# Assembleur x86 certifié

Roman Delgado

## Table des matières

# 1 Introduction

Un assembleur est un programme permettant de traduire des instructions lisibles par un être humain vers une de leur représentation binaire associée afin d'être éxécutées par le processeur. Contrairement à un compilateur qui effectue lui aussi un travail de traduction, l'assembleur se focalise uniquement sur l'analyse syntaxique et non sur l'aspect sémantique. Il existe de nombreux langages assembleurs pour citer les plus connus `x86`, `arm`. Le programme KVM utilise nottament une fonction de décodage des instructions. Cependant certains problèmes on été relevés au niveau de celle ci. Même si ces problèmes on été résolus on ne peut à l'heure actuelle pas garantir que d'autre bugs ne seront pas découverts dans les années à venir [Amit et al., 2015].

Prenons l'exemple d'un autre logiciel dont l'utilisation est très répandue : `gcc`. C'est l'un des compilateurs du langage C le plus utilisé à l'heure actuelle et pourtant il existe encore des problèmes avec celui- ci. C'est ce qu'a démontré Xavier Leroy en concevant un compilateur certifié pour le langage C `Compcert` [Leroy, 2014]. Après avoir testé les deux compilateurs certains bugs avec `gcc` on été relevés et aucuns avec `compcert`.

Mais comment garantir qu'un programme s'exécute correctement ? En quoi consiste la réalisation d'un logiciel certifié ? Ces questions sont des questions centrales depuis le développement de l'informatique, et de nombreux travaux ont été réalisés sur ce sujet. Les méthodes pour répondre à ces problèmatiques sont par exemple le model checking, les assistants de preuve ect...

Les assistants de preuves permettent de créer des programmes sur lesquels on énonce des propriétés que l'on prouve par la suite avec celui- ci. L'assistant de preuve permet d'assister comme son nom l'indique la rédaction de preuves mais aussi de vérifier automatiquement leur validitée.

La difficulté lors de la réalisation d'un assembleur ne se situe pas en hauteur ; en effet le nombre de fonctions nécéssaires n'est pas très conséquent. Par contre la complexité en largeur de ce type de programme est très grande. Cela est du au nombre d'instructions ainsi qu'à leur syntaxe qui selon la spécification du langage peut être particulièrement riche. Les assistants de preuves se prêtent très bien à ce genre de problème car l'on peut prouver des propriétés sur des ensembles (dans notre cas des instructions) et réaliser une seule preuve pour un ensemble donné. Pour ce projet l'assistant de preuve utilisé sera le langage `Coq` [Bertot Yves, 2004].

Une motivation supplémentaire est la continuation des travaux initiés par Compcert. Le compilateur `Compcert` génère du langage assembleur x86, mais il n'existe pas de programme permettant de garantir sa traduction en bits. C'est donc le but de ce projet qui est la réalisation d'un assembleur x86 certifié.

Afin de mener à bien ce projet et vue la complexité de la spécification de x86, nous allons tout d'abord réaliser un assembleur préliminaire basé sur le langage MMIX.

### 1.1 Contributions

Voici une liste des réalisations apportées pour mener à bien ce projet :

1. Ecole d'hiver de Coq à sophia Antipolis de Yves Bertot.
2. Réalisation de la conversion d'entiers vers une représentation binaire 3.
3. Première représentation des données pour MMIX (seulement 2 instructions).
4. Travail sur la réflexion afin d'implémenter les théorèmes sur la liste d'associations 4.3.
5. Première version des fonctions `encode` et `decode` 5
6. Ajout de l'ensemble des instructions de MMIX dans le programme.
7. Tentative de remaniement de la représentation des données afin de factoriser les preuves 5.4.
8. Lectures sur le sujet des librairies de parser pour l'appliquer à MMIX [Swierstra, 2008].

## 2 MMIX et représentation des données

Le langage d'assembleur MMIX à été crée par Donald Knuth en 1999. A l'heure actuelle il n'existe aucun processeur utilisant ce jeu d'instructions, il à été crée et utilisé dans un but pédagogique.

