Examen
Architecture Logicielle et Matérielle
RICM3

P. Sicard

Durée : 2 heures
Les parties *hard* et *soft* sont indépendantes et de durée à peu près égale.
Tous documents autorisés. Ordinateurs interdits. Un barème approximatif est donné.

1 Partie *SOFT* : Analyse du résultat d'une compilation en assembleur ARM (10 points)

L'annexe 1 donne le début d'un programme C où certaines parties sont omises (signalés par trois points ...) L'annexe 3 donne le résultat de la compilation de ce programme C en langage d'assemblage ARM (compilateur *gcc*).
Le but du jeu est de comprendre la traduction du compilateur C en langage d'assemblage ARM et de retrouver la fin du programme C ; pour cela répondre précisément aux questions ci dessous.
Vous pouvez rendre l'annexe 3 annotée avec votre copie.

RAPPELS pouvant vous servir :
- Les instructions assembleur du programme commencent à l'étiquette `main`.
- `sp` est le registre pointeur de sommet de pile
- `fp` est le pointeur de la base du contexte d'une procédure
- `ldr r0,.L6` charge dans `r0` la valeur sur 4 octets se trouvant à l'étiquette `.L6`
- L'instruction mvn r2, #19 affecte r2 par le complément booléen de 19, c'est-à-dire la valeur –20 codée en complément à 2.
  Exemple sur 4 bits pour la valeur 3 : complément_booléen(0011) = 1100 = -4 (codé en complément à 2).

1. Expliquez l'effet de l'instruction *stmfd sp !, {fp, ip, lr, pc}*. Pourquoi cette instruction est elle exécutée au début du programme ?

2. Traduisez cette instruction en utilisant seulement les instructions *sub* et *str*.

3. Dessiner l'état de la pile après l'exécution de l'instruction *sub sp, sp, #28*. Précisez les cases pointées par *sp* et *fp*.

4. Donnez et commentez la suite d'instructions d'assemblage qui réalise l'instruction en langage C :
   `for (i=0; i<N; i++)`
   Précisez les étiquettes utilisées et le calcul de la condition réalisée.
   En déduire l'emplacement de la variable *i* sur la pile. Complétez le dessin de la pile.

5. Donnez et commentez les instructions ARM qui réalisent `tab[i]=1;` . Comment est calculée l'adresse de `tab[i]` ? Donnez sa valeur en fonction de *fp* et de *i*. En déduire l'emplacement des éléments du tableau *tab* sur la pile. Complétez le dessin de la pile.

6. Que faut-il changer dans ce programme ARM si le tableau *tab* comporte 1000 éléments ?

7. Comment sait-on à la vue du programme en ARM que les éléments du tableau sont de type *char* ? Que faut il changer dans le programme ARM si `tab` est un tableau d'entiers (sur 4 octets) ?

8. Expliquez comment est traduit le premier `while (...)` en donnant les étiquettes utilisées et les instructions qui calculent la condition. Donnez cette condition en complétant le programme C.

9. Quelles sont les instructions ARM qui traduisent l'appel à la fonction `printf` apparaissant dans le programme C.
   Où sont stockés les deux paramètres de la fonction `printf` ? Quelles sont les valeurs de ces deux paramètres lors de l'exécution du programme au moment de l'appel à `printf` ? Que contiennent les registres *r0* et *r1* à ce moment là (adresse, constante, valeur de variable ...) ?

10. Traduisez l'instruction `bl printf` en utilisant les instructions `ba` et `mov`. On supposera que le pointeur de programme contient l'adresse de l'instruction qui suit l'instruction en cours d'exécution.

11. Retrouvez et commentez les instructions ARM qui traduisent `if (tab[i])` ?

12. Retrouvez la condition du deuxième `while` apparaissant dans le programme et les étiquettes utilisées pour le traduire.

13. Retrouvez les instructions en langage C du corps de ce `while` en commentant les instructions assembleur correspondantes.

14. Complétez la fin du programme C.

15. Expliquez l'effet de l'instruction *ldmea fp, {fp, sp, pc}* qui suit l'étiquette *.L7*. Quelle est la valeur de *pc* (Compteur Programme) après l'exécution de cette instruction ?
Traduisez cette instruction en utilisant seulement les instructions *sub* et *ldr*.

16. Si les variables *i* et *j* étaient déclarées en entier non signé (*unsigned int*) quelles modifications cela engendreraient dans cette traduction en assembleur ARM ?

