Examen du cours INF108 – Compilation

Cet examen contient trois parties indépendantes. L'examen est calibré de façon à ce que chaque partie puisse rapporter facilement quelques points, il est donc très fortement recommandé de s'attaquer à toutes les parties en terminant par les questions étoilées ★ puis doublement étoilées ★★.

Votre copie doit comporter votre nom de façon claire sur chaque feuille remplie. Il est possible de répondre en francais, anglais, Ocaml, Python, C, pseudo-code ou schéma tant que la réponse est claire. Les erreurs d'orthographe, de grammaire ou de syntaxe ne seront pas pénalisées tant qu'il est clair que la réponse est correcte. Il est conseillé d'accompagner les codes ou schémas par des phrases explicatives ou commentaires pour lever toute ambiguïté.

Pour rappel, une feuille A4 (recto-verso, manuscrite ou imprimée) de notes personnelles est autorisée.

Une partie de la note sera par groupe mais l'examen est individuel, toute communication entre étudiants ou aide extérieure sera lourdement pénalisée (note de 0 et action disciplinaire).

Partie 1 – Circuits

Pour toute cette partie, les circuits attendus doivent être construits en utilisant comme brique de base les portes binaires ET, OU, XOR ainsi que la porte unaire NON ainsi que les tensions ZERO et UN.

Il est autorisé de définir des blocs (en spécifiant bien ce qu'ils calculent) pour les utiliser ensuite. On peut, par exemple, réutiliser la question 1 dans la réponse de la question 2 ou définir un "OU 8 bits" et l'utiliser ensuite.

Pour finir, on utilisera dans cette partie la convention petit-boutiste, c'est à dire qu'un nombre a sur k bits est représenté avec k fils $a_0 ... a_{k-1}$ avec $a = \sum_{0 \leq i < k} a_i \times 2^i$. Par exemple, sur $k = 4$ bits le nombre 14 est représenté par les tensions $t_0 = 0$, $t_1 = t_2 = t_3 = 1$.

Question 1 – Additionneur 1 bit (full-adder) Étant données 3 entrées (aussi appelées fil ou tension dans le cours) a, b et c, proposer un circuit qui calcule les sorties somme et retenue. On rappelle que la somme vaut 1 quand $(a + b + c)$ est impair et que la retenue vaut 1 quand $a + b + c \geq 2$.

> On peut le faire de la façon suivante :
>
> ```
> somme = (a XOR b) XOR c
> retenue = (a ET b) OU (a ET c) OU (b ET c)
> ```
>
> La retenue peut aussi être calculée avec `(a ET (b OU c)) OU (b ET c)`.

Question 2 – Additionneur 4 bits Étant données 8 entrées $a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3$, représentant deux nombres a et b, proposer un circuit qui calcule la somme $c = a + b$, sur 5 bits c_0, c_1, c_2, c_3, c_4.

> Si on note FA le bloc de la question 1 et que l'on note `r,s = FA(a,b,c)` on a :
>
> ```
> (c0, r0) = FA(a0,b0,zero)
> (c1, r1) = FA(a1,b1,r0)
> (c2, r2) = FA(a2,b2,r1)
> (c3, c4) = FA(a3,b3,r2)
> ```
>
> Et on obtient les fils voulus.

Question 3 – Alerte sur flux 1 Pour cette question et la suivante, on rajoute un nouveau type de porte, la porte DELAI. On rappelle que si l'on a $a =$ DELAI b, c'est à dire que a est connecté à la sortie d'une porte DELAI dont b est l'entrée, alors au temps 0, $a = 0$ et pour tout instant $t > 0$, a prend la valeur qu'avait b au temps $t - 1$.

Proposer un circuit qui détecte s'il y a eu 1 instant où l'entrée était à 1. Formellement ce circuit a une entrée f et une sortie a et dont la sortie a vaut 1 aux instants t pour lesquels il y a eu un instant $t' \leq t$ où f valait 1. Remarquer que dès l'instant où $a = 1$ alors a reste toujours à 1 ensuite.

En abusant de la notation équationnelle, on a :

```
a = (DELAI a) OU f
```

Question 4 ★ – Alerte sur flux 8 Proposer un circuit qui détecte s'il y a eu 8 instants où l'entrée était à 1. Formellement ce circuit a une entrée f et une sortie a et dont la sortie a vaut 1 aux instants t pour lesquels il y a eu 8 instants $t_1 ... t_8$ avec $0 \leq t_1 < t_2 < \cdots < t_8 \leq t$ où f valait 1. Remarquer que les t_i ne sont pas forcément consécutifs et que dès l'instant où $a = 1$ alors a reste toujours à 1 ensuite.

