UNIVERSITE Joseph FOURIER, Grenoble
Licence de Sciences et technologie - Parcours MIN, INF, BIN et MIN international

Examen UE INF241 : Introduction aux Architectures Logicielles et Matérielles

Première session 2012-2013, 16 mai 2013
Durée 2 h
Documents, calculettes, téléphones portables non autorisés
Le barême est donné à titre indicatif
En annexe, un résumé des instructions du processeur ARM et des procedures de la bibliotheque `es.s`
Ne faire la question bonus (Question 2.5) que si le reste est terminé.

1 Structures de contrôle et fonction en ARM (12 points)

On considère la zone mémoire suivante :

```
 1: .data
 2: m:      .asciz "Entrer un nombre"
 3:         .balign 4
 4: tab:    .word 5
 5:         .word 19
 6:         .word 23
 7:         .word 45
 8:         .word 76
 9:         .word 105
10:         .word 119
11:         .word 123
12:         .word 145
13:         .word 176
14:
15: .bss
16:         x: .skip 4
```

1. Rappeler l'effet de la directive `.balign 4`. **(0,25 point)**
2. Rappeler la différence entre le segment `data` et le segment `bss`. **(0,25 point)**
3. Donner le code permettant de saisir au clavier un nombre et de le stocker dans la variable `x` (implantée dans le segment `bss`). **(0,5 point)**

Nous rappelons ci-dessous le code de l'algorithme de la recherche dichotomique de `val` dans un tableau `tab` de 10 éléments indicé de 0 à 9 et trié par ordre croissant :

```
 1: val := x;
 2: deb := 0;
 3: fin := 9;
 4: milieu := (deb+fin)/2;
 5: tant que deb <= fin et tab[milieu] != val faire
 6:      si val > tab[milieu] alors
 7:            deb := milieu+1
 8:      sinon
 9:            fin := milieu-1
10:      fin si
11:      milieu := (deb+fin)/2;
12: fin tant que
```

ATTENTION : Les indices sont des relatifs, car si on cherche une valeur inférieure à la plus petite du tableau, `fin` vaut -1 à la terminaison de l'algorithme. Nous supposons aussi que les valeurs du tableau sont des entiers relatifs.

4. Traduire en ARM le code de l'algorithme précédent. Les variables `val`, `deb`, `fin` et `milieu` seront implantées dans les **registres** $r2$, $r3$, $r4$ et $r5$. Cependant, nous rappelons que la variable `x` est implantée dans la zone `bss`. **(3 points)**

L'algorithme de recherche dichotomique précédent peut être transformé en une fonction récursive :

```
 1: fonction Dicho(val: entier, T: adresse d'un tableau, deb: entier, fin: entier): résultat entier
 2:
 3: milieu, D, F : variables locales entières;
 4:
 5: si deb > fin alors
 6:      retourner -1;
 7: sinon
 8:      milieu := (deb+fin)/2;
 9:      si val = T[milieu] alors
10:               retourner milieu;
11:      sinon
12:               D=deb;
13:               F=fin;
14:               si val < T[milieu] alors
15:                      F:=milieu-1;
16:               sinon
17:                      D:=milieu+1;
18:               fin si
19:               retourner Dicho(val,T,D,F);
20:      fin si
21: fin si
```

ATTENTION : Les paramètres, le résultat et les variables locales de `Dicho` sont placés dans la pile.

À partir de maintenant, nous supposons que :

- `x` = 23,
- Le segment `data` est stocké à partir de l'adresse `0x40C0` et
- Le segment `bss` est stocké à partir de l'adresse `0x80A4`.

