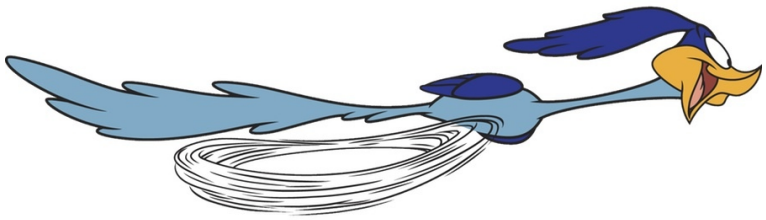
# Documentation MIPPv2

## Extension vectorielle RISC-V



Edgar Baucher

# Table des matières

# 1 L'extension vectorielle de RISC-V

L'architecture RISC-V possède une structure et un jeu d'instruction qui se montre sur certains points fondamentalement différent des autres jeux d'instructions. Sur l'extension vectorielle, la plus singulière des différences est certainement l'utilisation du LMUL :

## 1.1 LMUL (vector Length MULtiplier)

Le LMUL est un nombre pouvant prendre les valeurs : `1/8, 1/4, 1/2, 1, 2, 4` ou `8`. Il fait partie intégrante de la configuration d'un registre vectoriel dans le language assembleur, c'est à dire que le nombre d'éléments dans un tel registre dépend seulement de 3 choses :

— La taille des registres vectoriels simple (déterminée par le matériel).
— Le type des éléments contenu dans le vecteur.
— La valeur du LMUL.

Sa fonctionnalité quand il est inférieur à 1 est légèrement différente par rapport aux autres valeurs, mais sa logique reste la même.

### 1.1.1 LMUL>=1

Lorsque le LMUL est supérieur ou égal à 1, sa valeur désigne le nombre de registres vectoriels groupés qui vont former le vecteur final. Soit SEW (Selected Element Width) le nombre de bits sur lequel est matériellement codé un registre vectoriel, au bas niveau, lors de la configuration d'un registre vectoriel en assembleur, au lieu de réserver SEW bits, la machine va réserver $SEW \times LMUL$ bits pour le vecteur. C'est à dire que tous les registres vectoriels matériels groupés doivent être placés sur une mémoire contigüe. Ainsi, les registres vectoriels des jeux d'instructions plus classiques sont des registres vectoriels configuré avec LMUL=1. Un exemple avec LMUL=2 est illustré en figure 1, où les r$i$, $i \in \{0, nombreRegistresVectoriels - 1\}$ représentent les registres vectoriels matériels.

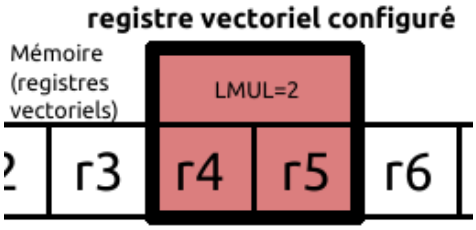



FIGURE 1 – Exemple avec LMUL=2

Cette fonctionnalité trouve son utilité dans l'optimisation de performances. En effet, une fois qu'un registre vectoriel est configuré avec un LMUL>1, effectuer une opération sur ce vecteur ne prendra qu'une seule instruction assembleur, ce qui permet de vectoriser jusqu'à 8 fois plus de données qu'une vectorisation classique.

Cependant, cet ajout fait naître un nouveau problème qui n'existe pas dans les autres jeux d'instructions : tous les registres vectoriels en programmation n'ont pas la même taille. Cette spécialité cause un problème dans l'implémentation de MIPP qui sera illustré en section 2.3.

### 1.1.2 LMUL<1

Puis, quand le LMUL est inférieur à 1, au lieu de grouper des vecteurs, il divise le vecteur : en configurant le registre vectoriel avec LMUL = 1/2, il n'y a que la moitié du registre vectoriel matériel qui est utilisée pour remplir les données et faire les opérations.

Le nombre d'applications de cette fonctionnalité étant relativement restreint, elle n'a pas été inclue à MIPP, il n'est possible que d'utiliser des LMUL supérieurs ou égaux à 1.