En utilisant `c0,c1,c2,c3,c4 = SOMME(a0,a1,a2,a3, b0,b1,b2,b3)` tel que défini à la question 2 et `ALERTE1(f)` comme défini en question 3, on a, en abusant de la notation équationnelle :

```
c0,c1,c2,c3,c4 = SOMME(DELAI c0, DELAI c1, DELAI c2, DELAI c3,
                       f,         ZERO,     ZERO,     ZERO)
a = ALERTE(c3)
```

Partie 2 – Compilation

Dans cette partie, on considère deux langages S (pour simple) et R (pour riche) qui comportent des expressions et des instructions (statements en anglais)[1]. Ces deux langages sont assez simples : il n'y a pas de fonctions, toutes les variables sont globales et il n'y a pas de définition de variable (seulement des assignations). On pourra supposer qu'une variable est toujours initialisée à 0.

Expressions, syntaxe et sémantique

Le deux langages comportent des instructions et des expressions. Les expressions des deux langages sont les mêmes et peuvent être représentées en OCaml par des arbres de syntaxe abstraite (AST) du type suivant :

```
type var_name = string

type expr =
| Const of int
| Add of expr*expr
| LessEq of expr*expr
| Var of var_name
```

[1] Des langages nommés S et R existent vraiment et sont utilisés notamment pour les statistiques mais il n'y a aucun rapport avec les langages étudiés ici.

Toutes les expressions manipulées sont entières, les variables stockent des entiers, la comparaison se fait avec l'ordre classique des entiers. Si on note env la fonction qui associe à chaque variable sa valeur (on rappelle qu'une variable non initialisée vaut 0). On peut définir la sémantique d'une expression de la façon suivante :

$$
\begin{aligned}
\text{sem}(\texttt{Const i}) &= \text{i} \\
\text{sem}(\texttt{Add}(e_1,e_2)) &= \text{sem}(e_1) + \text{sem}(e_2) \\
\text{sem}(\texttt{Var}(v)) &= \text{env}(v) \\
\text{sem}(\texttt{LessEq}(e_1,e_2)) &= \begin{cases} 1 & \text{si sem}(e_1) \leq \text{sem}(e_2) \\ 0 & \text{sinon, c'est à dire quand sem}(e_1) > \text{sem}(e_2) \end{cases}
\end{aligned}
$$

Les entiers ne peuvent pas être arbitrairement grands. Les entiers sont tous à interpréter comme des entiers 32 bits signés en convention complément à deux.

Instructions du langage S

Le langage S est très simple, il ne contient que trois types d'instruction et ses AST peuvent être représentés avec le type suivant :

```
type s_stmt =
  | Sassign of var_name * expr
  | Sbeqz of var_name * int
  | Sprint of var_name
```

Un programme dans le langage S est une liste numérotée d'instructions $i_0 ... i_n$. Une exécution commence par l'instruction 0 et, à chaque étape, on passe de l'instruction k à $k+1$ sauf s'il y a eu un branchement. Dès que l'on tente d'exécuter une instruction i qui n'existe pas ($i < 0$ ou $i > n$), le programme s'arrête. Pour exécuter l'instruction k :

- si $i_k =$ `Sassign(v,e)`, on remplace la valeur de la variable `v` par sem(`e`) ;
- si $i_k =$ `Sprint(v)`, on affiche la valeur de la variable `v` ;
- si $i_k =$ `Sbeqz(v,o)`, le programme saute à l'instruction $k + o$ quand la variable `v` vaut 0 ; d'où le nom beqz pour Branchement si EQal à Zéro. Remarquer que `o` est un décalage (offset en anglais) et non une adresse absolue !

Instructions du langage R

Le langage R est un peu plus complexe et les AST pour représenter ses programmes sont représentés par le type suivant :

```
type r_stmt =
  | Rassign of var_name * expr
  | Rprint of expr
  | Rifelse of expr * r_stmt * r_stmt
  | Rwhile of expr * r_stmt
  | Rblock of r_stmt list
```

Dans le langage R, un programme est une instruction. La sémantique d'un programme dans le langage R est relativement intuitive. Formellement, on peut la définir ainsi :

- `Rassign(v,e)` remplace la valeur de la variable `v` par sem(`e`),

- `Rprint(e)` affiche la valeur sem(`e`),
- `Rifelse(e,s1,s2)` exécute `s1` quand sem(`e`) $\neq 0$ et `s2` sinon,
- `Rwhile(e,s)` répète l'exécution de `s` tant que sem(`e`) $\neq 0$ (`s` peut être exécutée 0 fois ou indéfiniment).
- `Rblock([s1; ...; sn])` exécute `s1`, puis exécute `s2`, etc., jusqu'à `sn`.

