Assembleur x86 certifié

Roman Delgado

Table des matières

1 Introduction

parler du fait que le but pour les assembleurs que l'on va définir est d'avoir une preuve dans les deux sens que l'encodage et le decodage préserve l'instruction (a bien rédiger haha)

2 MMIX et représentation des données

finir cette introduction Le langage d'assembleur MMIX à été crée par Donald Knuth en 1999. Pour l'instant il n'existe aucune implémentation concrète de MMIX il à été crée et utilisée 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'instruction RISK ce qui signifie que la taille des instructions est fixer. Ici une instruction est de taille 32 dont voici le découpage.

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 foix un immediat ou un registre. Ici on ne s'attarde pas sur la sémantique des instructions ce qui nous intérresse est la syntaxe. Une des spécificité de la syntaxe est la possibilité d'obtenir des operandes de plus grande taille avec les formes d'instructions suivantes :

TAG	X	YZ

TAG	XYZ

2.2 Représentation des données

faire une phrase d'intro

Une représentation intuitive des operandes serait de les déclarer de la manière suivante,

Afin de faciliter les preuves pour faire un premier travail en hauteur, j'ai jugé nécéssaire d'éclater le type de donnée afin de ne pas surchager les informations nécéssaires dans la liste d'association mais nous reviendrons sur ce choix dans la partie 4. Le choix à été de stocker l'information dès les étiquettes de la forme

Parler du fait que c'est très exhaustif et que ça marche pour MMIX mais que ça va pas marcher pour le reste

```
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.
```

Ces différents types d'étiquettes vont nous permettre dans les définitions des instructions de les définir de crée un type d'étiquette par syntaxe. Dans le cas de MMIX ce n'est pas très génant étant donné que le nombre de combinaison est assez faible.

Remarque 2. Pour un langage assembleur tel que x86 il ne sera pas possible de definir les instructions de façon exhaustive il faudra utiliser une autre représentation des données. Nous en reparlerons dans la partie mettre la ref

Maintenant nous voulons définir un type de donnée qui puisse représenter une instruction. Etant donné que nous avons dissocié les différents types de tag en fonction de la syntaxe de l'instruction nous allons poursuivre de la même manière en définissant des types pour chaque type d'instructions. En voici un exemple pour les instructions avec 3 opérandes représentants 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 }.
```

Remarque 3. Le record ici est approprié étant donné que l'on souhaite que l'ensemble des champs soient définis dans une instruction pour ne pas laisser de champs vides (ce qui n'aurait pas de sens au niveau syntaxique

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. Si l'on souhaite définir une égalité le type de donnée suivant :

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

Remarque 4. Nous nous attarderons ici sur les types inductifs uniquement

Il va nous falloir crée une fonction exhaustive énumérant l'ensemble des cas pour determiner 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.
```

Ce genre de fonctions sont très fastidieuses à rédiger et l'on constate déja la taille de celle ci pour un type de donnée composé de deux éléments. Si nous souhaitions écrire cette fonction le type "tag" peut etre changer la police 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

Voici un premier lemme qui nous sera très utile par la suite :

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

Ici on désire montrer que si l'on montre que deux étiquettes sont équivalentes à l'aide de notre fonction sur les booléens alors on peut obtenir une preuve de cette égalité dans le monde propositionnel.

3 Conversion N liste B

faire une petite intro la dessus

3.1 Représentation des bits

Enfaite il faut parler de suite de bits (enfin c'est pas le bon mot) mais pas de bits tout court

Dans la section 2.2 nous avons parlé de la représentation des instructions sous forme trouver le mot. Maintenant il nous faut choisir un type de donnée pour représenter les instructions sous forme binaires. Notre type de donnée doit pouvoir nous permettre de :

1. Pouvoir recevoir une chaine de bits de taille n quelconque et multiple de 32 de tel sorte que lors d'un parsing d'un fichier binaire il n'y ai pas de traitement supplémentaire à faire.
2. 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 nécéssaire à son encodage. Nous ne pouvons pas nous permettre d'avoir une taille variable en fonction de la valeur encodée et cela contredirait le premier pré-requis que nous souhaitons.

La structure de donnée par excellence qui satisfait l'ensemble de nos prérequis est la liste. En effet elle permet de répondre à ces deux besoins, nous utiliserons donc des listes de booléens. je vais quand même pas dire que false c'est 0 et true c'est 1 non ? :p

3.2 Fonctions de conversion

Maintenant que nous avons fixé une représentation pour nos suite 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 prêtter attention à cette fonction car elle est une simple implémentation d'une formule bien connue pour determiner la valeur d'un nombre sous forme binaire en forme décimal. 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 des restrictions du 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 lol j'ai pas tout compris à ce truc je suis pas sur de ce que je met et dans ce dernier exemple l'appel récursif ne s'éffectue pas avec le prédécesseur de "k" again comment citer une var mais avec le résultat de "k" par deux. finir ça et trouver la vraie justification tranquillement Placer les ensembles finis (enfin c'est pas le mot exacte)

Remarque 5. Dans notre cas cette fonction nous convient parfaitement car nous souhaitons obtenir une liste de booléens de taille determinée. Cependant cela nous permet d'illustrer la façon dont Coq définie la récursion

enfaite ça va être la seconde (changer puis supprimer) 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" faut que je retrouve comment faire les quotes. Cela nous permet de dire que la fonction peut échouer sans pour autant devoir renvoyer une valeur par défaut où lever une exception. Ici le cas ou la fonction peut échouer est lorsque l'entier naturel que l'on souhaite encoder nécéssite 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 propriétées qui nous intérressent 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 :

En effet cela nous permettra de vérifier qu'une instruction produite par les fonctions des sections suivantes mettre la ref produisent bien des instructions de taille 32.

