Introduction à l'assembleur ARM:

variables et accès mémoire

GIF-1001 Ordinateurs: Structure et Applications, Hiver 2017

Jean-François Lalonde

Plan

- Cette semaine:
  - Déclarer des variables et leur affecter des valeurs
  - Effectuer des opérations mathématiques et logiques
- Les semaines prochaines:
  - Gérer la séquence des opérations avec des énoncés conditionnels et des boucles
  - Appeler des fonctions (diviser une tâche en sous-tâches)
  - Gérer les évènements et les exceptions

Addition en assembleur ARM

- Exemple (a = b + c):

Simulateur du TP1

```
MOV R0, #0x40    ; Adresse de la variable b
LDR R0, [R0]     ; Lire la variable b dans le registre R0
MOV R1, #0x41    ; Adresse de la variable c
LDR R1, [R1]     ; Lire la variable c dans le registre R1
ADD R1, R0       ; R1 = R1 + R0
MOV R3, #0x42    ; Adresse de la variable a
STR R1, [R3]     ; Écrire le registre R1 dans la variable a
```

ARM

```
MOV R0, #0x4   ; Adresse de la variable b
LDR R0, [R0]   ; Lire la variable b dans le registre R0
MOV R1, #0x8   ; Adresse de la variable c
LDR R1, [R1]   ; Lire la variable c dans le registre R1
ADD R1, R0, R1 ; R1 = R0 + R1
MOV R3, #0xC   ; Adresse de la variable a
STR R1, [R3]   ; Écrire le registre R1 dans la variable a
```

Démonstration

(addition)

Revenons à notre exemple d'addition

```
MOV R0, #0x4   ; Adresse de la variable b
LDR R0, [R0]   ; Lire la variable b dans le registre R0
MOV R1, #0x8   ; Adresse de la variable c
LDR R1, [R1]   ; Lire la variable c dans le registre R1
ADD R1, R0, R1 ; R1 = R0 + R1
MOV R3, #0xC   ; Adresse de la variable a
STR R1, [R3]   ; Écrire le registre R1 dans la variable a
```

- Pas pratique:
  - d'avoir à connaître les adresses de a, b, et c
  - d'avoir à utiliser deux instructions (MOV puis LDR) pour les charger
- Solution?
  - L'assembleur nous permet de donner un nom à des adresses mémoires: ce sont les constantes et les variables!
    - constante = ne change pas
    - variable = peut changer

« Constantes » — syntaxe

- Déclarer une constante pré-initialisée

```
nom DCss valeur
```

  - “`nom`” est le nom de la constante
  - “`DCss`": C=constante, ss indique la taille. Par exemple:
    - DC8: constante de 8 bits
    - DC32: constante de 32 bits
- “`valeur`”: la valeur de la constante
- exemple:

```
b   DC32 0xAB
c   DC32 0xF2
```

Variables — syntaxe

- Déclarer une variable (réserve de l'espace mais n'affecte pas de valeur):

```
nom DSss nombre
```

  - "`nom`" est le nom de la variable
  - "`DSss`": S=variable, ss indique la taille. Par exemple:
    - DS8: variable de 8 bits
    - DS32: variable de 32 bits
- "`nombre`": le nombre d'éléments à réserver
- exemple:

```
; dans notre exemple d'addition, la valeur initiale importe peu,
; car nous allons la remplacer, mais en général on peut initialiser
; les variables de la même façon que les constantes
a  DS32 1
```

« constantes » vs « variables »

- `nom DCxx valeur`
  - attribue une *valeur* à un espace mémoire
    - n'est pas « constant » à proprement parler… on peut modifier sa valeur par la suite!
  - c'est une *déclaration* (on déclare une valeur).

- `nom DSxx nombre`
  - réserve un *nombre* d'espaces mémoire *sans attribuer de valeur*.
  - c'est une *allocation* (on alloue la mémoire sans déclarer de valeur).