## 1.2 Taille agnostique des registres vectoriels

Un autre particularité de l'extension vectorielle du RISC-V est sa gestion agnostique des registres vectoriels lors de la compilation. Ce comportement est aussi utilisé par SVE, son utilité est de former un exécutable portable (pouvant être exécuté par un processeur différent du moment qu'il possède le même jeu d'instruction), mais cela provoque un gros défaut pour les programmeurs : la taille des registres vectoriels n'est pas accessible au moment de la compilation. Le seul moyen de l'obtenir est de l'appeler au moment de l'exécution, ce qui pose des problèmes pour les wrapper SIMD (Single Instruction Multiple Data) existant qui présuppose que la taille des registres vectoriels est connue statiquement.

Ces nouveaux types de registres dont la taille n'est pas connue sont référencés comme des types "sans taille" (`sizeless`) par les compilateurs. La plus-part des compilateurs actuels ne supportent pas la possibilité d'avoir des membres/attributs de type `sizeless` dans les structures/classes ('`sizeless struct`').

De plus, MIPP est un wrapper, or, les wrapper ne peuvent pas stocker de variables. Il est donc impossible de récupérer la taille des registres vectoriel au début du programme pour ensuite la réutiliser.

## 1.3 Fonctions intrinsèques vectorielles

MIPPv2 utilise les fonctions intrinsèques C de l'extension vectorielle de RISC-V. Pour chaque fonction intrinsèque, il faut une version pour chaque type et chaque valeur de LMUL, ce qui multiplie très rapidement le nombre de prototypes pour chaque opération. Afin de pouvoir facilement retrouver la syntaxe d'une fonction avec un type et un LMUL donné, les intrinsèques RISC-V respectent une syntaxe spéciale.

### 1.3.1 Types

Pour les types, la syntaxe est relativement simple ; si l'on veut déclarer un regsitre vectoriel `r` contenant des éléments d'un type `type` codé sur `nBits` bits, configuré avec un LMUL de valeur `LMUL`, on doit respecter la syntaxe :

```
vtypenBitsmLMUL_t r;
```

Par exemple, la déclaration d'un registre vectoriel d'entiers non signé sur 16 bits, avec un LMUL = 4, se fait comme ceci :

```
vuint16m4_t r;
```

Et comme ceci pour un LMUL=1/8 :

```
vuint16mf8_t r;
```

Les champs peuvent prendre les valeurs listées dans la table suivante :

| | |
|---|---|
| `type` | `int, uint, float` |
| `nBits` | `Pour les float: 32, 64.`<br>`Pour le reste: 8, 16, 32, 64.` |
| `LMUL` | `1, 2, 4, 8` |

### 1.3.2 Intrinsèques

La syntaxe des fonctions intrinsèques suit à quelques exeptions près cette règle :

```
vopération_listeMiniType_lettreTypenBitsmLMUL(arguments..., size_t vl)
```

Avec :
- `opération` : le nom simplifié de l'opération (`le` pour un load, `fadd` pour une addition flottante...).
- `listeMiniType` : une suite de lettres correspondant aux types généraux des arguments (par exemple : `vv` pour 2 registre vectoriels en argument, ou `vfv` pour un registre vectoriel, un flottant, puis un registre vectoriel). Avec `v` pour registre vectoriel, `x` pour entier, `f` pour flottant, `u` pour entier non signé et `m` pour masque.
- `lettreType` : une lettre indiquant le type avec lequel va jouer l'opération : `i` pour entier, `f` pour flottant et `u` pour entier non signé.
- `size_t vl` : entier majorant le nombre d'éléments du vecteur sur lequel l'opération a lieu.

Et les autres champs prennent leur valeurs dans la dernière table.

Voici un exemple de prototype C d'une intrinsèque (opération d'addition entre un registre vectoriel (configuré avec LMUL=2) et un entier sur 8 bits) :

```
vint8m2_t vadd_vx_i8m2(vint8m2_t op1, int8_t op2, size_t vl);
```

# 2 Wrapper de vectorisation : MIPPv2

## 2.1 Implémentation en 3 couches

MIPPv2 est une interface en 3 couches : une interface LLI (Low Level Interface) en C pour une programmation plus bas niveau, une interface MLI (Medium Level Interface) en C++, construite à partir de la première couche et offrant les mêmes fonctionnalités, mais avec une utilisation plus simple et moins verbeuse. Et une dernière interface HLI (Hight Level Interface) en C++, fondée sur la couche MLI, utilisant des classes et la surcharge d'opérateur C++ pour un maniement de la vectorisation plus aisé visuellement.

