# Sujet 0 de Composition d'informatique

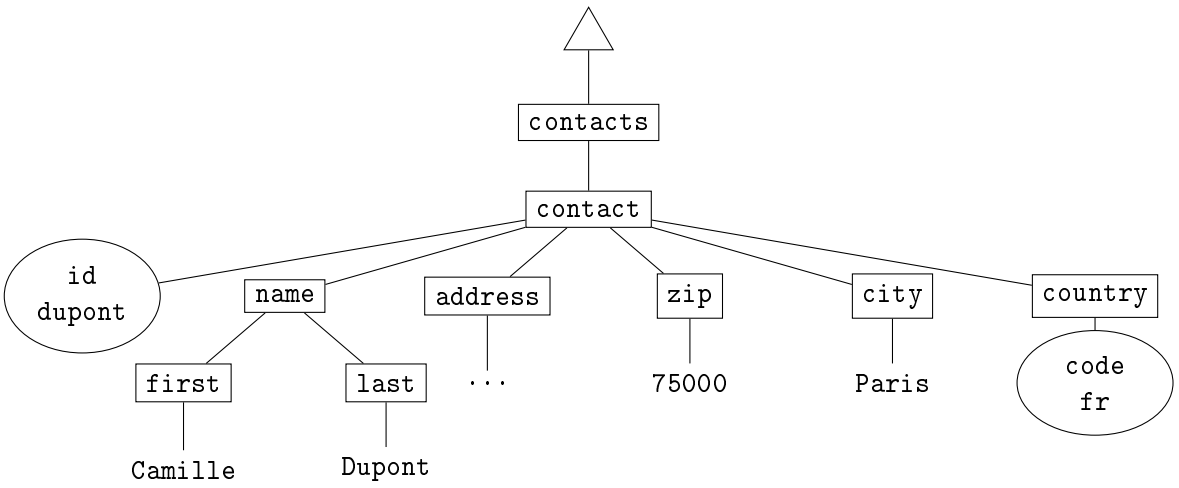
Ce sujet contient trois problèmes indépendants qui doivent être traités tous les trois.

## Problème 1 : Codage de documents XML dans un SGBD relationnel

Un *document XML* est un document mêlant texte et balises (de la forme `<balise>` ou `<balise attribut="valeur">` pour les *balises ouvrantes* et `</balise>` pour les *balises fermantes*), permettant de représenter de l'information semi-structurée. Les balises sont supposées être bien imbriquées (c'est-à-dire qu'à toute balise ouvrante `<balise>` correspond une balise fermante `</balise>` et réciproquement, et une balise ouverte après une autre balise doit être fermée avant celle-ci). Un document XML commence toujours par une balise ouvrante dont la balise fermante correspondante termine le document. Un exemple d'un tel document est le suivant, que l'on notera $d_0$ (les espaces et retours à la ligne entre balises sont juste pour la clarté de l'affichage, ils ne font pas partie du document) :

```
<contacts>
  <contact id="dupont">
    <name>
      <first>Camille</first>
      <last>Dupont</last>
    </name>
    <address>42, rue Turing</address>
    <zip>75000</zip>
    <city>Paris</city>
    <country code="fr"></country>
  </contact>
</contacts>
```

Un tel document XML peut se représenter de manière graphique comme un *arbre XML*, comme suit (le texte de l'adresse a été remplacé par « ... » pour qu'il tienne dans la figure) :




Un arbre XML est ainsi un arbre (fini) étiqueté, et dans lequel chaque nœud a un type :

— un unique nœud de type *document* (représenté par un triangle) est à la racine de l'arbre XML, et ne comporte pas d'étiquette ;

— les nœuds de type *élément* (représentés par des rectangles) correspondent aux balises du document, avec leur nom, et ont pour enfants une représentation sous forme de forêt du contenu entre chaque balise ouvrante et fermante correspondante ;

— les nœuds de type *texte* (sans cadre dans la figure) correspondent aux fragments de texte du document, avec leur valeur ;

— les nœuds de type *attribut* (représentés par des ellipses) correspondent aux attributs pouvant figurer sur les balises ouvrantes, avec leur nom et valeur.

Pour simplifier, dans cet exercice, nous supposerons que cet arbre est *non-ordonné*, c'est-à-dire que l'ordre des enfants de chaque nœud n'est pas spécifié et n'a pas d'importance.

On souhaite utiliser un système de gestion de bases de données (SGBD) relationnel pour stocker et interroger de manière efficace un document XML arbitraire (ne ressemblant pas forcément à $d_0$), représenté sous la forme d'arbres XML.

On propose d'encoder un arbre XML sous la forme de deux tables :

— une table `Noeud` dont les lignes encodent les nœuds de l'arbre XML avec pour attributs `id` (un identifiant unique), `type_noeud` (le type du nœud), `nom` (le nom des éléments et attributs), `valeur` (la valeur des nœuds texte et des attributs) ;

— une table `Parent` dont les lignes encodent le fait qu'un nœud est l'enfant d'un autre nœud avec pour attributs `noeud` (l'identifiant d'un nœud) et `parent` (l'identifiant du parent de ce nœud)

On choisit d'encoder le type des nœuds par un entier : 0 pour *document*, 1 pour *élément*, 2 pour *texte*, 3 pour *attribut*.

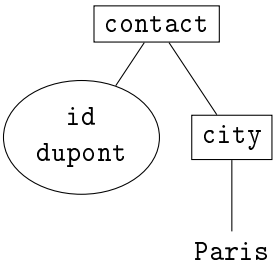
1. Représenter le document $d_0$ sous la forme de son codage dans les deux tables `Noeud` et `Parent`.
2. Dans ce schéma relationnel, quels sont les clés primaires de chaque table ? Qu'existe-t-il comme clés étrangères ?
3. Donner les ordres SQL de création des deux tables `Noeud` et `Parent`, en y faisant figurer les clés primaires et les clés étrangères, et en indiquant les colonnes ne pouvant prendre la valeur `NULL`. On fera l'hypothèse que les noms et valeurs des nœuds font au plus 256 caractères.
4. Écrire en SQL des requêtes retournant, pour n'importe quel document d'entrée :
   a) la liste des valeurs des nœuds texte ;
   b) la liste des noms distincts d'éléments ;
   c) la liste des chaînes de caractère (distinctes) apparaissant soit comme nom soit comme valeur d'attribut ;

d) le nombre d'éléments du document pour chaque nom distinct d'élément.

5. Écrire dans l'algèbre relationnelle une requête renvoyant le nom de l'*élément racine*, c'est-à-dire l'unique élément enfant du nœud document. Exprimer cette même requête en SQL.
6. Proposer deux plans d'exécution de cette requête (comme des arbres dont les nœuds internes sont les opérateurs de l'algèbre relationnelle) et discuter de la performance relative de ces deux plans.

   On souhaite maintenant permettre d'effectuer des requêtes sur un document XML stocké dans notre base. Pour cela, on utilise en général des langages de requêtes pour XML tels que XPath ou XQuery. On va donner une version très simplifiée d'un tel langage, les *requêtes de motif d'arbre*. Pour simplifier également, toutes les requêtes de motif d'arbre que l'on va considérer sont booléennes : elles retournent soit Vrai soit Faux.

   Une requête de motif d'arbre est un arbre dans lequel chaque nœud a un type (élément, texte, attribut), et possiblement un nom et une valeur, comme dans les arbres XML. Contrairement à un arbre XML, une requête de motif d'arbre ne comporte pas de nœud de type « document ». L'arbre suivant est par exemple une telle requête $q_0$ :

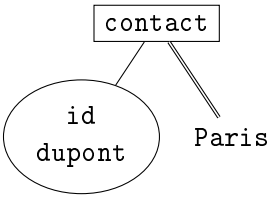



   Une requête de motif d'arbre $q$ s'évalue à Vrai sur un document XML $d$ (ce que l'on note $d \models q$) si et seulement s'il existe un homomorphisme de $q$ vers l'arbre XML $t$ correspondant à $d$ (c'est-à-dire une fonction $h$ qui envoie les nœuds de $q$ vers les nœuds de $t$ en préservant les types, noms, valeurs et la relation parent–enfant). Par exemple, la requête $q_0$ s'évalue à Vrai sur le document exemple $d_0$.