Exemple d'addition—avec variables & constantes

- Comment représenter notre programme d'addition en utilisant des variables et constantes?

```
; Définissons les constantes b et c
b DC32 0xAB
c DC32 0xF2
```

```
; Définissons la variable a (pour stocker le résultat)
a DS32 1
```

```
; Programme principal
LDR R0, b        ; Mettre la valeur de la constante b dans R0
LDR R1, c        ; Mettre la valeur de la constante c dans R1
ADD R1, R0, R1   ; R1 = R0 + R1
LDR R3, =a       ; Mettre l'adresse de la variable a dans R3
STR R1, [R3]     ; Écrire le registre R1 dans la variable a
```

Plus que de la “traduction” d’instructions

- En plus de traduire des mots/mnémoniques en binaire, l’assembleur interprète aussi le texte de plusieurs façons. Il permet:
  - d’associer des mots du programmeur à des adresses de mémoire.
  - au programmeur de déclarer des variables et il gère l’adresse de ces variables à travers les instructions du programme.
  - au programmeur d’identifier des fonctions ou des sections de codes avec des étiquettes (labels). Lorsque l’assembleur décode un appel de fonction ou un branchement (saut) dans le programme, il remplace les étiquettes par les adresses ou déplacements (offset) appropriées.
- L’assembleur supporte des directives qui lui disent comment placer le code en mémoire, comment gérer plusieurs fichiers, comment précompiler le code—modifier le code avant de le traduire en binaire—et plus. Les directives sont des mots réservés qui ne génèrent pas de code en binaire, mais qui dirigent la création du code exécuté.
- L’assembleur permet aussi d’insérer des commentaires dans le code!

Exemple de programme sur le simulateur

```
SECTION INTVEC

B main

; Déclaration des constantes b et c
b DC32 0xAB
c DC32 0xF2
```

Table des vecteurs d'interruption
(plus de détails dans 2 semaines!)

Cette section occupe 128 octets.

```
SECTION CODE

main

; Programme principal
LDR R0, b
LDR R1, c
ADD R1, R0, R1
LDR R3, =a
STR R1, [R3]

B main
```

Code principal

```
SECTION DATA

; Déclaration de la variable a
a DS32 1
```

Variables en mémoire
(non initialisées)

Démonstration

(addition, avec variables)

Tableaux

- Pour déclarer un tableau:

```
nom DCss el1 el2 el3 ... ; Déclaration
nom DSss nombreElements  ; Allocation
```

  - “`nom`” est le nom de la variable/constante
  - “`D*ss`”: ss indique la taille
  - “`el1 el2 el3`”: la valeur des éléments du tableau s’il s’agit d’une constante
  - “`nombreElements`”: le nombre d’éléments dans le tableau s’il s’agit d’une variable

- exemple:

```
a   DS32 3                  ; tableau de 3 mots de 32 bits chacun
b   DC8 0x01 0x02 0x03      ; tableau de 3 octets
```

Démonstration

(tableaux)

Tableaux

- Les tableaux peuvent être vus comme des chaînes de variables.

- Une chaîne texte est un exemple de tableau d'octets, chaque caractère est présenté comme un élément de code ASCII (0 à 255).

- Par exemple:

```
chA DC8 'Hello', 0
chB DC8 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x00
```

- `chA` est la copie exacte de `chB`. Lorsque l'assembleur voit une chaîne entourée par des ' ', il la convertit automatiquement en un ensemble d'octets.

Déplacement de Données: MOV

- L'instruction

```
MOV Rn Op1
```

  met l'opérande de type 1 `Op1` dans le registre `Rn`

- Opérande de type 1:
  - Constante: toujours précédée du symbole #
  - Registre
  - Registre décalé
    - Le décalage est fait avant l'opération. Ces opérations sont détaillées en Annexe A.

