Examen final d'assembleur–compilation

ÉNSIIE, semestre 3

vendredi 17 décembre 2021

Durée : 3h.
Tout document personnel autorisé (pas de prêt entre voisins).
Ce sujet comporte 7 exercices indépendants, qui peuvent être traités dans l'ordre voulu.
Il contient 4 pages.
Certaines questions, précédées par le symbole ($\star$) sont plus difficiles et pourront être traitées à la fin.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points.
Il va de soi que toute réponse devra être justifiée.

Exercice 1 : Assembleur RISC-V (3 points)

1. Expliquer pourquoi on doit parler *d'un* langage assembleur, et pas *du* langage assembleur.
2. Écrire une suite d'instructions RISC-V qui mettent dans le registre `a0` le reste par la division euclidienne du contenu de `a0` par celui de `a1`. On n'utilisera pas l'instruction `rem` (par exemple parce qu'on cible RV32I et non RV32IM), mais on codera l'algorithme suivant :

   ```
   tant que a0 >= a1
     a0 := a0 - a1
   ```

   On supposera pour l'instant que `a0` est positif ou nul, et que `a1` est strictement positif.
3. Modifier votre code pour pouvoir traiter les cas où `a0` et `a1` ne sont pas positifs. On supposera toutefois toujours que `a1` n'est pas nul.

   La valeur absolue du reste devra être égale au reste des valeurs absolues de `a0` et `a1`, et son signe sera égal à celui de leur produit (donc positif ssi `a0` et `a1` sont de même signe).
4. Transformer votre code en fonction dont le label d'entrée sera `reminder`, en pensant à mettre un retour à l'appelant.
5. Écrire une suite d'instructions qui appellent la fonction `reminder` pour mettre dans `a0` le produit du reste de 10 par 3 et du reste de 7 par 5 (ce qu'on écrirait en C `(10 % 3) * (7 % 5)`).
6. Comment faire pour pouvoir retrouver la valeur initiale de `ra` après les appels à `reminder` ?

Exercice 2 : Syntaxe (2,5 points)

Donner l'arbre de syntaxe abstraite des instructions Pseudo Pascal suivantes, ou expliquer pourquoi elles ne sont pas syntaxiquement correctes :

1. `a := c + b * 2`
2. `a - c := b * 2`
3. `t[i] := t[i] + 5`
4. `while a < 2 and b > 4 do a = a + 1`
5. `while a < 2 do if b > 4 then a := a + 1`

Exercice 3 : Analyse syntaxique (2,5 points)

On considère la grammaire suivante (symbole de départ S) :

$$\begin{aligned} S &\to aUb \\ U &\to \quad \text{(mot vide)} \\ &\mid cU \end{aligned}$$

1. Quel est le langage reconnu par cette grammaire ?
2. Construire l'automate déterministe LR(0) pour cette grammaire.
3. Cette grammaire est-elle LR(0) ?
4. ($\star$) Cette grammaire est-elle ambiguë ?
5. Proposer une grammaire LR(0) reconnaissant le même langage. On justifiera qu'elle est bien LR(0).

Exercice 4 : Réécriture (2,5 points)

Les système de réécriture suivants terminent-ils, sont-ils confluents ? Justifiez vos réponses.

1. $\texttt{li}_0 \to \texttt{li}_0$
2. $\texttt{li}_{128} \to \texttt{li}_{-128} \qquad \texttt{li}_{128} \to \texttt{li}_{127}$
3. $\texttt{add}(X, Y) \to \texttt{add}(Y, X) \qquad \texttt{add}(\texttt{li}_0, X) \to X \qquad \texttt{add}(\texttt{li}_1, X) \to \texttt{addi}_1(X)$
4. ($\star$) $\texttt{not}(\texttt{zero}(X)) \to \texttt{li}_1 \qquad \texttt{zero}(Y) \to \texttt{li}_0 \qquad \texttt{li}_1 \to \texttt{not}(\texttt{li}_0)$

Exercice 5 : Graphe de flot de contrôle (3 points)

Soit le programme Pseudo-Pascal suivant :

```
1 x := x * 4 + 1;
2 y := x *  4 + 1;
3 z := y * (x * 4 + 1)
```

1. Faire la sélection d'instruction pour transformer ce programme en UPP. On utilisera les règles de réécriture

$$\texttt{add}(X, \texttt{li}_k) \to \texttt{addi}_k(X)$$
$$\texttt{mul}(X, 2^k) \to \texttt{slli}_k(X)$$

2. Transformer le programme en RTL.
3. Calculer la valeur symbolique des pseudo-registres au fur et à mesure des instructions.
4. Supprimer les calculs redondants et donner le programme RTL résultant.

Exercice 6 : Convention d'appel (3 points)

On considère la fonction Pseudo-Pascal suivante :

```
1  function Ackermann(m, n : integer) : integer;
2  var tmp : integer;
3  begin
4    if m = 0
5    then Ackermann := n + 1
6    else if n = 0
7      then Ackermann := Ackermann(m - 1, 1)
8      else begin
9        tmp := Ackermann(m, n - 1);
10       Ackermann := Ackermann(m - 1, tmp)
11     end
12 end;
```

On suppose qu'on utilise la convention d'appel RISC-V comme vue en cours. On supposera que la variable locale `tmp` de la fonction `Ackermann` est stockée dans le registre sauvegardé par l'appelant `t0`.

1. Quels registres `Ackermann` doit-elle sauvegarder ? En déduire la trame de `f`.
2. Pour expliciter la convention d'appel, quelles instructions faut-il ajouter :
   a) Au début de `Ackermann` ?
   b) Avant l'appel à `Ackermann` à la ligne 11 ?
   c) Après l'appel à `Ackermann` à la ligne 11 ?
   d) Avant l'appel à `Ackermann` à la ligne 13 ?
   e) Après l'appel à `Ackermann` à la ligne 13 ?
   f) Avant l'appel à `Ackermann` à la ligne 14 ?
   g) Après l'appel à `Ackermann` à la ligne 14 ?
   h) À la fin de `Ackermann` ?
3. Quels appels récursifs dans `Ackermann` peut-on optimiser ? Comment ?

Exercice 7 : Allocation de registres (3,5 points)

On considère le programme RTL suivant :

```
var %0, %1, %2, %3
entry l0
exit l9
l0: mv %0, a0 -> l1
l1: li s0, 0 -> l2
l2: addi %2, s0, 1 -> l3
l3: bgt %2, %3 -> l4, l7
l4: mv %1, s0 -> l5
l5: mv %3, %1 -> l6
l6: add s0, s0, %2 -> l2
l7: addi a0, %3, -1 -> l8
l8: mv s0, %2 -> l9
```

1. Dessiner le graphe de flot de contrôle correspondant.
2. Donner les variables vivantes en chacun des points du programme, en supposant qu'à la sortie (en `l9`), les variables `a0` et `s0` sont vivantes.
3. Quelles instructions pourraient-elles être éliminées ?
4. Dessiner le graphe d'interférence (avec les arêtes de préférence). On indiquera sur les arêtes le label d'une instruction qui a causé l'interférence ou la préférence.
5. Essayer de 2-colorier ce graphe en appliquant l'algorithme de George et Appel. On détaillera le déroulement de l'algorithme et on justifiera le critère utilisé lors d'une fusion.
6. En utilisant ce 2-coloriage, donner le programme RTL correspondant au programme initial (on utilisera donc les deux registres `a0` et `s0`) et pour lequel on utilise `t0` et `t1` pour sauvegarder ou restaurer la valeur des variables éventuellement spillées. On supprimera les éventuelles instructions `mv` devenues inutiles.