7. Exprimer la requête $q_0$ sous la forme d'une requête SQL $Q_0$ sur le codage relationnel $D$ d'un document $d$, telle que $Q_0$ renvoie un résultat non-vide sur $D$ si et seulement si $d \models q_0$. (On pourra par exemple renvoyer la constante 1 dans le cas où $d \models q_0$, et ne rien renvoyer sinon.)
8. Proposer sous forme de pseudo-code un algorithme permettant de traduire n'importe quelle requête de motif d'arbre $q$ en une requête de l'algèbre relationnelle $Q$ sur le codage relationnel $D$ d'un document $d$, telle que $Q$ renvoie un résultat non-vide sur $D$ si est seulement si $d \models q$.
9. Exhiber une requête de l'algèbre relationnelle sur le codage relationnel d'un document XML qui n'est équivalente à la traduction d'aucune requête de motif d'arbre. Justifier votre affirmation.

   On ajoute aux requêtes de motif d'arbre une fonctionnalité supplémentaire. Une requête de motif d'arbre *étendue* est définie comme une requête de motif d'arbre simple, sauf que les arêtes d'un nœud $n$ de l'arbre à son fils $n'$ peuvent être doubles, on parle alors d'arêtes *descendantes*. Dans ce cas, la condition sur l'homomorphisme $h$ devient que $h(n)$ doit être un *ancêtre strict* de $h(n')$, pas forcément son parent.

   Ainsi, $q_1$ est la requête suivante exprime le fait qu'il existe un élément `contact` dont l'attribut `id` a la valeur `dupont` et qui a un descendant textuel ayant pour valeur `Paris` :




On voit immédiatement que $d_0 \models q_1$.

10. Une conséquence (admise) d'un résultat fondamental de théorie des modèles finis (le *théorème de localité de Gaifman*) est qu'il est maintenant impossible de fournir une traduction d'une requête de motif d'arbre étendue dans l'algèbre relationnelle avec le codage des documents XML précédemment considéré. Il nous faut donc changer notre codage. Proposer un nouveau codage d'un arbre XML sous forme d'une base relationnelle dans lequel il est possible de traduire toute requête de motif d'arbre étendue en une requête de l'algèbre relationnelle équivalente. Proposer sous forme de pseudo-code un algorithme de traduction.

## Problème 2 : Programmation fonctionnelle

Dans ce problème, on utilise le langage OCaml. On se donne le type `tree` suivant pour représenter des arbres binaires :

```
type tree = E | N of tree * tree
```

Le constructeur `E` représente l'arbre vide et le constructeur `N` représente un nœud, avec ses deux sous-arbres. Ici, les nœuds ne contiennent pas d'information, car seule la forme des arbres nous intéresse. La hauteur d'un arbre $t$, notée $h(t)$, est définie par

$$\begin{aligned} h(\texttt{E}) &= 0, \\ h(\texttt{N}(l,r)) &= 1 + \max(h(l), h(r)). \end{aligned}$$

Elle peut être calculée par la fonction `height` suivante, de type `tree -> int` :

```
let rec height t =
  match t with
  | E        -> 0
  | N (l, r) -> 1 + max (height l) (height r)
```

**Question 1.** Indiquer le nombre *total* d'appels à la fonction `height` dans le calcul de `height (N (N (E, N (E, E)), N (E, E)))`.

**Question 2.** On se donne la fonction suivante :

```
let rec left t n =
  if n = 0 then t else left (N (t, E)) (n - 1)
```

Donner son type et expliquer précisément ce que fait cette fonction.

**Question 3.** On cherche maintenant à exécuter les deux lignes de code suivantes :

```
let t = left E 1000000
let h = height t
```

La première ligne est exécutée sans problème mais la seconde provoque l'erreur suivante :

```
Fatal error: exception Stack_overflow
```

Expliquer cette erreur.

**Programmation par continuation.** Pour parvenir à calculer la hauteur en toute circonstance, une solution consiste à adopter un style de programmation dit *par continuation*. Plutôt que de calculer directement la hauteur $h(t)$ d'un arbre $t$, on va calculer $k(h(t))$ pour une fonction $k$ quelconque. La hauteur s'en déduira alors en prenant pour $k$ la fonction identité. La figure 1 contient

```
let rec aux1 t k =
  match t with
  | E ->
      k 0
  | N (l, r) ->
      aux1 l ((*1*) fun hl ->
        aux1 r ((*2*) fun hr ->
          k (1 + max hl hr)))

let height1 t =
  aux1 t ((*3*) fun h -> h)
```

FIGURE 1 – Une autre façon de calculer la hauteur.

un code OCaml qui réalise cette idée. On prendra le temps de bien lire et de bien comprendre ce code, en prêtant notamment attention au parenthésage. On note qu'il y a cinq fonctions en jeu : les deux fonctions `aux1` et `height1` et les trois fonctions anonymes respectivement marquées `(*1*)`, `(*2*)` et `(*3*)`.

**Question 4.** Indiquer quelles sont *toutes* les fonctions successivement appelées pendant le calcul de `height1 (N (E, N (E, E)))`. On indiquera les appels à `aux1` mais aussi les appels aux fonctions marquées `(*1*)`, `(*2*)` et `(*3*)`. Il n'est pas demandé d'indiquer les paramètres passés à ces différents appels, mais seulement la séquence des appels.

**Question 5.** Donner le type de la fonction `aux1`.

**Question 6.** Montrer que la fonction `height1` calcule bien la hauteur.

**Question 7.** On définit la taille d'un arbre $t$, notée $|t|$, comme son nombre de nœuds, c'est-à-dire

$$\begin{aligned} |\texttt{E}| &= 0, \\ |\texttt{N}(l,r)| &= 1 + |l| + |r|. \end{aligned}$$

Montrer que le calcul de `height1 t` est en $O(|\texttt{t}|)$. Indication : Montrer que la complexité de `aux1 t k` est bornée par $\alpha|\texttt{t}| + \beta + |\texttt{k}|$ où $\alpha$ et $\beta$ sont deux constantes que l'on déterminera et où $|\texttt{k}|$ désigne le coût de la fonction `k`. Toute séquence bornée de constructions OCaml atomiques pourra être considérée de coût constant.

**Défonctionnalisation.** On peut modifier le code de la figure 1 pour qu'il n'utilise plus de fonctions anonymes, mais des valeurs d'un type somme OCaml qui représente les différentes fonctions en jeu. On appelle cela la *défonctionnalisation*. La figure 2 contient un squelette de code OCaml qui réalise cette idée. Le type `cont` est le type somme qui représente les trois fonctions anonymes `(*1*)`, `(*2*)` et `(*3*)`. La fonction `apply` permet d'appliquer une continuation `k` de type `cont` à une valeur `v` de type `int`. La fonction `aux2` est l'analogue de la fonction `aux1`, avec `t` de type `tree` et `k` de type `cont`.

```
type cont =
  | K1 of tree * cont
  | K2 of int * cont
  | K3

let rec aux2 t k =
  match t with
  | E        -> apply ...
  | N (l, r) -> aux2 ...

and apply k v =
  match k with
  | K1 (r, k) -> aux2 ...
  | K2 (h, k) -> apply ...
  | K3        -> ...

let height2 t =
  aux2 ...
```

FIGURE 2 – Une troisième façon de calculer la hauteur.