### 2.1.1 Interface bas niveau (C)

Dans cette interface (nommée `mipp.h`, qui inclut les headers propres aux jeux d'instruction), les noms des fonctions et types sont relativement verbeux, mais elle trouve son utilité dans la normalisation des noms de fonctions MIPP : elle permet de fonder une syntaxe des intrinsèques commune, qui pourra donc être utilisé indépendamment du jeu d'instruction dans les interfaces plus haut niveau.

C'est à ce niveau que s'établit la jonction entre le jeu d'instruction et la bibliothèque MIPPv2. Le header `mipp.h` rassemble les autres headers de chaque jeux d'instructions et sélectionne à la compilation (phase préprocesseur) celui adapté. Ainsi, dans ces headers, sont déclarés et définis tous les prototypes des fonctions MIPPv2 basés sur les intrinsèques du jeu d'instruction en question. Et c'est aussi `mipp.h` qui se charge de créer pour chaque opération et chaque type vectoriel une version sans LMUL (fixé à 1 par défaut) pour des usages plus simple de MIPPv2 (en appliquant des simples renommage pour les types, et en faisant des simples appels aux fonctions originales avec LMUL=1 pour les fonctions).

Étant donné qu'il y a besoin, pour chaque opération vectorielle, d'une fonction associée à cette opération pour chaque type, et chaque valeur de LMUL, et que ces détails doivent apparaître dans le nom de la fonction (car une opération pour deux types de registres vectoriels différent fait appel à une intrinsèque différente), les noms des fonctions MIPPv2 deviennent très vite verbeux, d'où l'intérêt de l'interface de niveau supérieur (moyen niveau).

### 2.1.2 Interface moyen niveau (C++)

L'interface de moyen niveau (`mipp.hpp`) permet d'établir un jeu de fonctions bien plus lisible. Son rôle est de, pour chaque opération vectorielle, prendre toutes les fonctions MIPPv2 de l'interface C correspondant à cette opération, et de les surcharger en une seule fonction avec un nom très simple. La surcharge de fonctions en C++ permettra ainsi d'utiliser le même nom de fonction avec des registres vectoriels de types différents pour une même opération. Et les noms des types vectoriels, prennent un nom plus court à l'aide des templates C++.

Ce header est par conséquent indépendant du jeu d'instruction de la machine, il n'est donc codé qu'une seule fois et l'ajout d'un nouveau jeu d'instruction à MIPPv2 ne lui provoquera aucun changement. Les nouveaux noms de fonctions et de types obtenus, ainsi que leur syntaxe C sont détaillées dans la section 2.2.

### 2.1.3 Interface haut niveau (C++)

Ce header (`mipp_object.hpp`) est chargé de gagner en expressivité dans d'utilisation de MIPPv2 en créant des classes C++ représentant les registres vectoriels, et en utilisant la surcharge d'opérateurs pour écrire facilement du code vectoriel.

Cependant, les compilateurs ne pouvant actuellement pas gérer les `sizeless struct` dans des classes, il n'est pas compatible avec l'extension vectorielle de RISC-V, et il n'a pas encore été inclut à MIPv2.

## 2.2 Utilisation

### 2.2.1 Types

Dans l'interface bas niveau C, les noms des types dans MIPPv2 bas niveau sont le résultat de l'application d'un `typedef` sur un nom de type explicite. Ils prennent cette syntaxe :

| | |
|---|---|
| Registre vectoriel : | `rvd_type_mLMUL_t` |
| Masque : | `rvm_type_mLMUL_t` |

Où `type` et `LMUL` peuvent prendre les valeurs :

| | |
|---|---|
| `type` | `int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, float64` |
| `LMUL` | `1, 2, 4, 8` |

Pour l'interface moyen niveau C++, la déclaration d'un registre vectoriel ou d'un masque se fait par l'utilisation des templates :

```
rvd<type, lmul> r;
rvm<type, lmul> m;
```