### 2.1 Définition MMIX

Dans cette section il sera question de définir la syntaxe des instructions MMIX. MMIX est un jeu d'instructions RISK ce qui signifie que la taille des instructions est fixe. Une instruction a une taille de 32 bits, voici une première représentation :

| Etiquette | Operande 1 | Operande 2 | Operande 3 |
|---|---|---|---|
| 0-7 bits | 8-15 bits | 16-23 bits | 24-31 |
| TAG | X | Y | Z |

*Remarque* 1. La dernière ligne du tableau permet simplement d'associer à chaque partie de l'instruction un nom de variable afin d'alléger le texte dans la suite du rapport

Les operandes peuvent être à la fois un immediat ou un registre. Une des spécificité de MMIX est la possibilité d'obtenir des operandes de plus grande taille en en utilisant un nombre réduit :

| TAG | X | YZ |
|---|---|---|
| TAG | XYZ | |

## 2.2 Représentation des données

Dans un premier temps j'ai fait le choix d'utiliser une représentation des données très exhaustive sans chercher à la factoriser. Ce choix est du à mon apprentissage de Coq, il était préférable de ne pas rajouter une difficulté supplémentaire lors de la réalisation des preuves. Un autre choix à été de stocker l'information sur la syntaxe de l'instruction grâce au type des étiquettes. Cela permettra de ne pas mettre cette information dans la liste d'associations, nous y reviendrons dans la section 4.

```
Inductive tag :=
| tag_t_n : tag_ter_normal → tag
| tag_t_i : tag_ter_immediate → tag
| tag_t_i2 : tag_ter_immediate2 → tag
| tag_t_i3 : tag_ter_immediate3 → tag
| tag_t_i4 : tag_ter_immediate4 → tag
| tag_t_i5 : tag_ter_immediate5 → tag
| tag_t_i6 : tag_ter_immediate6 → tag
| tag_d_i : tag_duo_immediate → tag
| tag_d_i2 : tag_duo_immediate2 → tag
| tag_d_i3 : tag_duo_immediate3 → tag
| tag_d_n : tag_duo_normal → tag
| tag_u : tag_uno → tag.
```

Chaque type d'étiquette sera associé à une syntaxe d'instructions par exemple le type `tag_ter_normal` regroupe les tags ne pouvant être utilisés que pour les instructions dont les trois opérandes sont toutes des registres.

```
Inductive tag_ter_normal :=
| ADD : tag_ter_normal
| SUB : tag_ter_normal
```

...

*Remarque* 2. Pour un langage assembleur tel que x86 il ne sera pas possible de définir les instructions de façon exhaustive il faudra utiliser une autre représentation des données.

Maintenant nous voulons définir un type de données qui puisse représenter une instruction. Etant donné que nous avons dissocié les différents types de tags en fonction de la syntaxe de l'instruction, nous allons poursuivre de manière analogue. Voici un exemple pour les instructions dont les 3 opérandes représentent des registres.

```
Record instruction_tern_n :=
  mk_instr_t_n { instr_opcode_t_n : tag_ter_normal;
              instr_operande1_t_n : register ;
              instr_operande2_t_n : register ;
              instr_operande3_t_n : register }.
```

L'utilisation du type enregistrement convient parfaitement pour représenter les instructions car il permet de stocker l'ensemble des informations de celles-ci.

Nous obtenons donc le type suivant pour les instructions :

```
Inductive instruction :=
| instr_t_n : instruction_tern_n → instruction
| instr_t_i : instruction_tern_i → instruction
| instr_t_i2 : instruction_tern_i2 → instruction
| instr_t_i3 : instruction_tern_i3 → instruction
| instr_t_i4 : instruction_tern_i4 → instruction
| instr_t_i5 : instruction_tern_i5 → instruction
| instr_t_i6 : instruction_tern_i6 → instruction
| instr_d_n : instruction_duo_n → instruction
| instr_d_i : instruction_duo_i → instruction
| instr_d_n2 : instruction_duo_i2 → instruction
| instr_d_n3 : instruction_duo_i3 → instruction
| instr_u : instruction_uno → instruction.
```

## 2.3 Egalité entre deux éléments d'un type de donnée

Avant de passer à la section suivante attardons nous sur l'égalité de deux éléments de même type.

*Remarque* 3. Dans cette section nous nous attarderons uniquement sur les types de données inductifs
Si l'on souhaite définir une égalité pour le type de données suivant :

```
Inductive example_type :=
| elem1
| elem2.
```

Il va nous falloir créer une fonction exhaustive énumérant l'ensemble des cas pour déterminer si les deux élements sont égaux ou non. Ce qui nous donne pour l'exemple précédent la fonction suivante :

```
Fixpoint equal (e1 e2 : example_type) : bool :=
  match e1 with
  | elem1 => match e2 with
             | elem1 => true
             | elem2 => false
             end
  | elem2 => match e2 with
             | elem1 => false
             | elem2 => false
             end
  end.
```

