avec la commande "Dump".
7. Cliquer directement sur l'accumulateur A dans la fenêtre "Registres" : vous pouvez directement modifier la valeur d'un registre au clavier.

2.2 Premiers programmes en assembleur

Un programme utilisateur est constitué d'un fichier source, qu'il va falloir "traduire" en une suite de codes opératoires, le seul langage que le processeur "comprend". Cette tâche est réalisée par l'**assembleur**, qui est spécifique au processeur. Ce logiciel prend le fichier source en entrée, et sort 2 fichiers :
- un fichier listing (d'extension .lst)
- un fichier .s19, qui contient dans un format "texte" la séquence de codes opératoires correspondant au programme source. Le format de ce fichier est normalisé, de façon à permettre l'interopérabilité entre plusieurs logiciels et cartes-cibles

L'assembleur est un programme en ligne de commande, mais sa mise en oeuvre en sera facilitée par l'utilisation d'un **IDE** (*Integrated Development Environment*), qui permet de faire facilement l'édition du source et l'assemblage depuis le même écran.

2.3 Programme 1

1. Lancer le programme "MiniIDE" (laisser NoIce en route). Cliquer sur le bouton *New*, et saisir les lignes suivantes (utiliser des tabulations pour passer d'un champ à un autre) :

```
1 debut ldaa #1
2       ldab #2
3       aba
```

   Les 2 premières instructions chargent les accumulateurs A et B avec les valeurs '1' et '2'. La troisième ligne additionne les 2 accumulateurs, et place le résultat dans A

2. Il faut ensuite indiquer "l'arrêt" du programme. Pour l'instant, nous ferons ceci via une boucle infinie, en utilisant une instruction de branchement : `bra` (*BRanch Always*). Rajouter à la fin la ligne :
   `fin bra fin`

3. Il faut spécifier l'endroit de la mémoire ou ce programme doit être chargé. Ceci se fait avec la directive `org` (pour "origine"). Ceci n'est pas une instruction exécutable par le processeur, mais une instruction donnée à l'assembleur d'implanter ce programme à une adresse donnée. Saisir la ligne suivante, en tête du programme :
   ```
   org   $1000
   ```
4. Il reste à commenter le programme. Saisir la ligne suivante à la première ligne du source :
   ```
   ; programme qui additionne A et B
   ```
5. Sauvegarder dans le dossier `z:\TP_II2`, avec le nom `tp1_1.asm` (taper l'extension !)
6. Pour assembler le programme, cliquer sur le bouton *Build Current*. Vous devez avoir dans la fenêtre du bas :
7. En ouvrant un explorateur Windows, se placer sur le dossier `z:\TP_II2`, et ouvrir le fichier `tp1_1.lst` :
   - Quel est le code opératoire de l'instruction `ldaa` : ____________________
   - Combien d'octets occupera votre programme en mémoire : ____________
8. Ouvrir le fichier `tp1_1.s19` : retrouvez-vous les codes opératoires relevés dans le fichier "listing" ? Quelles sont les autres informations qui y figurent ?
9. Sous NoICE, cliquer sur le bouton *Load File to Memory*, et aller chercher le fichier `tp1_1.S19` dans le dossier `z:\TP_II2`
10. Dans la fenêtre "Registres-CPU", cliquer sur PC (compteur ordinal), et dans la boite de dialogue qui apparaît, indiquer l'adresse de son programme : une petite flèche jaune doit apparaître devant la ligne correspondante dans la fenêtre "code".
11. Vérifier que son programme est bien téléchargé en RAM, en le désassemblant : Menu "View→Disassemble at", et spécifier l'adresse où le programme est placé. (Ou bien clic-droit dans la fenêtre "code", et *View Source at PC*). On doit voir apparaître en tête du code les 4 lignes de son programme.
12. Cliquer sur le bouton *Step Instruction*, et vérifier que les registres changent bien de valeur à chaque exécution d'une instruction.