2 Partie *HARD* : Réalisation d'une machine algorithmique (10 points)

Soit l'algorithme suivant :

```
Lexique :
A et B : deux entiers > 0 ; { les données } ;
Q, R : deux entiers ≥ 0; { les résultats }
 X : un entier
Algorithme :
X=1 ;
Q=A ;
Tant que X ≤ A faire
Debut
si X < Q alors X= X+B ;
Q= Q -1 ;
Fin
R=X-A;
```

Nous voulons réaliser un circuit effectuant le même calcul. Nous choisissons une architecture *partie contrôle, partie opérative*. On ne s'occupe pas des entrées/sorties du circuit. *A* et *B* sont donnés dans deux registres. On veut avoir *Q* et *R* dans deux registres spécifiques en fin de calcul.

Répondre aux questions suivantes :

1. Soit le *node* Lustre décrivant une bascule D :

   ```
   node bascule ( d, reset, set, char : bool )
   returns ( q : bool );
   ```

   Cette bascule D sensible au front possède une entrée de remise à 0 (*reset*), une entrée de remise à 1 (*set*) ainsi qu'un signal de chargement *char*. En vous servant de cette bascule, donnez la description en Lustre d'un registre sur n bits possédant un signal de chargement *char* et une entrée *misea1* permettant d'initialiser ce registre à la valeur 1 :

   ```
   node registre (const n: int; e: bool^n; char, misea1: bool)
   returns (s: bool^n);
   ```

2. Enumérez les variables et les opérations apparaissant dans cet algorithme. Ne pas oublier les opérations permettant le calcul des conditions.

3. Complétez les connexions de la partie opérative donnée dans l'annexe 2 afin qu'elle puisse effectuer tous les calculs et affectations apparaissant dans le programme. Précisez les noms des signaux de commandes manquants. Appliquez vous à ne donner que les connexions et signaux strictement nécessaires. Vous pouvez rendre cette annexe avec la copie d'examen.

   Remarque : Le signal *setX* permet d'initialiser à 1 le registre *X*.
   Le signal *ChX* permet de charger le *Bus Resultat* dans le registre *X* au front montant d'une horloge (n'apparaissant pas sur le dessin).
   Ce registre est donc identique à celui réalisé à la première question.

4. Quelles sont les opérations que doit être capable d'exécuter l'UAL de cette partie opérative ? Précisez le codage des commandes de l'UAL (on notera *OpUAL*) pour chacune des opérations pouvant être effectuées.

5. Comment peut être calculée simplement la condition $X \leq A$ dans cette UAL sachant que les entiers sont codés en base 2 (non signés).

6. Comment peut être calculée simplement la condition $X<Q$ dans cette UAL sachant que les entiers sont codés en base 2 (non signés).

7. A l'aide de multiplexeurs et d'un seul additionneur (délivrant les flags *Z, N, C* et *V*) donnez le schéma de cette UAL. Expliquez votre solution.
   Ne pas oublier le calcul des conditions des questions précédentes.

8. Donnez le graphe de l'automate permettant de réaliser la partie contrôle de ce circuit. On associera dans un premier temps les instructions de l'algorithme aux états de cet automate.

9. Donnez les valeurs des signaux de commande apparaissant sur la partie opérative pour les états où sont effectués $Q=A$; $X= X+B$;$Q=Q-1$ et le calcul de la condition $X \leq A$.

3 Annexes

3.1 Annexe 1 : Programme C à compléter

```
#define N 20
main()
{
      char tab[N];
      int i, j;

      for (i=0; i<N; i++)
             tab[i]=1;
      i=2;
      while (...)
          {
         if (tab[i])  /* en C toute valeur differente de 0 est VRAI */
                  {
                     printf(" %d est premier\n", i);
                     ....
                     while (...)
                    ....
  }
```