Ces fonctions sont très fastidieuses à rédiger et l'on constate déja la taille de celle ci pour un type de données composé de deux éléments. Si nous souhaitions écrire cette fonction le type `tag` dans la section 2.2 la perte de temps serait considérable. Dans Coq il existe une commande qui va nous permettre de générer cette fonction automatiquement, la voici :

```
Scheme Equality for example_type.
```

### 2.4 Lemmes

Définisons notre premier lemme :

```
Lemma tag_beq_different : forall (t1 t2 : tag),
    tag_beq t1 t2 = true → t1 = t2.
```

Ici on désire montrer que si l'on prossède une preuve que deux étiquettes sont équivalentes à l'aide de notre fonction booléene alors on peut déduire une preuve de cette égalité dans le monde propositionnel.

## 3 Conversion N - bits

Cette section s'attarde sur la représentation que nous donnerons à nos instructions sous forme binaire. Il sera aussi également question de conversion des entiers vers cette représentation étant donné que les immédiats ainsi que les registres sont stockés sous forme d'entier pour les instructions.

### 3.1 Représentation des bits

Il nous faut déterminer un type de données pour représenter les instructions sous forme binaires. Notre type de données doit pouvoir nous permettre de :

1. Ne pas avoir de restriction au niveau de la taille car lors du parsing d'un fichier binaire, on souhaite générer un flux de bits puis le découper au fur et à mesure du décodage.
2. Pouvoir encoder un entier sur un nombre de bits quelconque.
3. Il doit être simple car il est préférable que les preuves qui sont le coeur de ce projet restent plus aisées à mettre en place.

Il existe dans la librairie standard de Coq une représentation binaire pour les entiers naturels. Le problème de celle- ci est que l'on ne peut pas représenter un entier naturel sur un nombre de bits différents que celui nécessaire à son encodage. Nous ne pouvons pas nous permettre d'avoir une taille variable en fonction de la valeur encodée et cela contredirait le second pré-requis.

La structure de données par excellence qui satisfait l'ensemble de nos pré-requis est la liste de booléens. Un booléens représentant un bit et la liste le flux.

### 3.2 Fonctions de conversion

Maintenant que nous avons fixé une représentation pour nos suites de bits, commençons par définir une fonction de conversion d'une liste de booléens vers un entier naturel :

```
Fixpoint bit_n (l : list bool) : nat :=
  match l with
    | [] => 0
```

```
    | a :: tl => 2 × bit_n tl + Nat.b2n a
  end.
```

Commençons par prêter attention à cette fonction car elle est une simple implémentation d'une formule permettant de déterminer la valeur d'un nombre sous forme binaire en forme décimale. Passons maintenant à la fonction effectuant l'opération inverse :

```
Fixpoint n_bit (n : nat) (k : nat) : option (list bool) :=
    match n with
      | 0 => match k with
             | 0 => Some []
             | S _ => None
             end
      | S n' => match n_bit n' (Nat.div2 k) with
                  | None => None
                  | Some l => Some (Nat.odd k :: l)
                end
    end.
```

Ici cette fonction possède plusieurs subtilités d'implémentation dues à une restriction du langage Coq.

La première chose que l'on peut remarquer se situe au niveau des arguments de la fonction. En effet nous souhaitons simplement encoder un entier naturel vers une liste de booléens et pourtant nous avons deux arguments. En réalité nous souhaiterions pouvoir écrire notre fonction comme ceci (implémentation en Ocaml) :

```
let rec n_bit k =
  match k with
  | 0 → []
  | n → let l = n_bit (k / 2) in
        (n mod 2) :: l
```

Cependant cette implémentation est impossible avec Coq car celui- ci utilise des opérateurs de points fixes et dans ce dernier exemple l'appel récursif ne s'éffectue pas avec le prédécesseur de `k` mais avec le résultat de `k` par deux. Lors des preuves si l'on fait des récursions sans faire appel au prédécesseur nous ne pourrons pas faire de raisonement par induction.

*Remarque* 4. Dans notre cas cette fonction nous convient parfaitement car nous souhaitons obtenir une liste de booléens de taille déterminée. Cependant cela nous permet d'illustrer la façon dont Coq définit la récursion

On pourra aussi noter le type de retour de la fonction. On utilise une monade bien connue des programmeurs fonctionnels au travers du type `option`. Cela nous permet de dire que la fonction peut échouer sans pour autant devoir renvoyer une valeur par défaut ou lever une exception (en d'autres termes que la fonction est partielle). Ici le cas ou la fonction peut échouer est lorsque l'entier naturel que l'on souhaite encoder nécessite plus de bits que l'entier n alors on ne peut pas réaliser la transformation.