**Question 8.** Compléter le code des fonctions `aux2`, `apply` et `height2`, en donnant les six morceaux de code aux endroits marqués par points de suspension. Le code inséré ne doit contenir *aucun* appel de fonction.

**Question 9.** L'un des intérêts de la défonctionnalisation est son application à des langages qui ne supportent pas les fonctions anonymes. Discuter de la réalisation du programme de la figure 2 dans le langage C. On ne demande pas d'écrire tout le code, mais seulement les types et les profils des fonctions.

**Application au tri fusion.** La figure 3 contient le code OCaml d'un tri fusion, sous la forme d'une fonction `mergesort` qui prend en paramètre une liste `l` et renvoie une nouvelle liste, triée par ordre croissant, contenant les mêmes éléments. Deux fonctions auxiliaires sont utilisées : la fonction `split` découpe une liste en deux listes de même longueur (à un près) et la fonction `merge` fusionne deux listes supposées déjà triées par ordre croissant.

**Question 10.** Tel que le code de la figure 3 est écrit, il est susceptible de provoquer un débordement de pile, tant dans la fonction `split` que dans la fonction `merge`. Réécrire ce code dans un style par continuation (dans le style de la figure 1 et non de la figure 2), en ajoutant une fonction `k` en paramètre des fonctions `split` et `merge`.

* *
*

```
let rec split l =
  match l with
  | []     -> [], []
  | x :: r -> let l1, l2 = split r in l2, x :: l1

let rec merge l1 l2 =
  match l1, l2 with
  | [], l | l, [] ->
      l
  | x1 :: r1, x2 :: r2 ->
      if x1 <= x2 then x1 :: merge r1 l2 else x2 :: merge l1 r2

let rec mergesort l =
  match l with
  | [] | [_] ->
      l
  | _ ->
      let l1, l2 = split l in merge (mergesort l1) (mergesort l2)
```

FIGURE 3 – Tri fusion d'une liste.

# Problème 3 : Architecture des Ordinateurs

**Exercice 1:** Dans cet exercice, les valeurs sont indicées par la base : ainsi, $1001_b$ est une valeur binaire et $235_d$ est décimale.

Les entiers relatifs sont codés sur $n > 0$ bits en complément à 2 : le mot binaire $m = m_{n-1} \dots m_0$ avec $m_i \in \{0, 1\}$ pour $i \in \{0, \dots, n-1\}$ représente l'entier relatif $-m_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} m_i 2^i$.

1. a) On suppose dans cette sous question que les entiers sont codés en complément à 2 sur 4 bits. Complétez le tableau suivant en le recopiant sur votre copie :

| Complément à 2 | Décimal |
|---|---|
| $0101_b$ | . |
| . | $6_d$ |
| $1010_b$ | . |
| . | $-3_d$ |

   b) Pour $n > 0$ fixé, quelles sont les valeurs entières relatives que l'on peut coder sur $n$ bits en complément à 2 ? Justifiez votre réponse.

2. On considère dans cette question l'addition bit à bit de deux mots binaires sur $n$ bits notés $A$ et $B$. On pose $A = a_{n-1} \dots a_0$ et $B = b_{n-1} \dots b_0$ avec $a_i \in \{0, 1\}$ et $b_i \in \{0, 1\}$ pour tout $i \in \{0, \dots, n-1\}$. Le mot du résultat $S = s_{n-1} \dots s_0$ est codé sur $n$ bits, celui de la retenue $C = c_n c_{n-1} \dots c_0$ sur $n+1$ bits.

   Le schéma de l'additionneur à propagation de retenue est représenté dans la figure 1. Chaque composant $ADD_i$ pour $i \in \{0, \dots, n-1\}$ est un additionneur 1 bit qui effectue la somme $a_i + b_i + c_i$ et qui renvoie $s_i$ le résultat et $c_{i+1}$ la retenue. Le schéma d'un composant $ADD_i$ pour $i \in \{0, \dots, n-1\}$ est présenté dans la figure 2.

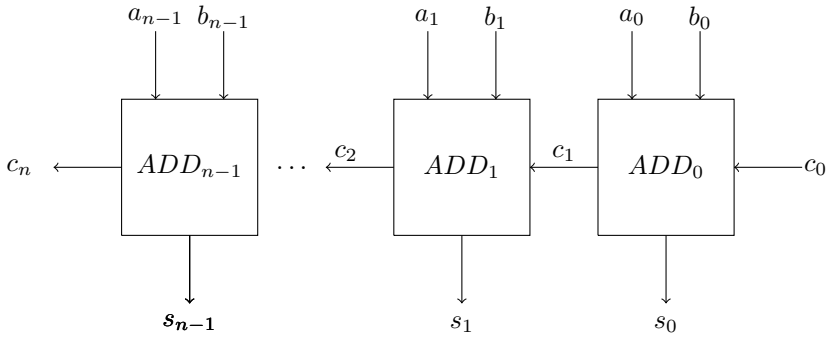



FIGURE 1 – Schéma d'un additionneur à propagation de retenue.

   a) On suppose dans cette sous-question que $n = 4$. Les couples de valeurs $A$ et $B$ considérés sont les suivants :
   — $A = 0110_b$ et $B = 1010_b$ ;
   — $A = 0101_b$ et $B = 0111_b$.
   Pour chacun des couples de valeurs précédents, calculez les mots $S$ et $C$ obtenus par l'addition bit à bit des deux mots $A$ et $B$. On fixe $c_0 = 0$. Quelle est la valeur décimale des mots $S$ obtenus ? Est-ce que cette valeur est toujours égale à la somme des valeurs décimales associées à $A$ et $B$ ?

   b) Donnez la table de vérité de l'additionneur 1 bit $ADD_i$ pour $i \in \{0, \dots, n-1\}$.

   c) Montrez que $s_i = a_i \oplus b_i \oplus c_i$ et $c_{i+1} = a_i b_i + c_i(a_i \oplus b_i)$. L'opérateur $\oplus$ correspond au ou exclusif (XOR) : $a \oplus b = (a + b)(\bar{a} + \bar{b}) = \bar{a}.b + a.\bar{b}$.

   d) Donnez un circuit logique pour $ADD_i$ composé de portes AND, OR et XOR à deux entrées.

3. Le délai d'un circuit combinatoire $\mathcal{S}$ est le temps maximum pour la propagation d'un signal entre une entrée et une sortie de $\mathcal{S}$. On suppose ici qu'il est proportionnel au nombre $\delta(\mathcal{S})$ maximum de cellules logiques AND, OR et XOR d'une entrée à une sortie de $\mathcal{S}$ ; cela correspond donc au nombre maximum de cellules d'un chemin d'une entrée à une sortie.

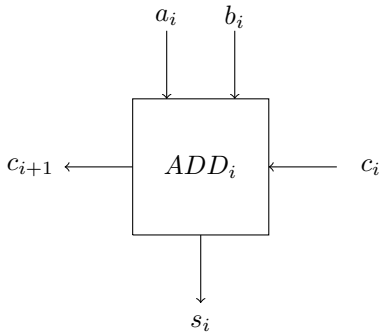



FIGURE 2 – Schéma d'un additionneur 1 bit pour $i \in \{0, \ldots, n-1\}$.

a) Que vaut $\delta(ADD_i)$ pour $i \in \{0, \ldots, n-1\}$, et quels sont les plus longs chemins associés ?

b) Que vaut le délai d'un additionneur à propagation de retenue pour $n = 2$ ?

c) Evaluez l'ordre de grandeur de la valeur $\delta(\mathcal{S})$ où $\mathcal{S}$ est un additionneur à propagation de retenue à $n$ bits avec $n \geq 2$. Justifiez votre réponse.

4. On souhaite par la suite implanter un additionneur dont le délai est plus petit. Soit $i \in \{0, \ldots, n-1\}$.

   a) Que vaut la retenue $c_{i+1}$ si $a_i = b_i$ ?

   b) Même question pour $a_i \neq b_i$ ?

   c) Par la suite, on note $p_i = a_i \oplus b_i$. Que peut on déduire de la valeur de $c_{i+1}$ si $p_i = 1$ ?