5. Dessiner l'état de la pile au moment de l'appel `Dicho(x,tab,0,9)`. Vous ferez notamment apparaître les noms des paramètres effectifs, ainsi que leurs valeurs respectives. Nous rappelons que `x` est définie dans le segment `bss` et `tab` dans le segment `data`. **(1 point)**
6. En supposant l'existence de la fonction `Dicho`. Donner la séquence d'instructions ARM permettant d'appeler `Dicho(x,tab,0,9)`, puis d'afficher le résultat dans une procédure principale (`main`). **(2 points)**
7. Dessiner l'état de la pile lorsque l'appel `Dicho(x,tab,0,9)` atteint la ligne 19 (juste avant le branchement `bl`). Vous ferez notamment apparaître les noms des paramètres effectifs, ainsi que leurs valeurs respectives. **(1 point)**
8. Donner le code ARM de la fonction `Dicho(val,T,deb,fin)`. Ci-dessous, nous rappelons les différentes étapes que vous devrez suivre. **(4 points)**

```
 1: sauvegarde lr
 2: sauvegarde fp
 3: mise en place du nouveau fp
 4: création des variables locales milieu, D et F
 5: sauvegarde des variables temporaires (registres)
 6: corps de fonction
 7: restauration des registres utilisés comme variables temporaires
 8: libération des variables locales
 9: restauration de l'ancien fp
10: restauration de lr
11: retour
```

2 Processeur à pile (12 points)

2.1 Calculatrices extraterrestres (sans questions)

La représentation des expressions algébriques peut-être faite en utilisant plusieurs notations différentes. Par exemple, on peut représenter les expressions algébriques sous la forme d'arbres. Ainsi, l'expression `(1 + 2)` prise sous la forme traditionnelle est représentable par l'arbre proposé en Figure 1.

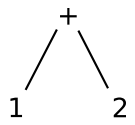

FIGURE 1 – Arbre représentant l'expression $(1 + 2)$

À partir de cet arbre, on peut retrouver facilement les 3 notations équivalentes de cette expression en utilisant différents types de parcours.

notation	
préfixe	`+ 1 2`
infixe	`1 + 2`
postfixe	`1 2 +`

TABLE 1 – Notations algébriques

Les calculatrices classiques utilisent la notation infixée (traditionnelle). Dans les années 1960, des calculatrices utilisant la notation postfixée ont été conçues : Hewlett-Packard 9100A. Une longue lignée de calculatrices a ensuite suivi. On appelle aussi cette notation, la notation *polonaise inverse*. Cette notation est intéressante car elle est dépourvue de parenthèses. Par exemple, la représentation de l'expression algébrique `2 * (3 - 4)` dans laquelle il est impossible de retirer les parenthèses devient `2 3 4 - *`. L'interprétation d'une expression en polonaise inverse se fait de gauche à droite. À chaque fois qu'une opération est rencontrée, elle est évaluée avec les deux opérandes à sa gauche. Ainsi, on remplace les deux opérandes et l'opération par le résultat dans l'expression d'origine. Par exemple, notre expression `2 3 4 - *` est évaluée de la manière suivante : on parcourt de gauche à droite jusqu'à rencontrer la première opération, ici l'opération $-$. On évalue l'opération $-$ avec les deux opérandes à sa gauche, ici 3 et 4. Ainsi, `2 (3 4 -) *` devient `2 -1 *`. On continue le parcours vers la droite depuis le -1, on arrive à la multiplication que l'on effectue. `(2 -1 *)` devient alors -2, le résultat. On obtient bien -2 comme pour l'expression en notation infixée.

2.2 Processeur à pile

Une famille de processeurs, nommée les *processeurs à pile*, a un modèle de programmation assez semblable. Dans ces processeurs, on n'utilise pas de registre à l'intérieur du processeur pour stocker les valeurs des opérandes, mais une pile. Nous allons maintenant étudier un processeur à pile simplifié, STACK. L'accès à la pile de STACK se fait via deux primitives : `push`(x), qui ajoute x au sommet de la pile ; et `pop`(), qui enlève le sommet de la pile et le renvoie. Dans ce type de processeur, toutes les instructions font donc des opérations en prenant leurs valeurs depuis la pile (`pop()`), et en plaçant leur résultat (x) sur la pile (`push(`x`)`). La table 2 donne un synoptique des instructions de ce processeur. Attention, l'ordre d'évaluation des opérandes est important : il faut suivre l'ordre de lecture donné dans la table. Par exemple, l'opération `sub` sur la pile donnée en Figure 2

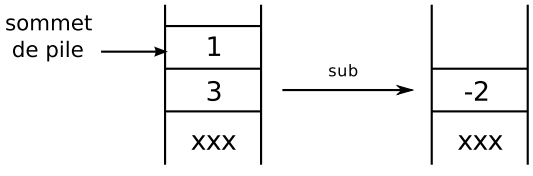

FIGURE 2 – Évolution de la pile de STACK lors de l'exécution de l'instruction `sub`

consiste à dépiler 1 (résultat du premier `pop`()), puis dépiler 3 (résultat du deuxième `pop`()), et enfin empiler le résultat de l'opération $1 - 3$, c'est-à-dire, -2.