### 3.3 Lemmes

Il nous faut maintenant nous demander quelles sont les propriétés qui nous interressent sur les deux fonctions que nous venons de définir dans la section 3.2.

Une propriété qui s'avèrera surement utile par la suite est le Lemme suivant :

```
Theorem size_n_bit : forall (n k: nat) (l : list bool),
    n_bit n k = Some l → length l = n.
```

Ce lemme permet de vérifier que la taille d'une instruction produite par la fonction `n_bit` est bien de la taille de l'argument n.

Une des informations importante à retenir de ce théorème est sa structure. En effet comme nous avons vu précédement 3.2 la fonction `n_bit` peut échouer, il nous faut donc vérifier cette propriété uniquement lorsque celle- ci termine correctement. C'est pour cette raison que nous devons utiliser l'implication pour ne considérer que ces cas précis.

Il est maintenant question de définir les deux théorèmes les plus importants de cette section. Notre but pour l'ensemble des fonctions que l'on définit est de montrer qu'elles sont inverses. car elles seront utilisées dans l'encodage et le décodage des instructions 5 Voici donc le premier théorème que nous voulons prouver :

```
Lemma n_bit_n : forall (l : list bool) (n k : nat),
    n_bit n k = Some l → bit_n l = k.
```

Comme pour le lemme précédent ce théorème suit la même structure avec une implication en raison des effets de bords. Pour le théorème suivant nous avons besoin d'émettre une hypothèse supplémentaire sur l'argument `n`. Si nous n'effectuons pas cette assertion supplémentaire le théorème n'a pas de sens et n'est par conséquent pas prouvable.

```
Theorem bit_n_bit : forall (l : list bool) (n : nat),
    n = length l → (n_bit n (bit_n l)) = Some l.
```

# 4 Conversion Etiquette - N

Une instruction possède une étiquette ainsi que des opérandes (2). Pour réussir à encoder ou décoder une instruction il nous faut donc d'abord réussir à convertir une étiquette en un entier $\mathbb{N}$. Pourquoi cette conversion s'effectue-t-elle vers l'ensemble des entiers naturels et non vers les listes de $\mathbb{B}$ ? Tout simplement pour éviter de les rédiger statiquement. Nous utiliserons les fonctions définies dans la section précédente 3 lorsque nous souhaiterons transformer nos entiers naturels.

## 4.1 Structure de donnée

Dans la spécification du langage MMIX (section 2) une étiquette est associée à un unique opcode. Nous souhaitons donc utiliser un type de données permettant de réaliser l'association de ces deux éléments, ce qui nous conduit à utiliser le produit cartésien suivant :

```
Definition assoc : Set :=
  tag × nat.
```

Etant donné que le nombre d'instructions est de 256, l'utilisation des listes est envisageable. Un argument supplémentaire est que la liste est un type inductif qui nous permettra de raisonner plus aisément lors des preuves étant donné sa relative simplicité. En effet on pourrait utiliser un dictionnaire pour réduire la complexité, cependant l'utilisation de ce type de données rendrait les preuves plus compliquées. Nous obtenons donc le type suivant pour représenter notre collection d'associations :

```
Definition tag_opcode_assoc :=
  list assoc.
```

*Remarque* 5. L'utilisation des listes à l'instar des dictionnaires permet également de n'utiliser qu'une seule liste d'associations qui permettra d'encoder et de décoder contrairement à ce dernier.

*Remarque* 6. J'ai fait le choix de ne pas me reposer sur la liste d'associations pour stocker des informations sur le décodage des instructions. En effet lors du décodage avec une telle liste d'associations nous n'aurons pas d'information sur la syntaxe de celle- ci (type des opérandes et nombre d'arguments). Cependant cette information sera obtenue ici grâce au type de l'étiquette qui nous indique la syntaxe. Ce choix sera critiqué dans la section 5

## 4.2 Fonctions de recherche

Il est désormais question de réaliser deux fonctions permettant la recherche au sein d'une liste d'associations telles que celle définies dans la section précédente. 4.1.

```
Fixpoint lookup (t : tag) (l : tag_opcode_assoc) : option nat :=
  match l with
    | [] => None
    | (t',n) :: tl => if tag_beq t t'
                      then Some n
                      else lookup t tl
  end.
```