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

Une des information importante à retenir de ce théorème est sa structure. En effet comme nous avons vus 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 important de cette section. Notre but pour l'ensemble des fonctions que l'on définis est de montrer qu'elles sont trouver le mot ici (c'est pas symétriques mais un truc dans le genre) car elles seront utilisé dans l'encodage et decodage des instructions mettre la ref 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 d'implication en raison des effets de bords induit par la monade option. Pour le théorème suivant nous avons besoin d'émettre une hypothèse supplémentaire sur l'argument "n" encore les vars. Si nous n'effectuons pas cette assertion supplémentaire le théorème n'a pas de sens et n'est pas conséquent pas prouvable.

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

la phrase suivante est nul mais pourquoi pas mettre une petite phrase de conclusion de la partie Nous avons en notre possession l'ensemble des théorèmes nécéssaires pour la suite de nos preuves

4 Conversion Opcode liste B

faire une phrase d'intro de ça

4.1 Structure de donnée

La première étape d'encodage ou de décodage d'une instruction se situe au niveau de l'étiquette. Dans le cas de MMIX comme vus en section 2 une étiquette est associée à un unique opcode. Une façon de stocker cette correspondance est d'utilisé des paires.

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

Nous avons maintenant une représentation pour les associations il nous faut maintenant stocker l'ensemble de celles ci. Etant donné que le nombre d'instructions est de 256 utilisé une représentation sous forme de liste est envisageable. Un argument supplémentaire est que la liste est un type inductif qui nous permettra de raisonner plus aisement lors des preuves étant donné sa relative simplicité. En effet on pourrait utiliser un dictionnaire pour réduire la complexité, cependant l'utilisation de ce genre de type de donnée rendrais les preuves plus complexes pas sur au final :p. Nous obtenons donc le type suivant pour représenté notre collection d'associations :

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

Remarque 6. Un autre argument en faveur de l'utilisation des listes à l'instard des dictionnaires est le fait que cette liste sera définie une unique fois et pourra être utiliser pour effectuer les deux sens de la correspondance.

peut etre placer une remarque sur le fait que on stocke un nat et pas directement la liste de booleen qui correspond C'est même sur qu'il va falloir en parler de ça

faire un paragraphe de pk il faut pas nécéssairement plus d'informations dans la liste d'association avec notre représentation des données "c'est maintenant que le fait d'avoir tout mis a plat prend son sens étant donné que notre liste d'association est très simple ce qui nous permet de faciliter les preuves sur celles ci

4.2 Fonctions de recherche

Il est désormais question de réaliser deux fonctions permettant la recherche au sein d'une liste d'association tel que celle définie 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 mettre la ref. Voici la fonction effectuant la recherche 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.
```

peut etre conclure

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" encore typo et leur entier naturel associé.

faire cet import et faire un truc sympas pour montrer que les premières lignes

Tous les théorèmes que nous définirons se baserons sur cette liste “enc-dec” typo.
Nous somme ici face à un problème car pour vérifier que la liste contient bien l'ensemble des associations nécéssaires le seul moyens de le faire est de calculer dynamiquement cela. Mais nous avons besoin de cette propriété afin de pouvoir réaliser nos futures preuves. Il va nous falloir introduire un nouveau méchanisme permettant à partir d'un résultat obtenue par calcul d'obtenir une proposition traduisant ce résultat dans le monde des propositions. peut être améliorer la phrase précédente. Cette opération en Coq peut être réaliser avec l'aide du type “reflect” :

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 raisonement 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 que 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 un ensemble de fonctions permettant de vérifier une propriété sur l'ensemble de nos tag. Il nous faut donc définir des fonctions comme celles ci pour chaque sous type de tag : sous type ça fait ambigu

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

Pour obtenir la fonction suivante :

```
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
                                              (forall_tag_ter_i2 (fun x => p (tag_t_i
                                              (forall_tag_ter_i3 (fun x => p (tag_t_i
                                              (forall_tag_ter_i4 (fun x => p (tag_t_i
                                              (forall_tag_ter_i5 (fun x => p (tag_t_i
                                              (forall_tag_ter_i6 (fun x => p (tag_t_i
                                              (forall_tag_duo_n (fun x => p (tag_d_n
                                              (forall_tag_duo_i (fun x => p (tag_d_i
                                              (forall_tag_duo_i2 (fun x => p (tag_d_i
                                              (forall_tag_duo_i3 (fun x => p (tag_d_i
                                              (forall_tag_uno (fun x => p (tag_u x)))
```

Grâce à ces fonctions nous allons pouvoir vérifier si un prédicat est vraie pour l'ensemble des tag de notre language.

Nous allons maintenant prouver un premier théorème :

```
Lemma forall_tagP: forall (P : tag → Prop)(f : tag → bool),
    (forall (t : tag), reflect (P t) (f t)) →
    reflect (forall t, P t) (forall_tag f).
```

Ce théorème est très intérressant (même si en l'état il n'est pas très utile) car il nous permet d'affimer au travers de reflect de passer du monde des propositions "(forall (t : tag), reflect (P t) (f t))" à celui du calcul avec notre fonction "forall_tag" "reflect (forall t, P t) (forall_tag f)". Pour prouver ce théorème il nous faut deux théorème préliminaires permettant parler des théorèmes help mais enfaite c'est un des help qui fait ce que je dis au dessus

motiver le imply proprement

```
Lemma implyP: forall A B a b, reflect A a → reflect B b → reflect (A →
B) (imply a b).
```

finir cette partie à tranquille demain

5 Encode Decode

introduction peut etre parler à l'avance du problème d'avoir utiliser un truc bien exhaustif

5.1

6 Conclusion