Question 5 – Fibonnacci en R Traduisez en langage R le programme suivant. Ce code est écrit dans un pseudo code similaire à Python et calcule le n-ième nombre de Fibonnaci.

```
n # est une variable qui a une certaine valeur
if n <= -1:
    print(-1)
else:
    courant = 1
    precedent = 0
    prochain = 1
    iteration = 0
    while i <= n:
        n = n+1
        prochain = courant + precedent
        precedent = courant
        courant = prochain
    print(courant)
```

La traduction se fait assez facilement :

```
Rifelse(
  LessEq(Var "n",Const (-1)),
  Rprint(Const (-1)),
  Rblock([
       Rassign("courant",Const 1);
       Rassign("precedent",Const 0);
       Rassign("prochain",Const 1);
       Rwhile(
         LessEq(Var "i",Var "n"),
         Rblock([
              Rassign("n",Add(Var "n",Const 1)) ;
              Rassign("prochain",
                    Add(Var "courant",Var "precedent")) ;
              Rassign("precedent", Var "courant") ;
              Rassign("courant", Var "prochain") ;
            ])
        ) ;
       Rprint(Var "courant")
     ])
 )
```

Question 6 – Fibonnaci en S Traduisez le programme de la question précédente en langage S.

Ici ma réponse correspond à la traduction demandée à la question d'après.

```
[|
  (* 0  *) Sassign ("_test",
               LessEq (Var "n", Const (-1))));
  (* 1  *) Sbeqz ("_test", 5));
  (* 2  *) Sassign ("_out", Const (-1)));
  (* 3  *) Sprint "_out");
  (* 4  *) Sassign ("_test", Const 0));
  (* 5  *) Sbeqz ("_test", 14));
  (* 6  *) Sassign ("courant", Const 1));
  (* 7  *) Sassign ("precedent", Const 0));
  (* 8  *) Sassign ("prochain", Const 1));
  (* 9  *) Sassign ("_test",
                   LessEq (Var "i", Var "n")));
  (* 10 *) Sbeqz ("_test", 7));
  (* 11 *) Sassign ("n", Add (Var "n", Const 1)));
  (* 12 *) Sassign ("prochain",
                Add (Var "courant", Var "precedent")));
  (* 13 *) Sassign ("precedent", Var "courant"));
  (* 14 *) Sassign ("courant", Var "prochain"));
  (* 15 *) Sassign ("_test", Const 0));
  (* 16 *) Sbeqz ("_test", -8));
  (* 17 *) Sassign ("_out", Var "courant"));
  (* 18 *) Sprint "_out")
|]
```

Question 7 ★ – Compilation R → S Proposer une méthode pour compiler le langage R vers le langage S. Il n'est pas nécessaire pour cette question de répondre avec du code mais votre réponse doit être précise, c'est à dire : comment gérer les expressions ? comment traduire chacun des types d'instruction ? quelles sont les structures de données nécessaires ? si la traduction est récursive que renvoie la traduction d'une instruction ?

```
let trad s = (* prend un r_stmt et renvoie un s_stmt Array *)
  let rec trad = function
    | Rassign(v, e) ->
       [Sassign(v, e)]
       (* Rien à faire pour les assign *)
    | Rprint(e) ->
    (* pour print il faut introduire une variable
    intermédiaire, on suppose qu'aucune variable
    ne commence par _ *)
      [Sassign("_out", e) ; Sprint("_out") ]
    | Rifelse(e,s1,s2) ->
    (* pour test il faut introduire une variable intermédiaire
       et comptabiliser la taille de chaque sous-statement *)
       let l1 = trad s1
       and l2 = trad s2 in
       [ Sassign("_test", e) ] @
       [ Sbeqz("_test",List.length(l1)+3) ] @ (* fait le test*)
         l1 @
       [ Sassign("_test", Const 0) ] @ (* prépare un saut *)
       [ Sbeqz("_test",List.length(l2)+1) ] @
         l2     (* fais le else *)
    | Rwhile(e,s) ->
       let l = trad s in
       [ Sassign("_test", e) ] @ (* fais le test*)
       [ Sbeqz("_test",List.length(l)+3) ] @
          (* le +3 compte le test et le saut final *)
         l @
         [ Sassign("_test", Const 0) ] @ (* revient au debut*)
         [ Sbeqz("_test",-List.length(l)-4) ]
    | Rblock sl ->
       List.concat_map trad sl
  in
  trad s |> Array.of_list
```

Question 8 ★★– Compilation R → MIPS Proposer une méthode pour compiler le langage R vers l'assembleur MIPS. Comme pour la question précédente, il n'est pas nécessaire de répondre avec du code pour cette question.

Ceci n'est pas une réponse complète mais un élément de réponse :

- on commence par traduire le terme R en terme S
- on stocke les variables globales dans le segment GP
- ensuite on "compile" les expressions comme vu en cours
- enfin on traduit les instruction en rajoutant un label par instruction (`L42` pour l'instruction `42`, etc.), ensuite on traduit `Sprint` en syscall, le `Sassign` en compilant l'expression puis `Sw` et enfin `Sbeqz` en branchement conditionnel `Beqz` avec le label $o + k$ où k est l'instruction courante et o est l'offset du `Sbeqz`.