Encore une fois l'utilisation de la monade option est indispensable ici car on ne peut pas garantir que pour une liste donnée nous trouverons l'entier qui lui est associé. Notre travail sera de démontrer que cette fonction n'échoue pas sur une liste contenant l'ensemble des associations. Nous y reviendrons dans

la section 4.3 Voici la fonction effectuant la recherche dans le sens inverse en suivant le même principe :

```
Fixpoint lookdown (n : nat) (l : tag_opcode_assoc) : option tag :=
  match l with
    | [] => None
    | (t,n') :: tl => if  eqb n n'
                      then Some t
                      else lookdown n tl
  end.
```

## 4.3 Lemmes

Avant de commencer toute preuve il nous faut définir de manière statique une liste contenant l'ensemble des associations entre les `tag` et leur entier naturel associé.

```
Definition encdec : tag_opcode_assoc :=
[(tag_t_n(ADD),0);
(tag_t_n(SUB),1);
(tag_t_n(MUL),2);
(tag_t_n(DIV),3);

.....
```

Tous les théorèmes que nous définirons se baserons sur la liste `encdec`. Nous avons besoin de vérifier que la liste contient bien l'ensemble des associations, il existe deux manières de faire. Une première façon qui consiste à faire une analyse de cas au sein de la preuve sur le type `tag`. Une autre méthode consiste à vérifier de façon calculatoire cette propriété, puis de montrer une équivalence entre ce calcul et sa proposition associée. Cette opération en Coq peut être réalisée avec l'aide du type `reflect` :

```
Inductive reflect (P : Prop) : bool → Set :=
| ReflectT : P → reflect P true
| ReflectF : ~ P → reflect P false.
```

Pour mieux comprendre en quoi ce type va nous être utile, il faut comprendre son utilisation. Le type `reflect` est composé de deux constructeurs, si l'on possède un élément de type `reflect P true` alors cela signifie que notre élément est le constructeur `ReflectT` et celui- ci possède une preuve du prédicat `P`. Le raisonnement est le même pour `reflect P false` sauf que dans ce cas on pourra obtenir une preuve de non `P`. Pour vulgariser ce principe on peut dire qu' avoir une preuve de `reflect P b` implique que la preuve de P est équivalente à `b = true`. Ce principe est illustré par le théorème suivant :

Mettons de côté le type `reflect` et définissons maintenant une fonction pour vérifier calculatoirement un prédicat sur l'ensemble des tags :

```
Definition forall_tag (p : tag → bool): bool :=
  (forall_tag_ter_n (fun x => p (tag_t_n x))) &&
  (forall_tag_ter_i (fun x => p (tag_t_i x))) &&
  (forall_tag_ter_i2 (fun x => p (tag_t_i2 x))) &&
  (forall_tag_ter_i3 (fun x => p (tag_t_i3 x))) &&
  (forall_tag_ter_i4 (fun x => p (tag_t_i4 x))) &&
  (forall_tag_ter_i5 (fun x => p (tag_t_i5 x))) &&
  (forall_tag_ter_i6 (fun x => p (tag_t_i6 x))) &&
  (forall_tag_duo_n (fun x => p (tag_d_n x))) &&
  (forall_tag_duo_i (fun x => p (tag_d_i x))) &&
  (forall_tag_duo_i2 (fun x => p (tag_d_i2 x))) &&
  (forall_tag_duo_i3 (fun x => p (tag_d_i3 x))) &&
  (forall_tag_uno (fun x => p (tag_u x))).
```

L'ensemble des fonctions `forall_tag_...` suivent le même schéma que la fonction suivante :

```
Definition forall_tag_uno (p : tag_uno → bool): bool :=
  (p JMP) &&
  (p SAVE) &&
  (p UNSAVE) &&
  (p RESUME) &&
  (p SYNC).
```

Tout comme nous l'avions fait lorsque nous avions défini la fonction d'égalité sur les tags 2.4, nous souhaitons pouvoir passer du monde calculatoire au monde propositionnel :

```
Lemma helpBefore1 : forall (f : tag → bool), forall_tag f = true →
(forall (t: tag), f t = true).

Lemma helpBefore2 : forall (f : tag → bool), (forall (t: tag), f t = true) →
forall_tag f = true.
```

Afin de définir des prédicats pour la fonction `forall_tag` il nous faut définir un équivalent au connecteur propositionnel → :

```
Definition imply (a b : bool): bool := if a then b else true.
```