Instruction	codage	opération
`ldx a`	0x01 + a	`push`($mem[a]$)
`stx a`	0x02 + a	$mem[a] \leftarrow$ `pop`()
`ldi i`	0x03 + i	`push`($mem[pc+1]$)
`jmp a`	0x04 + a	$pc \leftarrow a$
`jnz a`	0x08 + a	if `pop`() $\neq 0$ then $pc \leftarrow a$
`jz a`	0x09 + a	if `pop`() $= 0$ then $pc \leftarrow a$
`add`	0xDE	`push`(`pop`() + `pop`())
`sub`	0xBE	`push`(`pop`() − `pop`())
`mult`	0xAD	`push`(`pop`() × `pop`())
`ld`	0x80	`push`(mem[`pop`()])
`st`	0xA0	mem[`pop`()] ← `pop`()
`eq`	0xEA	`push`(`pop`() = `pop`())
`ne`	0xFF	`push`(`pop`() ≠ `pop`())
`band`	0xBA	`push`(`pop`() & `pop`()) (et bit-à-bit)
`borr`	0xBB	`push`(`pop`() | `pop`()) (ou bit-à-bit)
`not`	0xCD	`push`(!`pop`()) (complément)
`dup`	0xEF	$x \leftarrow$ `pop`(); `push`(x); `push`(x)

TABLE 2 – Instructions du processeur à pile

Le processeur STACK est un processeur **16 bits** grand boutiste (big endian) (ainsi, chaque élément de la pile est stocké sur deux octets) dont les instructions sont codées sur un octet. Certaines instructions sont suivies d'un opérande de type valeur immédiate (`i`) qui est codée sur un octet supplémentaire ou d'un opérande de type adresse (`a`) qui est codée sur deux octets, ce ou ces octets supplémentaires étant placés juste après le code de l'instruction. Par exemple, `ldi 10` se code sur deux octets : 0x03, suivi de 0x0A. Il n'y a pas de contrainte d'alignement sur les instructions et les adresses et constantes immédiates. Les instructions `eq` et `ne` empilent des résultats « à la mode C ». Si la condition est vérifiée, elles empilent la valeur entière 1, sinon elles empilent la valeur entière 0.

1. Donner le programme en langage d'assemblage STACK calculant les expressions suivantes (**2 points**) :
   a. `(1 + 2) * (7 - 8)`
   b. `-x & x`. Vous supposerez que `x` est déjà en sommet de pile. De plus, vous n'avez le droit d'utiliser qu'une seule fois l'instruction `ldi`.

Voici un programme en langage d'assemblage STACK :

```
.data
         .balign 2
foo:     .skip 20
.text
mystere: ldi 0
         jmp Label1
Label0:  dup
         dup
         ldi 2
         mult
         ldx Label3
         add
         st
         ldi 1
         add
Label1:  dup
         ldi 10
         eq
         jz Label0
         jmp fin
```

```
Label3: .short foo
fin:
```

2. Donner la représentation mémoire du programme en langage machine STACK en supposant que la section `.text` est chargée à l'adresse `0x1000` et que la section `.data` est chargée à l'adresse `0x8000`. N'oubliez pas de préciser l'adresse de chaque instruction et opérande. (**2 points**)
3. Dessiner l'évolution de la pile du processeur STACK au cours de l'exécution du programme. (**4 points**)
4. Que fait ce programme ? (**1 point**)

L'architecture du processeur STACK, ou plus particulièrement sa partie opérative est donnée à la figure 3. Dans ce processeur, l'UAL est connectée à deux registres *top* et *pot* qui stockent temporairement les valeurs des opérandes de l'opération à effectuer. Ces registres sont alimentés par la pile. Ils peuvent donc contenir le sommet et le sous-sommet de pile. Dans le cas d'une opération à un opérande, c'est *pot* qui contient l'opérande (le sommet de pile). Dans le cas d'une opération à deux opérandes, *top* contient le premier opérande et *pot* le deuxième opérande. Ainsi *top* contient le sommet de pile et *pot* le sous-sommet de pile.