3.2 Annexe 2

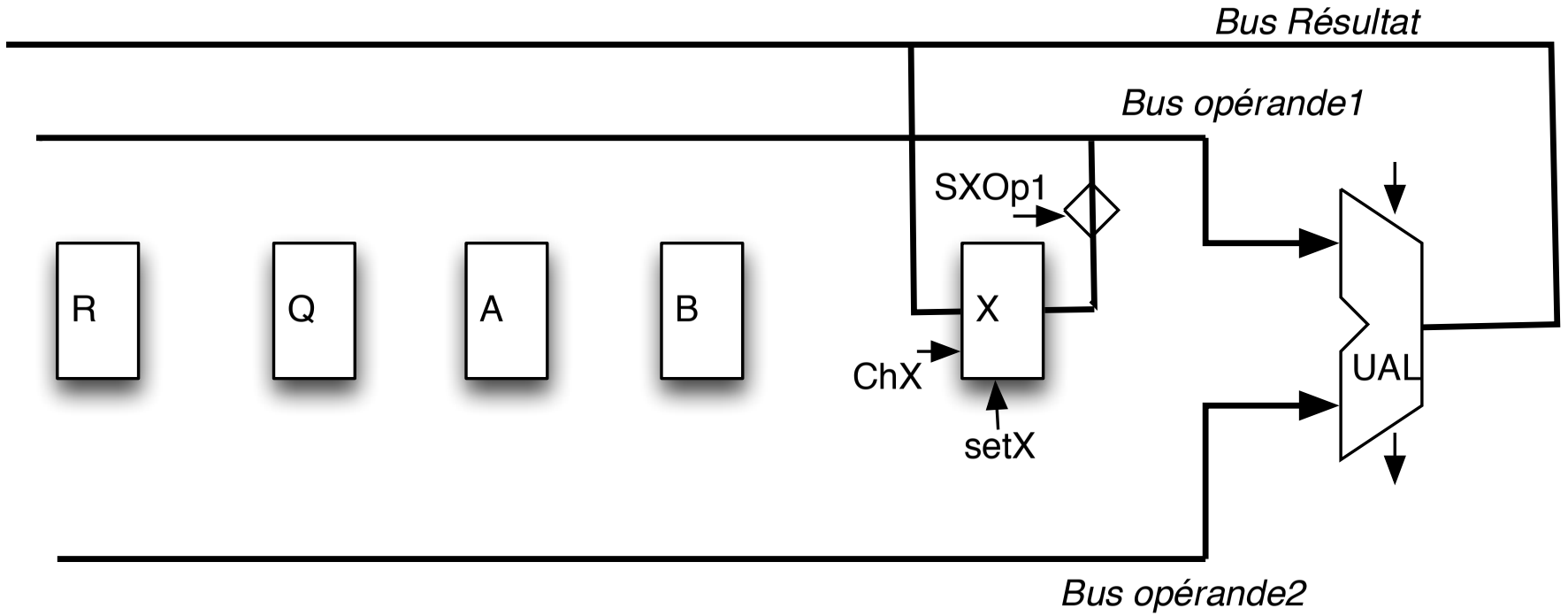

Figure 1 – Partie opérative à compléter

3.3 Annexe 3 : Résultat de la compilation du programme de l'annexe1

```
.file « prog.c »
          .section
          .rodata
          .align 2
.LC0:
          .ascii «  %d est premier\n\000 »
.text
          .align 2
          .global main
          .type main,function
main:
          mov  ip, sp
          stmfd sp!, {fp, ip, lr, pc}
          sub fp, ip, #4
          sub sp, sp, #28
          mov r3, #0
          str r3, [fp, #-36]
.L2:
          ldr r3, [fp, #-36]
          cmp r3, #19
          ble .L5
          b .L3
.L5:
          mvn  r2, #19
          ldr r3, [fp, #-36]
          sub r1, fp, #12
          add r3, r1, r3
          add r2, r3, r2
          mov  r3, #1
          strb r3, [r2, #0]
          ldr r3, [fp, #-36]
          add r3, r3, #1
          str r3, [fp, #-36]
          b .L2
.L3:
          mov r3, #2
          str r3, [fp, #-36]
.L6:
          ldr r3, [fp, #-36]
          cmp r3, #19
          ble .L8
          b .L7
.L8:
```

```
          mvn r2, #19
          ldr r3, [fp, #-36]
          sub r1, fp, #12
          add r3, r1, r3
          add r3, r3, r2
          ldrb r3, [r3, #0] @ zero_extendqisi2
          cmp  r3, #0
          beq .L9
          ldr r0, .L13
          ldr r1, [fp, #-36]
          bl printf
          ldr r3, [fp, #-36]
          mov r3, r3, asl #1
          str r3, [fp, #-40]
.L10:
          ldr r3, [fp, #-40]
          cmp r3, #19
          ble .L12
          b .L9
.L12:
          mvn  r2, #19
          ldr r3, [fp, #-40]
          sub r1, fp, #12
          add r3, r1, r3
          add r2, r3, r2
          mov r3, #0
          strb r3, [r2, #0]
          ldr r2, [fp, #-40]
          ldr r3, [fp, #-36]
          add r3, r2, r3
          str r3, [fp, #-40]
          b .L10
.L9:
          ldr r3, [fp, #-36]
          add r3, r3, #1
          str r3, [fp, #-36]
          b .L6
.L7:
          mov r0, r3
          ldmea fp, {fp, sp, pc}
.L14:
          .align 2
.L13:
          .word .LC0
.Lfe1:
          .size main,.Lfe1-main
.ident "GCC: (GNU) 3.2.2"
```