L'obectif final de cette section est de démontrer les théorèmes suivants (dont la structure est analogue aux lemmes prouvés en dans la section précédente 3.3) :

```
Theorem lookup_lookdown : forall (n : nat) (t : tag) ,
    lookup t encdec = Some n → lookdown n encdec = Some t.

Theorem lookdown_lookup : forall (n : nat) (t : tag),
    lookdown n encdec = Some t → lookup t encdec = Some n.
```

Si nous souhaitons utiliser la réflection il nous faut définir des fonctions reflétant ces théorèmes dans l'ensemble des booléens. La fonction pour le théorème `lookup_lookdown` est la suivante :

```
Definition lookup_encdec : bool :=
  forall_bounded 226 (fun n =>
              forall_tag (fun t =>
                 imply (eq_mnat (lookup t encdec) (Some n))
                       (eq_mtag (lookdown n encdec) (Some t)))).
```

`eq_mtag` et `eq_mtag` sont simplement des fonctions d'égalité sur les types `option tag` et `option nat`. L'intérêt d'avoir utilisé reflect pour réaliser nos preuves peut se voir immédiatement quant à la taille du script de preuves du théorème `lookup_lookdown` que voici :

```
Theorem lookup_lookdown : forall (n : nat) (t : tag) ,
    lookup t encdec = Some n → lookdown n encdec = Some t.

Proof.
  pose proof lookup_encdecP.
  assert (lookup_encdec = true) by apply lookup_encdec_true.
  rewrite H0 in H.
  inversion H.
  intros n.
  specialize (Nat.le_gt_cases n 226).
  intros.
  destruct H2.
  -apply H1.
   exact H2.
   exact H3.
  - pose proof lookup_val t n H3.
    now apply lt_not_le in H4.
Qed.
```

Certains lemmes ont ici été passés volontairement sous silence car l'objectif n'est pas de comprendre reflect dans les moindres détails mais simplement de comprendre son intérêt.

# 5 Encode et decode

Nous arrivons à la dernière partie du programme, l'enjeu de cette section sera de combiner l'ensemble des fonctions déjà définies pour obtenir un assembleur fonctionnel.

## 5.1 Fonctions prélminiaires

Afin de réaliser les fonctions d'encodage et de décodage de nos instructions il nous manque encore quelques opérations. Les premières fonctions que nous allons définir sont deux fonctions permettant de transformer une opérande en liste de booleens.

```
Definition operand_to_bin (o : operande) : option (list bool) :=
  match o with
```

```
    | imm_o (imm k) => n_bit 8 k
    | reg_o (reg k) => n_bit 8 k
  end.
```

Cette fonction encode des operandes sur 8 bits, il nous faut une fonction analogue pour les opérandes sur 16 bits.

```
Definition operand_to_bin_double (o : operande) : option (list bool) :=
```

Définisson quelques lemmes sur ces nouvelles fonctions.

```
Lemma operand_to_bin_hypothesis_reg : forall (l : list bool) (r : register),
    operand_to_bin (reg_o r) = Some l → reg (bit_n l) = r.

Lemma operand_to_bin_hypothesis_imm : forall (l : list bool) (i : imediate),
    operand_to_bin (imm_o i) = Some l → imm (bit_n l) = i.

Lemma operand_to_bin_size : forall (o : operande) (l : list bool),
    operand_to_bin o = Some l → length l = 8.
```

Pour la fonction de décodage nous aurons besoin d'une fonction permettant de découper une liste de booléens en plusieurs sous listes. Voici la fonction qui effectura cette operation :

```
Fixpoint get_first_n_bit  (bi : list bool) (size : nat) : (list bool×list bool) :=
  match size with
  | 0 => ([],bi)
  | S n => match bi with
           | h :: tl => match get_first_n_bit tl n with
                        | (l1,l2) => (h :: l1,l2)
                        end
           | [] => ([],[])
           end
  end.
```

La fonction retourne un couple de deux listes, la première correspond aux `size` premiers bits de la liste et la seconde au reste de la liste. Le fait de retourner le reste de la liste nous permettra de découper celle- ci à nouveau.