On commande la mémorisation d'une valeur dans ces registres par les commandes `wet` (pour *top*) et `wep` (pour *pot*) : 0 signifiant conserver la valeur précédente et 1 signifiant mémoriser la nouvelle valeur.

De la même manière, les commandes `wepc`, `wei`, `wea` permettent le chargement de valeurs dans les registres *pc* (compteur ordinal), *IR* (registre contenant l'instruction courante) et *addr* (adresse accédée en mémoire).

La pile n'est pas en mémoire centrale, mais est réaliser par un bout de matériel interne au processeur. Sa profondeur reste donc limitée par ce matériel, mais suffisamment grande pour ne pas avoir à s'en préoccuper ici. On commande cette pile en utilisant `stack_op` qui peut prendre trois valeurs : `rien` pour ne rien faire, `stack_push` pour empiler une valeur présentée en entrée sur la pile et `stack_pop` pour dépiler le sommet de pile et présenter la valeur sur la sortie de la pile.

Pour charger le registre *pot* avec la valeur du sommet de pile, il faut donc mettre `stack_op` à la valeur `stack_pop` et placer la commande `wep` à 1. De même, pour charger le registre *pot* avec la valeur du sous-sommet de pile et *top* avec le sommet de pile, on s'y prend donc en deux fois. Le premier cycle est le même que précédemment, où le sommet de pile est stocké dans *pot*. Dans un second cycle, on active `wep` et `wet` à 1 ; et on met `stack_op` à la valeur `stack_pop`. Ce qui a pour effet de déplacer le sommet de pile de *pot* vers *top* et de mettre le sous-sommet de pile dans *pot*.

Pour permettre de choisir la source de la valeur à empiler, on a la possibilité d'utiliser la commande `src` qui peut prendre les valeurs `UALo` pour empiler le résultat de l'opération effectuée avec l'UAL ou `DONo` pour empiler la valeur lue en mémoire.

L'UAL quant à elle est commandée en utilisant `UAL_op` qui peut prendre pour valeurs : `UAL_add`, `UAL_sub`, `UAL_mult`, `UAL_band`, `UAL_borr`, `UAL_eq`, `UAL_ne`, `UAL_not` en fonction de l'opération à effectuer.

La commande `id` permet de choisir l'adresse à présenter sur le bus d'adresse mémoire (`adPC` pour le compteur ordinal, ou `adStack` pour la valeur dépilée).

L'exécution d'une instruction `not` se décompose ainsi de la manière suivante :

Cycle 1 : `wep` ← 1, `stack_op` ← `stack_pop`, c'est-à-dire, la tête de pile est dépilée dans *pot*

Cycle 2 : `UAL_op` ← `UAL_not`, `src` ← `UALo`, `stack_op` ← `stack_push`, c'est-à-dire, le complément de la valeur dans *pot* est empilé.

Bonus : 5. Donner cycle par cycle l'exécution d'une instruction `sub`. On ne soucie ici que de l'exécution de l'instruction, les étapes de fetch et de décodage sont considérées comme déjà réalisées. (**3 points**)