- Exemples:

```
MOV R0, #1234          ; R0 = 1234
MOV R0, R1             ; R0 = R1
MOV R0, R1, ASR #2     ; R0 = R1 / 4
```

Accès Mémoire: Load/Store

- Les accès à la mémoire se font avec deux instructions:
  - LDR (LoaD Register) lit la mémoire et met la valeur lue dans un registre.
  - STR (STore Register) met la valeur d'un registre dans la mémoire.
- Ces instructions ont le format

```
LDR Rd, Op2
STR Rs, Op2
```

- `Rd` et `Rs` décrivent le registre de destination ou de source
- `Op2` est une opérande de type 2

Opérande de type 2

- Symbolise tous les modes d'adressage du microprocesseur: toutes les façons permises pour désigner une adresse de la mémoire.

- Se découpe ainsi

```
LDR Rd, [Rb, Offset]
```

  - `Rb` est le registre de base
  - `Offset` est une opérande de type 1

```
LDR R0, [R2]              ; R0 = Mémoire[R2]
LDR R0, [R2, #4]          ; R0 = Mémoire[R2+4]
LDR R0, [R2, R3]          ; R0 = Mémoire[R2+R3]
LDR R0, [R1, R2, LSL #2] ; R0 = Mémoire[R1 + (R2 * 4)]
```

- Pour calculer l'adresse, on additionne Rb et Offset

Opérande de type 2

- Pour faciliter les accès aux tableaux, on peut modifier Rb:
  - avant le calcul d'accès mémoire (pre-indexing)
    - symbole "!"

```
LDR R0, [R1, #4]!      ; R1 = R1 + 4, suivi de R0 = Memoire[R1]
```

  - après le calcul d'accès à la mémoire (post-indexing).
    - en dehors des [].

```
LDR R0, [R1], #4       ; R0 = Memoire[R1], suivi de R1 = R1 + 4
```

Démonstration

(Tableaux avec adressage)

Récapitulation: MOV vs LDR/STR

- MOV: déplacements *entre des registres seulement*

```
MOV R0, #0xFF          ; R0 <- 0xFF
MOV R0, R1             ; R0 <- R1
MOV R0, R1, ASR #2      ; R0 <- (R1 / 4)
```

- LDR/STR: déplacements *entre le CPU et la mémoire*

```
LDR R0, [R1]           ; R0 <- Memoire[R1]
LDR R0, [R1, #4]       ; R0 <- Memoire[R1 + 4]
LDR R0, [R1, R2]       ; R0 <- Memoire[R1 + R2]
LDR R0, [R1], #4       ; R0 <- Memoire[R1], R1 <- R1 + 4
```

```
STR R0, [R1]           ; Memoire[R1] <- R0
STR R0, [R1, #4]       ; Memoire[R1 + 4] <- R0
STR R0, [R1, R2]       ; Memoire[R1 + R2] <- R0
STR R0, [R1], #4       ; Memoire[R1] <- 0, R1 <- R1 + 4
```

Accès mémoire avec PC

- On peut aussi se servir de PC pour accéder à la mémoire

```
LDR Rd, [PC, #16]  ; Rd = Memoire[PC + 16]
```

- On se rappelle:
  - PC contient l'adresse de l'instruction courante **+ 8**
    - PC est "en avance": il pointe 2 instructions plus loin.
  - Donc, dans l'exemple ci-haut, si l'instruction courante est à l'adresse 0x80, nous aurons

```
0x80     LDR Rd, [PC, #16]  ; Rd = Memoire[(0x80+8) + 16]
```

Démonstration

(Adressage en ARM)

Décalage de bits

- **LSL (Logical Shift Left)**: décale les bits vers la gauche et met des zéros à droite. Décaler d'un bit vers la gauche équivaut à multiplier par 2.

- **LSR (Logical Shift Right)**: décale les bits vers la droite et met des 0 à gauche. Décaler d'un bit vers la droite équivaut à diviser un nombre *non-signé* par 2.

- **ASR (Arithmetical Shift Right)**: décale les bits vers la droite et copie le bit le plus significatif à gauche. Décaler d'un bit vers la droite en conservant le bit de signe équivaut à diviser un nombre *signé* par 2.

- **ROR (Rotate Right)**: décale les bits vers la droite.