## 5.2 Monade bind

La fonction encode utilise les fonctions que nous avons définies dans les sections précédentes qui sont généralement des fonctions partielles. Lorsque l'on souhaite utiliser leur valeur de retour nous sommes obligés de faire un filtrage de la forme :

Cette construction est très lourde dans la définition des fonctions nous allons donc créer une monade afin de faciliter l'utilisation de ces fonctions partielles :

```
Definition M A := option A.

Definition ret {A} (a : A) : M A := Some a.
```

```
Definition bind {A B} (ma : M A)(k : A → M B): M B :=
  match ma with
  | Some a => k a
  | None => None
  end.
```

Nous avons maintenant une fonction bind qui nous permet d'automatiser ce filtrage. Coq nous permet de définir de nouvelles notations comme celle ci :

```
Notation "'let!' x ':=' ma 'in' k" := (bind ma (fun x => k)) (at level 30).
```

## 5.3 Implémentation des fonctions Encode et Decode

Commençons par la définition de la fonction encode. D'après le type instruction 2 chaque instruction s'encode différement, nous allons devoir définir une fonction comme celle- ci pour chaque type d'instruction :

```
Definition encode_t_n (i : instruction_tern_n) : option binary_instruction :=
  let! k := lookup (tag_t_n (i.(instr_opcode_t_n))) encdec in
  let! code := n_bit 8 k in
  let! o1 := operand_to_bin (reg_o i.(instr_operande1_t_n)) in
  let! o2 := operand_to_bin (reg_o i.(instr_operande2_t_n)) in
  let! o3 := operand_to_bin (reg_o i.(instr_operande3_t_n)) in
  ret (code ++ o1 ++ o2 ++ o3).
```

Ce qui nous permet de définir la fonction encode suivante :

```
Definition encode (i : instruction) : option binary_instruction :=
  match i with
  | instr_t_n t => encode_t_n t
  | instr_t_i t => encode_t_i t
  | instr_t_i2 t => encode_t_i2 t
  | instr_t_i3 t => encode_t_i3 t
  | instr_t_i4 t => encode_t_i4 t
  | instr_t_i5 t => encode_t_i5 t
  | instr_t_i6 t => encode_t_i6 t
  | instr_d_n t => encode_d_n t
  | instr_d_i t => encode_d_i t
  | instr_d_n2 t => encode_d_i2 t
  | instr_d_n3 t => encode_d_i3 t
  | instr_u t => encode_u t
  end.
```

La fonction de décodage quant à elle n'est pas composée de plusieurs fonctions car l'entrée étant une liste de booléens. Voici une version partielle du code de décode :

```
Definition decode (bi : binary_instruction) : option instruction :=
  if length bi =? 32
  then
  match get_first_n_bit bi 8 with
```

```
  | (li,next) => let! t := lookdown (bit_n li) encdec in
                 match t with
                 | tag_u u =>
                   match get_first_n_bit next 24 with
                   | (op,[]) =>
                     ret (instr_u (mk_instr_uno u (imm (bit_n op))))
                   | _ => None
                   end
```

Ici la définition que nous avons utilisée pour les tags s'avère utile car elle nous permet une fois récupérée dans la liste d'associations de déterminer immédiatement quel type d'instruction il nous faut créer.

## 5.4 Lemmes

De même que pour la définition d'encode nous allons diviser le travail pour chaque type d'instructions.

```
Lemma encode_decode_t_n : forall (i : instruction_tern_n)
                                 (bi : binary_instruction),
    encode_t_n i = Some bi → decode bi = Some (instr_t_n i).
```

Le problème de la représentation actuelle des données fait que nous sommes obligés de réaliser une preuve similaire mais non triviale pour chaque type d'instruction. Cela crée énormément de redondance dans les preuves et cela n'est jamais bon signe. Nous verrons dans la conclusion 6 les solutions envisageables afin de surmonter ce problème. Cela nous permet finalement d'obtenir la preuve qu'une instruction encoder par notre assembleur puis decoder par celui ci est identique. On a donc une preuve que la syntaxe est belle et bien préservée.

Définissons le théorème opposé à celui- ci à savoir :

```
Lemma decode_encode : forall (bi : binary_instruction)
                             (i : instruction),
    decode bi = Some i → encode i = Some bi.
```

Tout comme pour la preuve d'encode ici nous devons effectuer la preuve pour chaque type d'instructions ce qui rend une nouvelle fois la preuve très verbeuse et redondante.