Où les types et les valeur de `lmul` possible sont listées dans ce tableau :

| | |
|---|---|
| `type` | `int8_t, int16_t, int32_t, int64_t (int), uint8_t, uint16_t, uint32_t, uint64_t (unsinged int), float32_t (float), float64_t (double)` |
| `LMUL` | `1, 2, 4, 8, ∅` |

Afin d'épargner la gestion et compréhension du `LMUL` aux utilisateurs qui n'en n'ont pas besoin, il est possible dans les deux interfaces MIPPv2 de ne pas indiquer de `LMUL`, qui aura pour valeur par défaut `LMUL=1`. Ce qui donne ces syntaxes de type possible :

| | Moyen niveau (C++) | Bas niveau (C) |
|---|---|---|
| Registre vectoriel | `rvd<type>` | `rvd_type_t` |
| Masque | `rvm<type>` | `rvm_type_t` |

### 2.2.2 Fonctions

**Interface LLI :**

La surcharge de fonction n'existant pas en C, le nommage des fonctions C suit cette règle (à quelques exceptions près, indiquées dans la Table 2) :

| Version | Prototypes valides C (syntaxe) |
|---|---|
| Sans masque | `mipp_nomFonction_type_mLMUL`<br>`mipp_nomFonction_type` (LMUL=1 par défaut) |
| Avec masque | `mipp_nomFonction_type_mLMUL_m`<br>`mipp_nomFonction_type` (LMUL=1 par défaut) |

Où les champs à changer peuvent prendre ces valeurs :

| | |
|---|---|
| `nomFonction` | `load, store, add...` |
| `type` | `int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, float64` |
| `LMUL` | `1, 2, 4, 8` |

Par exemple, en C, la fonction d'addition masquée de deux registres vectoriel flottants sur 32 bits avec un LMUL=2 a pour nom : `mipp_add_float32_m2_m`.

**Interface MLI :**

Dans la liste de prototypes C++ qui va suivre, les * indiquent que le rôle fonction est détaillé dans l'explication 2.2.2.

```
int  N<type, lmul> N()                                                    MIPP N*
void load(const type* addr, rvd<type,lmul>& r)                            load*
void loadu(const type* addr, rvd<type,lmul>& r)                           unaligned load*
void cmask(const type* addr, rvd<type,lmul>& r)                           cmask*
void store(const type* addr, rvd<type,lmul> r)                            store*
void set(rvd<type,lmul>& r, type scalar)                                  set
void set1(rvd<type,lmul>& r, type scalar)                                 set
void set1(rvm<type, lmul>& r, bool scalar)                                set for mask
void set0(rvd<type,lmul>& r)                                              fill with 0
void set0(rvm<type, lmul>& r)                                             fill with 0
rvd<type,lmul> add(rvd<type,lmul> r1, rvd<type,lmul> r2)                   r1+r2
rvd<type,lmul> sub(rvd<type,lmul> r1, rvd<type,lmul> r2)                   r1-r2
rvd<type,lmul> mul(rvd<type,lmul> r1, rvd<type,lmul> r2)                   r1*r2
rvd<type,lmul> div(rvd<type,lmul> r1, rvd<type,lmul> r2)                   r1/r2
rvd<type,lmul> min(rvd<type,lmul> r1, rvd<type,lmul> r2)                   min
rvd<type,lmul> max(rvd<type,lmul> r1, rvd<type,lmul> r2)                   max
```