5. Pour tout couple $(i, j) \in \{0, \ldots, n-1\}^2$, on note les sous mots binaires $A_{i,j} = a_j \ldots a_i$, $B_{i,j} = b_j \ldots b_i$ et $S_{i,j} = s_j \ldots s_i$. On souhaite définir un additionneur binaire $CSA_{i,j}$ qui calcule $S_{i,j}$ et $c_{j+1}$ en fonction de $A_{i,j}$, $B_{i,j}$ et $c_i$ (voir figure 3). La taille de cet additionneur est définie comme le nombre de bits des opérandes et du résultat, soit $j - i + 1$.
   Cet additionneur permet de sauter le calcul de la retenue en attribuant la valeur $c_i$ à la sortie $c_{j+1}$ dans le cas où $\prod_{k=i}^{j} p_k = 1$. Complétez le schéma de la figure 4 correspondant à l'additionneur $CSA_{0,3}$. Le composant $ADD_{0,3}$ est un additionneur à propagation de retenue sur 4 bits.

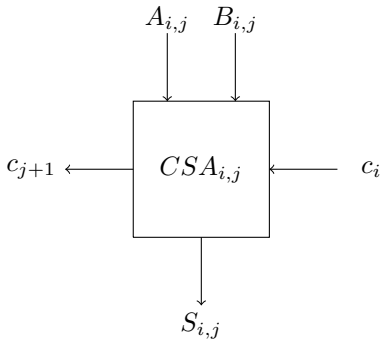



FIGURE 3 – Schéma d'un additionneur $j - i + 1$ bits pour $(i, j) \in \{0, \ldots, n-1\}^2$ avec $i \leq j$.

6. On dispose du circuit $\mathcal{S}'$ d'un additionneur binaire de taille $n$ composé de $m$ blocs $CSA$ connectés en série et de taille $t \geq 2$ comme pour un additionneur à propagation de retenue. Clairement, $n = mt$ et les blocs sont $CSA_{0,t-1}$, $CSA_{t,2t-1}$, … $CSA_{(m-1)t,mt-1}$.
   On cherche à évaluer le délai $\delta(\mathcal{S}')$ de ce circuit. Pour cela, on suppose que le temps nécessaire à une retenue pour traverser $p$ additionneurs 1 bit dans $ADD_{ij}$ est égal à $K_1 p$, où $K_1$ est une constante. D'autre part, on suppose également que le temps nécéssaire pour sauter $p$ blocs $CSA$ est $K_2 p$, où $K_2$ est aussi une constante avec $K_2 < K_1 t$. On dit que la retenue saute le bloc $CSA_{ij}$ si on a attribue la valeur $c_i$ à $c_{j+1}$ sans passer par $ADD_{ij}$.

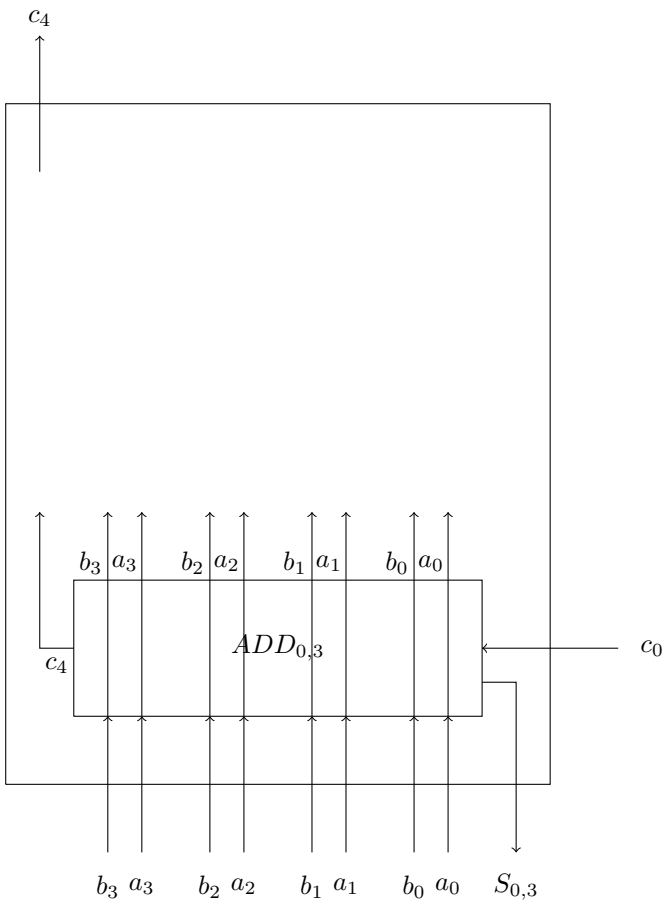



FIGURE 4 – Schéma à compléter de $CSA_{0,3}$.

a) Avec les hypothèses précédentes, quel est le temps maximum pour propager une retenue dans le premier additionneur $ADD_{0,t-1}$ ? Quelles sont les valeurs correspondantes de $a_i$ et $b_i$ pour $i \in \{0, \ldots, t-1\}$ ?

b) Est-ce que une retenue générée lors de la somme $ADD_{0,t-1}$ peut traverser $ADD_{t,2t-1}$ ? En déduire le temps maximum de propagation de la retenue de la sortie de $CSA_{0,2t-1}$ jusqu'à atteindre l'entrée du dernier bloc $CSA_{(m-1)t,mt-1}$. Quelles sont les valeurs correspondantes de $a_i$ et $b_i$ pour $i \in \{t, \ldots (m-1)t-1\}$ ?

c) En déduire le temps maximum pour la propagation d'une retenue. On suppose que $c_0 = 0$. A quelle valeur des opérandes $A$ et $B$ cela correspond ?

d) Montrez que en éliminant $t$, le temps maximum est une fonction de $m$ de la forme $f(m) = \alpha m + \frac{\beta}{m} + \gamma$, où $\alpha$, $\beta$ et $\gamma$ sont des valeurs qui dépendent de $K_1$, $K_2$ et $n$.

7. On souhaite dans cette question determiner les valeurs $m^\star$ et $t^\star$ entières qui minimisent l'évaluation du délai $f(m)$.

a) Quelle est la valeur $\overline{m}$ réelle qui minimise $f$ ?

b) En déduire une définition d'une valeur entière $m^\star$ qui est un diviseur de $n$.

c) Application numérique : on suppose dans cette question que $n = 32$ et que $K_2 = 3K_1$. Calculez $m^\star$ et la valeur $t^\star$ de $t$ correspondante.

d) Dans le cas général, quel est l'ordre de grandeur du délai minimum obtenu pour un additionneur binaire composé de $m^\star$ blocs $CSA$. Quelle conclusion pouvez vous en tirer par rapport à un additionneur à propagation de retenue ?

**Exercice 2:** Dans cet exercice, on considère l'assembleur RISC-V RV32I. Un mémento des instructions est fourni en annexe.

1. a) Donnez l'interprétation du mot `0x00a2a023` si l'on considère qu'il s'agit d'une instruction RV32I. Justifiez votre réponse.

   b) Donnez le codage en hexadécimal de l'instruction `addi x5,x5,-1`. Justifiez votre réponse.

   Les registres sont codés sur 4 bits par leur numéro : par exemple, le registre `x10` a pour codage $1010_b$.