2.4 Programme 2

Nous allons voir comment on peut agir sur les broches d'un port de sortie par programme.

1. Sur la carte, les 8 broches PB0 à PB7 sont connectées à 8 DELs. Chercher ces broches et les DELs sur le schéma de la carte.
   Ces broches sont accessibles par programme via 2 registres 8 bits, appelés PORTB et DDRB (pour *Data Direction Register port B*) :
   - la valeur contenue dans le registre DDRB définira le sens des broches : '0' → broche PBx en entrée, '1' → broche PBx en sortie.
   - la valeur contenue dans le registre PORTB définira les niveaux logiques sur PB0 à PB7 (si ces broches sont en sortie...)
2. Créer un nouveau document, y saisir le source ci-dessous, et l'enregistrer comme `tp1_2.asm` :

```
1         org  $1000
2  debut  cli
3         ldaa #$ff
4         staa DDRB
5  encore ldaa #$0f
6         staa PORTB
7         ldaa #$f0
8         staa PORTB
9         bra  encore
```

3. Pour l'instant, un assemblage débouche sur une erreur : les symboles DDRB et PORTB sont indéfinis. Vous allez pour l'instant les définir manuellement, en ajoutant en tête du source les lignes suivantes :

```
1  DDRB   equ  $0003
2  PORTB  equ  $0001
```

   Remarque : `equ` n'est **pas** une instruction exécutable du processeur, c'est une **directive assembleur**, indiquant simplement de remplacer chaque occurrence d'un symbole par une valeur numérique
   (équivaut au `#define` du langage C).
4. Assembler le programme, (bouton *Build Current*), vérifier qu'il n'y a pas d'erreurs, et le télécharger dans la carte cible avec NoIce.
5. Positionner le compteur ordinal (PC) à l'adresse du programme, et exécuter en pas à pas (Bouton "Step Instruction"). Observez les DELs, et vérifier qu'elles changent bien d'état.
6. Exécuter le programme en mode "normal" (bouton *Go/Halt*), **après** avoir repositionné PC à l'adresse de début du programme ($1000). Que se passe-t-il sur les DELs ? Pourquoi ?

2.5 Programme 3

Afin de ralentir le fonctionnement du programme, nous allons insérer un sous programme de temporisation. Nous allons voir comment :
- utiliser des sous programmes,
- inclure un fichier de définition des ports d'E/S.

1. Créer un nouveau document, y saisir le source ci-dessous, et l'enregistrer comme `tp1_3.asm` :

```
1        org   $1000
2 debut  cli
3        ldaa  #$ff
4        staa  DDRB
5        clr   PORTB  ; RAZ
6 encore inc   PORTB  ; incrémente
7        bsr   TEMPO
8        bra   encore
```

   L'instruction `bsr` (*Branch to SubRoutine*) est un appel à un sous-programme, qui doit être défini.

2. Ajouter en dessous les lignes suivantes

```
1 TEMPO ldy  #0
2 ICI   dey        ;decrémente Y
3       bne  ICI   ; test et branchement
4       rts        ;retour de SP
```

   Ce sous programme ne fait "rien", si ce n'est faire décompter le registre-CPU Y de 0 jusqu'à 0, c'est à dire effectuer 65536 itérations (2^{16}). L'instruction `bne` (*Branch if Not Equal*) teste le bit Z, et se branche à l'étiquette indiquée si le bit est à 0.

3. Ajouter en tête la ligne suivante, permettant d'inclure automatiquement les définitions des registres d'E/S du processeur :

```
1     .nolist
2 #include regs_hc12.inc
3     .list
```

4. Assembler le programme, (bouton *Build Current*), vérifier qu'il n'y a pas d'erreurs. Observez le fichier listing. A quoi servent les directives ".nolist" et ".list" ?
5. Télécharger dans la carte cible le programme, et l'exécuter à vitesse normale. Combien de temps faut-il au processeur pour faire un "tour complet" (approximativement) ?