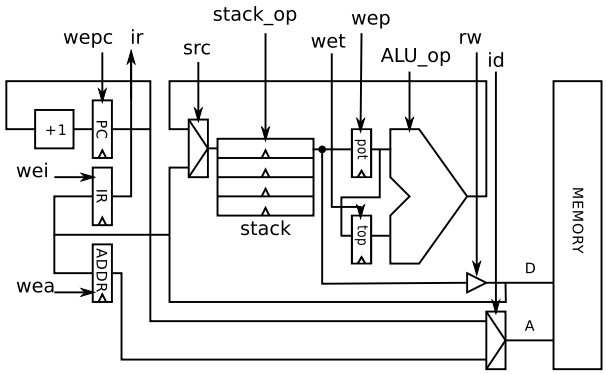

FIGURE 3 – Partie Opérative du processeur STACK

3 ANNEXE I : instructions du processeur ARM

Nom	Explication du nom	Opération	remarque
`AND`	AND	et bit à bit	
`EOR`	Exclusive OR	ou exclusif bit à bit	
`SUB`	SUBstract	soustraction	
`RSB`	Reverse SuBstract	soustraction inversée	
`ADD`	ADDition	addition	
`ADC`	ADdition with Carry	addition avec retenue	
`SBC`	SuBstract with Carry	soustraction avec emprunt	
`RSC`	Reverse Substract with Carry	soustraction inversée avec emprunt	
`TST`	TeST	et bit à bit	pas `rd`
`TEQ`	Test EQuivalence	ou exclusif bit à bit	pas `rd`
`CMP`	CoMPare	soustraction	pas `rd`
`CMN`	CoMpare Not	addition	pas `rd`
`ORR`	OR	ou bit à bit	
`MOV`	MOVe	copie	pas `rn`
`BIC`	BIt Clear	et not bit à bit	
`MVN`	MoVe Not	not (complément à 1)	pas `rn`
`Bxx`	Branchement		xx = condition Cf. table ci-dessous
`BL`	Branchement à un sous-programme		adresse de retour dans r14=LR
`LDR`	"load"		
`STR`	"store"		

L'opérande source d'une instruction `MOV` peut être une valeur immédiate notée `#5` ou un registre noté `Ri`, `i` désignant le numéro du registre. Il peut aussi être le contenu d'un registre sur lequel on applique un décalage de `k` bits ; on note `Ri, DEC #k`, avec `DEC` $\in$ {`LSL`, `LSR`, `ASR`, `ROR`}.

4 ANNEXE II : codes conditions du processeur ARM

La table suivante donne les codes de conditions arithmétiques `xx` pour l'instruction de branchement `Bxx`.

mnémonique	signification	condition testée
EQ	égal	Z
NE	non égal	$\overline{Z}$
CS/HS	$\geq$ dans N	C
CC/LO	$<$ dans N	$\overline{C}$
MI	moins	N
PL	plus	$\overline{N}$
VS	débordement	V
VC	pas de débordement	$\overline{V}$
HI	$>$ dans N	$C \wedge \overline{Z}$
LS	$\leq$ dans N	$\overline{C} \vee Z$
GE	$\geq$ dans Z	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
LT	$<$ dans Z	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
GT	$>$ dans Z	$\overline{Z} \wedge ((N \wedge V) \vee (\overline{N} \wedge \overline{V}))$
LE	$\leq$ dans Z	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
AL	toujours	

5 ANNEXE III : fonctions d'entrée/sortie

Nous rappelons les principales fonctions d'entrée/sortie du fichier `es.s`.

Les fonctions d'affichages :
- `bl EcrHexa32` affiche le contenu de `r1` en hexadécimal.
- `bl EcrZdecimal32` affiche le contenu de `r1` en décimal sous la forme d'un entier relatif de 32 bits.
- `bl EcrZdecimal16` affiche le contenu de `r1` en décimal sous la forme d'un entier relatif de 16 bits.
- `bl EcrZdecimal8` affiche le contenu de `r1` en décimal sous la forme d'un entier relatif de 8 bits.
- `bl EcrNdecimal32` affiche le contenu de `r1` en décimal sous la forme d'un entier naturel de 32 bits.
- `bl EcrNdecimal16` affiche le contenu de `r1` en décimal sous la forme d'un entier naturel de 16 bits.
- `bl EcrNdecimal8` affiche le contenu de `r1` en décimal sous la forme d'un entier naturel de 8 bits.

Les fonctions de saisie clavier :
- `bl Lire32` récupère au clavier un entier 32 bits et le stocke à l'adresse contenue dans `r1`.
- `bl Lire16` récupère au clavier un entier 16 bits et le stocke à l'adresse contenue dans `r1`.
- `bl Lire8` récupère au clavier un entier 8 bits et le stocke à l'adresse contenue dans `r1`.
- `bl LireCar` récupère au clavier un caractère et stocke son code ASCII à l'adresse contenue dans `r1`.