2. On considère le programme assembleur donné par la figure 5.

```
1  .globl main
2  .data
3     tab:    .word 4, -8, 10   # tableau tab de 3 elements
4     taille: .word 3
5     somme:  .word -5
6  .text
7  main:
8     la   x28, somme     # x28 contient l'adresse de somme
9     la   x29, tab
10
11    lw   x6, 0(x28)
12    lw   x7, 0(x29)
13    add  x6, x6, x7
14
15    lw   x7, 4(x29)
16    add  x6, x6, x7
17
18    lw   x7, 8(x29)
19    add  x6, x6, x7
20
21    sw   x6, 0(x28)
```

FIGURE 5 – Programme assembleur associé à la question 2

   a) On suppose que les adresses mémoire sont sur 32 bits et que le segment de données démarre à l'adresse `0x10010000`. Quelles sont les adresses du tableau `tab` et des valeurs `taille` et `somme` ?

   b) Indiquez le contenu des premières lignes du segment data et des registres suivant une fois le programme terminé.

| Registres | x6 | x7 | x28 | x29 |
|---|---|---|---|---|
| Contenu | 0x........ | 0x........ | 0x........ | 0x........ |

| Adresse | mot |
|---|---|
| 0x10010000 | 0x........ |
| 0x10010004 | 0x........ |
| 0x10010008 | 0x........ |
| 0x1001000c | 0x........ |
| 0x10010010 | 0x........ |

   c) Que fait ce programme ?

3. On considère le programme C suivant :

```
int taille = 10;
int somme = 12;
int tab[] = {2, 7, 3, 1, 7, -9, 45, 2, 8, 19};
void main()
    { int i;
      for(i = 0 ; i < taille ; i++)
        somme = somme + tab[i];
    }
```

   Complèter le programme assembleur présenté par la figure 6 correspondant à ce code C.

```
1  .globl main
2  .data
3     taille:     .word 10
4     somme:      .word 12
5     tab:        .word 2, 7, 3, 1, 7, -9, 45, 2, 8, 19
6
7  .text
8  main:
9      la   x28, taille     # x28 contient l'adresse du démarrage de la zone data
10     lw   x29, 0(x28)     # x29 contient la valeur de la taille
11     lw   x30, 4(x28)     # x30 contient la valeur de somme
12     addi x31, x28, 8     # x31 contient l'adresse du premier element de tab
```

FIGURE 6 – Début du programme assembleur associé à la question 3

4. On considère le code C suivant et un code assembleur associé donné en figure 7 :

```
int taille = 10;
int tab[] = {2, 7, 3, 1, 7, -9, 45, 2, 8, 19};

void afficheTab(int* tableau, int n)
    { int i;
      for(i = 0 ; i < n ; i++)
        printf("%d\n", tableau[i]);
    }

void main(){
    afficheTab(tab,taille);
}
```

   a) Quels sont les registres utilisés sur cet exemple pour passer des paramètres lors des appels de fonction et des appels systèmes (`ecall`) ?

   b) Quels sont les registres `s` stockés dans la pile de la ligne 18 à 20 et pourquoi sont ils sauvegardés ? Quand sont ils restaurés ? Qu'est ce que cela garantit ?

   c) Est-ce que les paramètres de la fonction `afficheTab` sont stockés dans la pile ? D'une manière générale, décrire en deux/trois phrases comment s'effectuent les passages de paramètres utilisant les registres `a` et `s`.

   d) Les registres temporaires `t` sont des registres généraux qui ne sont pas sauvegardés. Est-ce que on aurait pu utiliser dans le corps de la fonction des registres temporaires `t` plutôt que des registres `s` ?

   e) Est-ce que le stockage du registre `ra` est nécessaire pour toutes les fonctions en général ? Justifiez votre réponse.

5. Ecrire le code assembleur de la fonction récursive `void afficheRec(int *tableau, int n)` qui affiche les `n` éléments du tableau. Vous veillerez à ne sauvegarder dans la pile que les valeurs nécessaires en utilisant les mêmes conventions que la fonction itérative précédente.

```
1  .globl _start            # adresse de démarrage du programme
2  .data
3      taille:     .word 10
4      tab:        .word 2, 7, 3, 1, 7, -9, 45, 2, 8, 19
5
6  .text
7  _start:
8      la   a0, tab          # a0 contient l'adresse du premier élément du tableau
9      la   t0, taille
10     lw   a1, 0(t0)        # a1 contient la taille du tableau
11     jal  ra, afficheTab   # appel de la fonction afficheTab
12
13     addi a0, x0, 0        # code de retour 0
14     addi a7, x0, 93       # le code de commande 93
15     ecall                 # Appel pour finir
16
17 afficheTab:
18 # prologue de la fonction
19     addi sp, sp, -16       # réservation sur la pile
20     sw   ra, 0(sp)        # sauvegarde de l'adresse de retour
21     sw   s0, 4(sp)        # sauvegarde du registre s0
22     sw   s1, 8(sp)        # sauvegarde du registre s1
23     sw   s2, 12(sp)       # sauvegarde du registre s2
24
25 #corps de la fonction
26
27     mv   s0, a1           # s0 contient la taille du tableau
28     mv   s1, a0           # s1 contient l'adresse du premier element de tab
29
30     or   s2, x0, x0       # i=0 (inialisation) dans s2
31 debut:
32     bge  s2, s0, fin      # branche à fin si x>=taille
33
34     lw   a0, 0(s1)        # a0 contient tab[i]
35     addi a7, x0, 1        # le code de commande 1 correspondant au print
36     ecall                 # appel pour imprimer
37
38
39     addi s2, s2, 1        # incremente i
40     addi s1, s1, 4        # passage à l élément suivant de tab
41     jal  x0, debut        # retour au test de boucle
42
43 fin:
44 # epilogue de la fonction
45     lw   ra, 0(sp)        # récupération de l'adresse de retour
46     lw   s0, 4(sp)        # récupération du registre s0
47     lw   s1, 8(sp)        # récupération du registre s1
48     lw   s2, 12(sp)       # récupération du registre s2
49     addi sp, sp, 16      # restauration de la pile
50     ret                    # retour a l'appelant
```

FIGURE 7 – Programme assembleur associé à la question 4

# Annexes de l'exercice 2

La majorité des documents en annexe sont issus du livre de David Patterson et Andrew Waterman, the RISC-V reader, An open Architecture Atlas.

## A Instructions et pseudo-instructions

La partie gauche de la figure 8 liste l'ensemble des commandes de base (ne considérer que la colonne RV32I Base). Les figures 9 et 10 regroupent l'ensemble des pseudo-instructions considérées et leur instructions équivalentes.