| | |
|---|---|
| `rvd<type,lmul> andb(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1&r2* |
| `rvm<type, lmul> andb(rvm<type, lmul> r1, rvm<type, lmul> r2)` | r1 && r2* |
| `rvd<type,lmul> orb(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1 \| r2 |
| `rvm<type, lmul> orb(rvm<type, lmul> r1, rvm<type, lmul> r2)` | r1 \|\| r2 |
| `rvd<type,lmul> xorb(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1 ^ r2 |
| `rvm<type, lmul> xorb(rvm<type, lmul> r1, rvm<type, lmul> r2)` | (r1 && !r2) \|\| ( !r1 && r2) |
| `rvd<type,lmul> andnb(rvd<type,lmul> r1, rvd<type,lmul> r2)` | ( !r1)&r2 |
| `rvm<type, lmul> andnb(rvm<type, lmul> r1, rvm<type, lmul> r2)` | !r1 && r2 |
| `rvd<type,lmul> lshiftr(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1«r2* |
| `rvd<type,lmul> rshiftr(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1»r2 |
| `rvd<type,lmul> lshift(rvd<type,lmul> r1, type scalar)` | r1«scalar |
| `rvd<type,lmul> rshift(rvd<type,lmul> r1, type scalar)` | r1»scalar |
| `rvd<type,lmul> blend(rvd<type,lmul> r1, rvd<type,lmul> r2, rvm<type, lmul> m)` | blend* |
| `rvd<type,lmul> sat(rvd<type,lmul> r1, type min, type max)` | saturate* |
| `rvd<type,lmul> fmadd(rvd<type,lmul> r1, rvd<type,lmul> r2, rvd<type,lmul> r3)` | (r1*r2)+r3 |
| `rvd<type,lmul> fnmadd(rvd<type,lmul> r1, rvd<type,lmul> r2, rvd<type,lmul> r3)` | -(r1*r2)+r3 |
| `rvd<type,lmul> fmsac(rvd<type,lmul> r1, rvd<type,lmul> r2, rvd<type,lmul> r3)` | (r2*r3)-r1 |
| `rvd<type,lmul> fnmsac(rvd<type,lmul> r1, rvd<type,lmul> r2, rvd<type,lmul> r3)` | -(r2*r3)+r1 |
| `rvd<type,lmul> fmacc(rvd<type,lmul> r1, rvd<type,lmul> r2, rvd<type,lmul> r3)` | (r2*r3)+r1 |
| `rvd<type,lmul> fnmacc(rvd<type,lmul> r1, rvd<type,lmul> r2, rvd<type,lmul> r3)` | -(r2*r3)-r1 |
| `rvd<type,lmul> sqrt(rvd<type,lmul> r)` | $\sqrt{r}$ |
| `rvd<type,lmul> rsqrt(rvd<type,lmul> r)` | $1/\sqrt{r}$ |
| `rvd<type,lmul> notb(rvd<type,lmul> r)` | binar NOT |
| `rvm<type, lmul> notb(rvm<type, lmul> r1)` | !r |
| `rvd<type,lmul> neg(rvd<type,lmul> r)` | -r |
| `rvm<type, lmul> cmpeq(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1==r2* |
| `rvm<type, lmul> cmpge(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1>=r2 |
| `rvm<type, lmul> cmple(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1<=r2 |
| `rvm<type, lmul> cmpgt(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1>r2 |
| `rvm<type, lmul> cmplt(rvd<type,lmul> r1, rvd<type,lmul> r2)` | r1<r2 |
| `rvm<type, lmul> sign(rvd<type,lmul> r)` | the sign of r |
| `unsigned long popc(rvm<type, lmul> m)` | number of 1 |
| `rvd<type,lmul> div2(rvd<type,lmul> r)` | r/2 |
| `rvd<type,lmul> div4(rvd<type,lmul> r)` | r/4 |
| `type testz(rvd<type,lmul> r)` | r==(0,0...0) |
| `bool testz(rvm<type, lmul> m)` | m==(0,0...0) |
| `type sum(rvd<type,lmul> r)` | sum of r |
| `type hadd(rvd<type,lmul> r)` | sum of r |
| `type hmul(rvd<type,lmul> r)` | mul of r |
| `type hmin(rvd<type,lmul> r)` | min of r |

| | |
|---|---|
| `rvd<type,lmul> lrot(rvd<type,lmul> r)` | 1left rotation |
| `rvd<type,lmul> rrot(rvd<type,lmul> r)` | 1right rotation |
| `rvd<type,lmul> interleavehi(rvd<type,lmul> r1, rvd<type,lmul> r2)` | high interleave* |
| `rvd<type,lmul> interleavelo(rvd<type,lmul> r1, rvd<type,lmul> r2)` | low interleave* |
| `rvd<type,lmul> suff(rvd<type, lmul> r1, rvd<uint_type, lmul> r2)` | shuffle* |
| `rvd<type,lmul> slideup(rvd<type,lmul> r1, rvd<type,lmul> r2, size_t d)` | right slide* |
| `rvd<type,lmul> slidedown(rvd<type,lmul> r1, rvd<type,lmul> r2, size_t d)` | left slide* |
| `type getfirst(rvd<type,lmul> r1)` | getfirst element |
| `rvd<type1, lmul> cvt(rvd<type2, lmul> r)` | convert* |
| `type reduction<type,lmul, mipp_function>(rvd<type,lmul> r)` | reduction* |

TABLE 1 – Liste des prototypes MIPPv2 moyen niveau.