## 5.5 Theorèmes finaux

A l'heure actuelle nous avons une preuve que notre assembleur suit bien la specification pour une seule instruction. Cependant un assembleur complet travaille sur un ensemble d'instructions ou bien un flux de bits. Dans notre cas nous allons définir une fonction décodant une liste d'instructions et qui retournera une liste de booléens de taille `32 ×length(liste_instructions)`. Pour ce faire il nous faut une fonction préalable :

```
Fixpoint concat_listes_32 (l : list (list bool)) : option (list bool) :=
  match l with
```

```
  | [] => Some []
  | h :: tl => let! res := concat_listes_32 tl in
               if length h =? 32 then Some (h ++ res)
               else None
end.
```

A noter le test sur la taille de la liste qui nous permettra d'obtenir une hypothèse supplémentaire lors de nos preuves. Etant donné que la fonction encode utilise la monade option nous sommes obligés de définir une fonction traverse permettant de transformer une liste de type `list (option A)` en une liste de type `list A`.

```
Fixpoint traverse {A} (l : list (option A)) : option (list A) :=
  match l with
  | [] => Some []
  | (Some e) :: tl => match traverse tl with
                      | Some l => Some (e :: l)
                      | None => None
                      end
  | None :: _ => None
  end.
```

Voici la définition de la fonction encodant une liste d'insructions (pour la lisibilité celle- ci est divisée en 3 fonctions)

```
Definition encode_flux_opt
   (li : list instruction) : list (option binary_instruction) :=
  map encode li.
Definition encode_flux
   (li : list instruction) : option (list binary_instruction) :=
  traverse (encode_flux_opt li).

Definition encode_flux_b (li : list instruction) : option (list bool) :=
  match encode_flux li with
  | None => None
  | Some res => concat_listes_32 res
  end.
```

Passons maintenant au décodage d'un flux de booléens. Tout comme pour l'encodage il nous faut une fonction préliminaire. Cette fois ci nous avons besoin de découper la liste de booléens en entrée en sous listes de taille 32, ce qui nous amène à la définition suivante :

```
Fixpoint cut32_n (n : nat) (l : list bool) : (list (list bool)) :=
  match n with
  | 0 => []
  | S n' => (firstn 32 l) :: (cut32_n n' (skipn 32 l))
  end.

Definition cut32 (l : list bool) : option (list (list bool)) :=
  let n := length l in
  if n mod 32 =? 0 then Some (cut32_n (n / 32) l) else None.
```

Nous pouvons maintenant définir la fonction permettant de transformer un flux de booléens en une liste d'instructions :

```
Definition decode_flux_opt (lbi : list binary_instruction) : list (option instruction
  map decode lbi.
Definition decode_flux_decoup (lbi : list binary_instruction) : option (list instruct
  traverse (decode_flux_opt lbi).
Definition decode_flux (lb : list bool) : option (list instruction) :=
  let! lbi := cut32 lb in
  decode_flux_decoup lbi.
```

Nous pouvons maintenant définir les théorèmes finaux de notre assembleur qui sont les suivants :

*Remarque* 7. Prouver ces deux théorèmes nécessite un travail assez conséquent au niveau des fonctions de découpage. Cependant ces preuves restant dans la même veine que celles que nous avons déjà vues précédement leur explication ne semble pas justifiée

# 6 Conclusion

Nous avons désormais un assembleur fonctionnel pour MMIX cependant celui- ci ne permet pas encore de se lancer dans la réalisation de l'assembleur certifié x86.
La première raison qui me pousse à d'abord améliorer le parseur de MMIX est la représentation des données. En effet la représentation présentée dans ce rapport est exhaustive car cela est encore possible avec MMIX. D'une part l'architecture RISK limite le nombre de formats d'instructions. En x86 une instruction possède beaucoup plus de champs et la taille de celle ci diffère selon les instructions. J'ai commencé à explorer certaines pistes afin de factoriser MMIX que je ne présente pas dans ce rapport étant donné que celles- ci n'ont pas encore abouties.
Une façon de résoudre ce problème de façon élégante et efficace serait de crée une librairie de parser. En effet un assembleur s'attardant uniquement sur la syntaxe est un cas particulier de parser. Grâce aux combinateurs de parseurs nous pourrions crée dynamiquement de nouveaux parser pour permettre de parser l'ensemble des instructions avec un code factoriser. Le but serait donc de l'implémenter dans un premier temps pour MMIX puis il faudrait adapter celui- ci pour suivre la spécification de x86.

# Références

N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo. Virtual cpu validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 311–327, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi : 10.1145/2815400.2815420. URL `http://doi.acm.org/10.1145/2815400.2815420`.

C. P. Bertot Yves. *Coq'Art : The Calculus of Inductive Constructions*. EATCS Series, 2004. ISBN 3-540-20854-2.

X. Leroy. The compcert c verified compiler – documentation and user's manual – version 2.4, 2014.

S. D. Swierstra. Combinator parsing : A short tutorial. In *LerNet ALFA Summer School*, 2008.