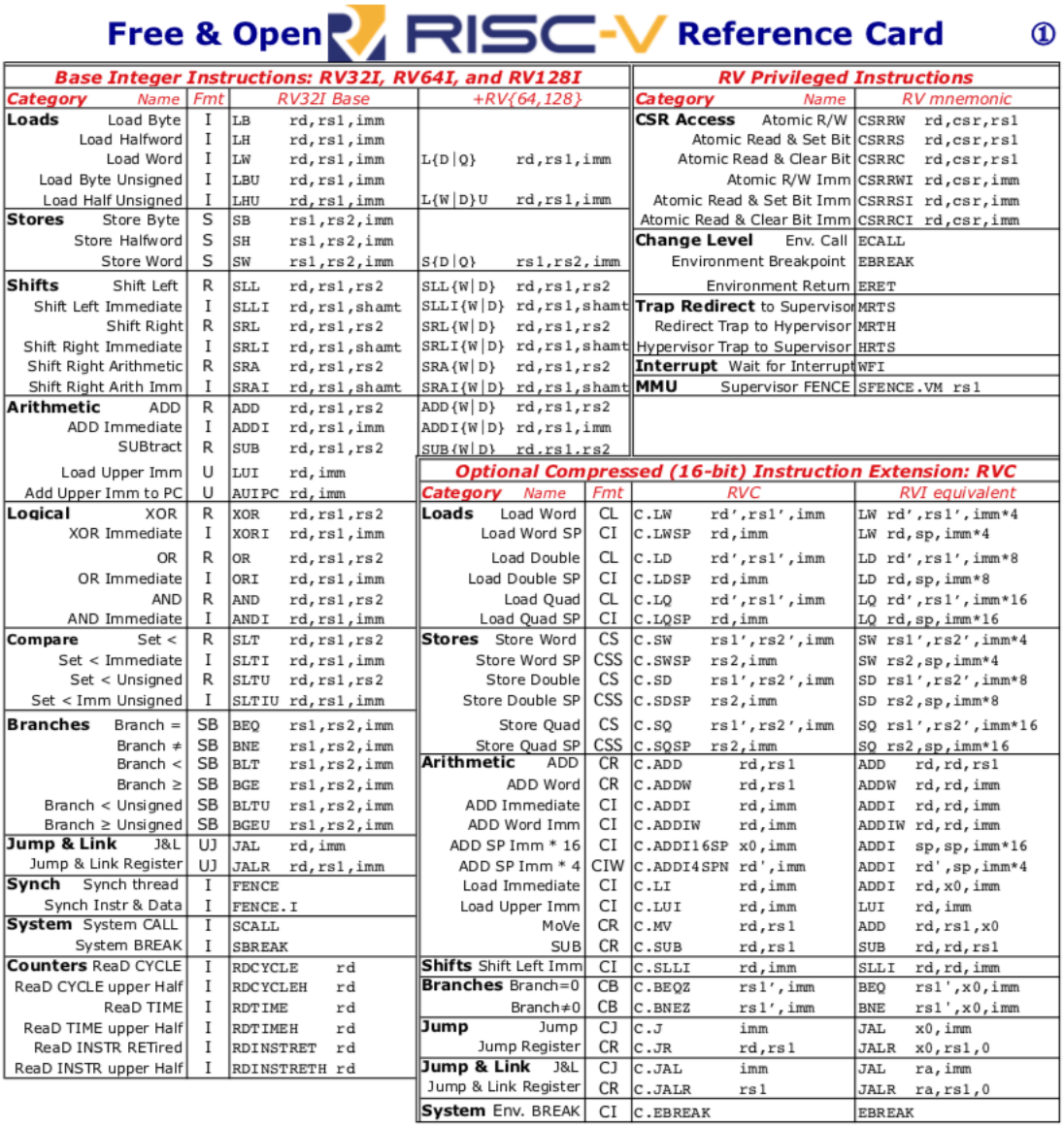
# B Registres

La version de base du RISC-V comporte 32 registres indifférenciés et le PC (Program Counter) listés par la figure 11. Cependant, leur utilisation est fixée par des conventions d'utilisation. De plus, on peut les désigner par leur nom, ou leur numéro : par exemple, le 5ième registre peut être désigné par `x5` ou `t0`.

- Le registre `zero` est bloqué à 0 ;
- Le registre `ra` est utilisé pour stocker l'adresse de retour lors d'un appel de fonction ;
- Le registre `sp` est l'adresse du sommet de la pile ;
- Les registres temporary `t0`,...,`t6` peuvent être utilisés sans contrainte particulière ;
- Les registres `a0`,...,`a7` sont utilisés pour passer les arguments d'une fonction, et les registres `a0` et `a1` permettent également de renvoyer des valeurs lors d'un appel de fonction ;
- Les registres `s0`,...,`s11` ne doivent pas être modifiés par un appel de fonction, ce qui nécessite de les sauvegarder.

# C Codage des instructions

Les instructions sont codées sur 32 bit. La figure 12 résume les 6 types d'instructions et le codage des instructions de base. Les immédiats sont codés 12 ou 20 bits selon les instructions. La notation `imm[X|Y:Z]` désigne les bits de $Y$ à $Z$ d'un immédiat codé sur $X$ bits.



# Free & Open RISC-V Reference Card ①

## Base Integer Instructions: RV32I, RV64I, and RV128I

| Category | Name | Fmt | RV32I Base | +RV{64,128} |
|---|---|---|---|---|
| **Loads** | Load Byte | I | LB rd,rs1,imm | |
| | Load Halfword | I | LH rd,rs1,imm | |
| | Load Word | I | LW rd,rs1,imm | L{D\|Q} rd,rs1,imm |
| | Load Byte Unsigned | I | LBU rd,rs1,imm | |
| | Load Half Unsigned | I | LHU rd,rs1,imm | L{W\|D}U rd,rs1,imm |
| **Stores** | Store Byte | S | SB rs1,rs2,imm | |
| | Store Halfword | S | SH rs1,rs2,imm | |
| | Store Word | S | SW rs1,rs2,imm | S{D\|Q} rs1,rs2,imm |
| **Shifts** | Shift Left | R | SLL rd,rs1,rs2 | SLL{W\|D} rd,rs1,rs2 |
| | Shift Left Immediate | I | SLLI rd,rs1,shamt | SLLI{W\|D} rd,rs1,shamt |
| | Shift Right | R | SRL rd,rs1,rs2 | SRL{W\|D} rd,rs1,rs2 |
| | Shift Right Immediate | I | SRLI rd,rs1,shamt | SRLI{W\|D} rd,rs1,shamt |
| | Shift Right Arithmetic | R | SRA rd,rs1,rs2 | SRA{W\|D} rd,rs1,rs2 |
| | Shift Right Arith Imm | I | SRAI rd,rs1,shamt | SRAI{W\|D} rd,rs1,shamt |
| **Arithmetic** | ADD | R | ADD rd,rs1,rs2 | ADD{W\|D} rd,rs1,rs2 |
| | ADD Immediate | I | ADDI rd,rs1,imm | ADDI{W\|D} rd,rs1,imm |
| | SUBtract | R | SUB rd,rs1,rs2 | SUB{W\|D} rd,rs1,rs2 |
| | Load Upper Imm | U | LUI rd,imm | |
| | Add Upper Imm to PC | U | AUIPC rd,imm | |
| **Logical** | XOR | R | XOR rd,rs1,rs2 | |
| | XOR Immediate | I | XORI rd,rs1,imm | |
| | OR | R | OR rd,rs1,rs2 | |
| | OR Immediate | I | ORI rd,rs1,imm | |
| | AND | R | AND rd,rs1,rs2 | |
| | AND Immediate | I | ANDI rd,rs1,imm | |
| **Compare** | Set < | R | SLT rd,rs1,rs2 | |
| | Set < Immediate | I | SLTI rd,rs1,imm | |
| | Set < Unsigned | R | SLTU rd,rs1,rs2 | |
| | Set < Imm Unsigned | I | SLTIU rd,rs1,imm | |
| **Branches** | Branch = | SB | BEQ rs1,rs2,imm | |
| | Branch ≠ | SB | BNE rs1,rs2,imm | |
| | Branch < | SB | BLT rs1,rs2,imm | |
| | Branch ≥ | SB | BGE rs1,rs2,imm | |
| | Branch < Unsigned | SB | BLTU rs1,rs2,imm | |
| | Branch ≥ Unsigned | SB | BGEU rs1,rs2,imm | |
| **Jump & Link** | J&L | UJ | JAL rd,imm | |
| | Jump & Link Register | UJ | JALR rd,rs1,imm | |
| **Synch** | Synch thread | I | FENCE | |
| | Synch Instr & Data | I | FENCE.I | |
| **System** | System CALL | I | SCALL | |
| | System BREAK | I | SBREAK | |
| **Counters** | ReaD CYCLE | I | RDCYCLE rd | |
| | ReaD CYCLE upper Half | I | RDCYCLEH rd | |
| | ReaD TIME | I | RDTIME rd | |
| | ReaD TIME upper Half | I | RDTIMEH rd | |
| | ReaD INSTR RETired | I | RDINSTRET rd | |
| | ReaD INSTR upper Half | I | RDINSTRETH rd | |

## RV Privileged Instructions

| Category | Name | RV mnemonic |
|---|---|---|
| **CSR Access** | Atomic R/W | CSRRW rd,csr,rs1 |
| | Atomic Read & Set Bit | CSRRS rd,csr,rs1 |
| | Atomic Read & Clear Bit | CSRRC rd,csr,rs1 |
| | Atomic R/W Imm | CSRRWI rd,csr,imm |
| | Atomic Read & Set Bit Imm | CSRRSI rd,csr,imm |
| | Atomic Read & Clear Bit Imm | CSRRCI rd,csr,imm |
| **Change Level** | Env. Call | ECALL |
| | Environment Breakpoint | EBREAK |
| | Environment Return | ERET |
| **Trap Redirect** | to Supervisor | MRTS |
| | Redirect Trap to Hypervisor | MRTH |
| | Hypervisor Trap to Supervisor | HRTS |
| **Interrupt** | Wait for Interrupt | WFI |
| **MMU** | Supervisor FENCE | SFENCE.VM rs1 |

## Optional Compressed (16-bit) Instruction Extension: RVC

| Category | Name | Fmt | RVC | RVI equivalent |
|---|---|---|---|---|
| **Loads** | Load Word | CL | C.LW rd',rs1',imm | LW rd',rs1',imm*4 |
| | Load Word SP | CI | C.LWSP rd,imm | LW rd,sp,imm*4 |
| | Load Double | CL | C.LD rd',rs1',imm | LD rd',rs1',imm*8 |
| | Load Double SP | CI | C.LDSP rd,imm | LD rd,sp,imm*8 |
| | Load Quad | CL | C.LQ rd',rs1',imm | LQ rd',rs1',imm*16 |
| | Load Quad SP | CI | C.LQSP rd,imm | LQ rd,sp,imm*16 |
| **Stores** | Store Word | CS | C.SW rs1',rs2',imm | SW rs1',rs2',imm*4 |
| | Store Word SP | CSS | C.SWSP rs2,imm | SW rs2,sp,imm*4 |
| | Store Double | CS | C.SD rs1',rs2',imm | SD rs1',rs2',imm*8 |
| | Store Double SP | CSS | C.SDSP rs2,imm | SD rs2,sp,imm*8 |
| | Store Quad | CS | C.SQ rs1',rs2',imm | SQ rs1',rs2',imm*16 |
| | Store Quad SP | CSS | C.SQSP rs2,imm | SQ rs2,sp,imm*16 |
| **Arithmetic** | ADD | CR | C.ADD rd,rs1 | ADD rd,rd,rs1 |
| | ADD Word | CR | C.ADDW rd,rs1 | ADDW rd,rd,imm |
| | ADD Immediate | CI | C.ADDI rd,imm | ADDI rd,rd,imm |
| | ADD Word Imm | CI | C.ADDIW rd,imm | ADDIW rd,rd,imm |
| | ADD SP Imm * 16 | CI | C.ADDI16SP x0,imm | ADDI sp,sp,imm*16 |
| | ADD SP Imm * 4 | CIW | C.ADDI4SPN rd',imm | ADDI rd',sp,imm*4 |
| | Load Immediate | CI | C.LI rd,imm | ADDI rd,x0,imm |
| | Load Upper Imm | CI | C.LUI rd,imm | LUI rd,imm |
| | MoVe | CR | C.MV rd,rs1 | ADD rd,rs1,x0 |
| | SUB | CR | C.SUB rd,rs1 | SUB rd,rd,rs1 |
| **Shifts** | Shift Left Imm | CI | C.SLLI rd,imm | SLLI rd,rd,imm |
| **Branches** | Branch=0 | CB | C.BEQZ rs1',imm | BEQ rs1',x0,imm |
| | Branch≠0 | CB | C.BNEZ rs1',imm | BNE rs1',x0,imm |
| **Jump** | Jump | CJ | C.J imm | JAL x0,imm |
| | Jump Register | CR | C.JR rd,rs1 | JALR x0,rs1,0 |
| **Jump & Link** | J&L | CJ | C.JAL imm | JAL ra,imm |
| | Jump & Link Register | CR | C.JALR rs1 | JALR ra,rs1,0 |
| **System** | Env. BREAK | CI | C.EBREAK | EBREAK |

FIGURE 8 – Extrait de la Reference card du RISC-V

| Pseudoinstruction | Base Instruction(s) | Meaning |
|---|---|---|
| `nop` | `addi x0, x0, 0` | No operation |
| `neg rd, rs` | `sub rd, x0, rs` | Two's complement |
| `negw rd, rs` | `subw rd, x0, rs` | Two's complement word |
| `snez rd, rs` | `sltu rd, x0, rs` | Set if $\neq$ zero |
| `sltz rd, rs` | `slt rd, rs, x0` | Set if $<$ zero |
| `sgtz rd, rs` | `slt rd, x0, rs` | Set if $>$ zero |
| `beqz rs, offset` | `beq rs, x0, offset` | Branch if $=$ zero |
| `bnez rs, offset` | `bne rs, x0, offset` | Branch if $\neq$ zero |
| `blez rs, offset` | `bge x0, rs, offset` | Branch if $\leq$ zero |
| `bgez rs, offset` | `bge rs, x0, offset` | Branch if $\geq$ zero |
| `bltz rs, offset` | `blt rs, x0, offset` | Branch if $<$ zero |
| `bgtz rs, offset` | `blt x0, rs, offset` | Branch if $>$ zero |
| `j offset` | `jal x0, offset` | Jump |
| `jr rs` | `jalr x0, rs, 0` | Jump register |
| `ret` | `jalr x0, x1, 0` | Return from subroutine |
| `tail offset` | `auipc x6, offset[31:12]`<br>`jalr x0, x6, offset[11:0]` | Tail call far-away subroutine |
| `rdinstret[h] rd` | `csrrs rd, instret[h], x0` | Read instructions-retired counter |
| `rdcycle[h] rd` | `csrrs rd, cycle[h], x0` | Read cycle counter |
| `rdtime[h] rd` | `csrrs rd, time[h], x0` | Read real-time clock |
| `csrr rd, csr` | `csrrs rd, csr, x0` | Read CSR |
| `csrw csr, rs` | `csrrw x0, csr, rs` | Write CSR |
| `csrs csr, rs` | `csrrs x0, csr, rs` | Set bits in CSR |
| `csrc csr, rs` | `csrrc x0, csr, rs` | Clear bits in CSR |
| `csrwi csr, imm` | `csrrwi x0, csr, imm` | Write CSR, immediate |
| `csrsi csr, imm` | `csrrsi x0, csr, imm` | Set bits in CSR, immediate |
| `csrci csr, imm` | `csrrci x0, csr, imm` | Clear bits in CSR, immediate |
| `frcsr rd` | `csrrs rd, fcsr, x0` | Read FP control/status register |
| `fscsr rs` | `csrrw x0, fcsr, rs` | Write FP control/status register |
| `frrm rd` | `csrrs rd, frm, x0` | Read FP rounding mode |
| `fsrm rs` | `csrrw x0, frm, rs` | Write FP rounding mode |
| `frflags rd` | `csrrs rd, fflags, x0` | Read FP exception flags |
| `fsflags rs` | `csrrw x0, fflags, rs` | Write FP exception flags |

**Figure 3.3: 32 RISC-V pseudoinstructions that rely on x0, the zero register. Appendix A includes includes the RISC-V pseudoinstructions as well as the real instructions. Those that read the 64-bit counters can read by upper 32 bits in RV32I by using the "h" version of the instructions and the lower 32 bits using the plain version. (Tables 20.2 and 20.3 of [Waterman and Asanović 2017] are the basis of this figure.).**

FIGURE 9 – Pseudo-instructions.

| Pseudoinstruction | Base Instruction(s) | Meaning |
|---|---|---|
| `lla rd, symbol` | `auipc rd, symbol[31:12]`<br>`addi rd, rd, symbol[11:0]` | Load local address |
| `la rd, symbol` | *PIC*: `auipc rd, GOT[symbol][31:12]`<br>`l{w\|d} rd, rd, GOT[symbol][11:0]`<br>*Non-PIC*: Same as `lla rd, symbol` | Load address |
| `l{b\|h\|w\|d} rd, symbol` | `auipc rd, symbol[31:12]`<br>`l{b\|h\|w\|d} rd, symbol[11:0](rd)` | Load global |
| `s{b\|h\|w\|d} rd, symbol, rt` | `auipc rt, symbol[31:12]`<br>`s{b\|h\|w\|d} rd, symbol[11:0](rt)` | Store global |
| `fl{w\|d} rd, symbol, rt` | `auipc rt, symbol[31:12]`<br>`fl{w\|d} rd, symbol[11:0](rt)` | Floating-point load global |
| `fs{w\|d} rd, symbol, rt` | `auipc rt, symbol[31:12]`<br>`fs{w\|d} rd, symbol[11:0](rt)` | Floating-point store global |
| `li rd, immediate` | *Myriad sequences* | Load immediate |
| `mv rd, rs` | `addi rd, rs, 0` | Copy register |
| `not rd, rs` | `xori rd, rs, -1` | One's complement |
| `sext.w rd, rs` | `addiw rd, rs, 0` | Sign extend word |
| `seqz rd, rs` | `sltiu rd, rs, 1` | Set if = zero |
| `fmv.s rd, rs` | `fsgnj.s rd, rs, rs` | Copy single-precision register |
| `fabs.s rd, rs` | `fsgnjx.s rd, rs, rs` | Single-precision absolute value |
| `fneg.s rd, rs` | `fsgnjn.s rd, rs, rs` | Single-precision negate |
| `fmv.d rd, rs` | `fsgnj.d rd, rs, rs` | Copy double-precision register |
| `fabs.d rd, rs` | `fsgnjx.d rd, rs, rs` | Double-precision absolute value |
| `fneg.d rd, rs` | `fsgnjn.d rd, rs, rs` | Double-precision negate |
| `bgt rs, rt, offset` | `blt rt, rs, offset` | Branch if $>$ |
| `ble rs, rt, offset` | `bge rt, rs, offset` | Branch if $\leq$ |
| `bgtu rs, rt, offset` | `bltu rt, rs, offset` | Branch if $>$, unsigned |
| `bleu rs, rt, offset` | `bgeu rt, rs, offset` | Branch if $\leq$, unsigned |
| `jal offset` | `jal x1, offset` | Jump and link |
| `jalr rs` | `jalr x1, rs, 0` | Jump and link register |
| `call offset` | `auipc x1, offset[31:12]`<br>`jalr x1, x1, offset[11:0]` | Call far-away subroutine |
| `fence` | `fence iorw, iorw` | Fence on all memory and I/O |
| `fscsr rd, rs` | `csrrw rd, fcsr, rs` | Swap FP control/status register |
| `fsrm rd, rs` | `csrrw rd, frm, rs` | Swap FP rounding mode |
| `fsflags rd, rs` | `csrrw rd, fflags, rs` | Swap FP exception flags |

**Figure 3.4: 28 RISC-V pseudoinstructions that are independent of x0, the zero register. For `la`, GOT stands for Global Offset Table, which holds the runtime address of symbols in dynamically linked libraries. Appendix A includes the RISC-V pseudoinstructions as well as the real instructions. (Tables 20.2 and 20.3 of [Waterman and Asanović 2017] are the basis of this figure.)**

FIGURE 10 – Pseudo-instructions (suite).

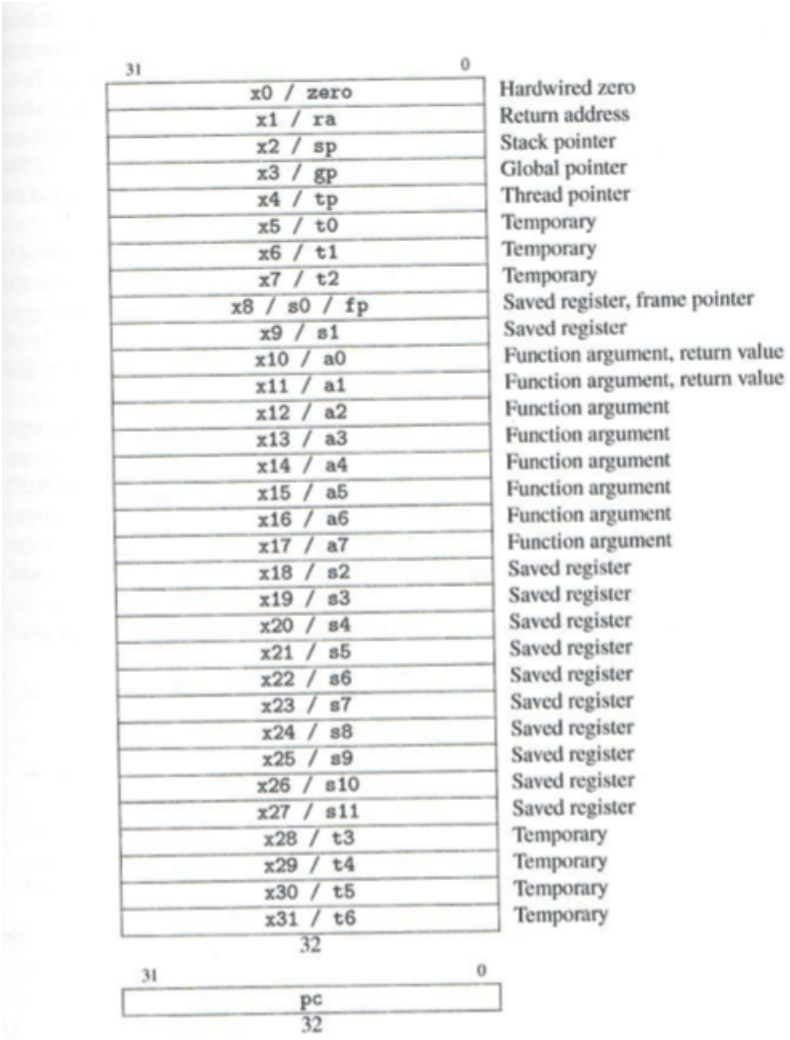



**Figure 2.4: The registers of RV32I. Chapter 3 explains the RISC-V calling convention, the rationale behind the various pointers (sp, gp, tp, fp), Saved registers (s0-s11), and Temporaries (t0-t6). (Figure 2.1 and Table 20.1 of [Waterman and Asanović 2017] is the basis of this figure.)**

FIGURE 11 – Registres et convention d'utilisation.

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |
| imm[12\|10:5] | | rs2 | rs1 | funct3 | imm[4:1\|11] | opcode | B-type |
| imm[31:12] | | | | | rd | opcode | U-type |
| imm[20\|10:1\|11\|19:12] | | | | | rd | opcode | J-type |

**RV32I Base Instruction Set**

| | | | | | |
|---|---|---|---|---|---|
| imm[31:12] | | | | rd | 0110111 | LUI |
| imm[31:12] | | | | rd | 0010111 | AUIPC |
| imm[20\|10:1\|11\|19:12] | | | | rd | 1101111 | JAL |
| imm[11:0] | | rs1 | 000 | rd | 1100111 | JALR |
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | | rs1 | 101 | rd | 0000011 | LHU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |
| fm \| pred \| succ | | rs1 | 000 | rd | 0001111 | FENCE |
| 000000000000 | | 00000 | 000 | 00000 | 1110011 | ECALL |
| 000000000001 | | 00000 | 000 | 00000 | 1110011 | EBREAK |

FIGURE 12 – Convention de codage des instructions RISC-V de base sur 32 bits.