Compléments explicatifs :

- **`N`** : Renvoie le nombre d'éléments dans un registre vectoriel.
- **`load/loadu`** : Rempli le registre rentré en paramètre avec les valeurs présente à l'adresse indiquée.
- **`store`** : Écrit le contenu du registre vectoriel à l'adresse indiquée
- **`andb`** : opération bit à bit entre chaque élément du registre/masque (ET binaire ici, mais le comportement est le même pour toutes les opérations binaires comme &, | ou ^).
- **`lshiftr`** : Chaque élément `e1` du registre vectoriel 1 subit l'opération `e1«e2` avec `e2` l'élément de registre vectoriel 2 correspondant. Pour les versions scalaires, c'est l'opération `e1»scalar` qui est appliquée.
- **`blend`** : Le registre vectoriel retourné possède les valeurs de `r1` là où le masque `m` contient des `1`, et les valeurs de `r2` là où `m` contient des `0`.
- **`sat`** : Les éléments `e` du registre vectoriel subissent l'opération : `e = (e > max) ? (max) : ( (e < min) ? (min) : e ).`
- **`cmpeq`** : Les opérations de comparaisons renvoient un masque contenant 0 là où la comparaison a renvoyé **`faux`** et 1 là où **`vrai`** a été renvoyé. Ici la comparaison est le test d'égalité, mais le fonctionnement est le même pour toutes les autres comparaisons.
- **`interleavehi`** : Prend la première moitié des deux registres vectoriels pour former le registres vectoriel de retour
- **`interleavelo`** : Prend la deuxième moitié des deux registres vectoriels pour former le registres vectoriel de retour
- **`shuff`** : Renvoie un registre vectoriel construit par placement de chaque éléments de `r1` à l'indice indiqué par l'entier non signé de `r2` correspondant.
- **`slideup`** : Décalle tous les éléments de `r1` de `d` indices vers la droite, et les `d` premiers éléments prendront les valeurs des éléments des `d` premiers éléments de `r2`.
- **`slidedown`** : Décalle tous les éléments de `r1` de `d` indices vers la gauche, et les `d` derniers éléments prendront les valeurs des éléments des `d` derniers éléments de `r2`.
- **`convert`** : Converti un vecteur flottant en un vecteur entier (signé ou non), ou l'inverse.
- **`reduction`** : Applique une opération au choix à la chaîne sur tous les éléments du registre vectoriel : pour un `add`, renvoie la somme des éléments du vecteur, pour un `add`, renvoie le produit, pour un `div`, renvoie le résultat de la division successive des éléments, pour un `min`, renvoie le plus petit élément ect...

La Table 1 désigne les prototypes des versions non masquées des fonctions de MIPPv2 moyen niveau (C++, sans classe). Toutes ces fonctions sont surchargées pour tous les types, valeurs de LMUL, et versions masquées, non masquées, et masquées avec un src (registre vectoriel indiquant la valeur à prendre pour les éléments masqués). Ainsi, pour une fonction donnée, on peut faire appel à sa version masquée en ajoutant un argument de masque, et de même pour la version masquée avec src où il suffit

d'ajouter en argument le src :

| Version | Utilisation (C++) |
|---|---|
| Sans masque | `nomFonction(arguments...)` |
| Avec masque | `nomFonction(arguments..., masque)` |
| Avec masque et src | `nomFonction(arguments..., masque, src)` |

Cependant, certaines fonctions ne sont pas compatibles avec tous les types, d'autres n'ont pas de versions masquées/avec src, et enfin d'autres ont un nommage légèrement différent. Ces spécificités sont énumérées dans le tableau suivant (table 2) :

| **Nom fonction (C++)** | **Masquage**<br>NM = non masquée<br>M = masquée<br>MR = avec src<br>* = tout compatible | **Types compatibles**<br>* = tout | **Syntaxe C (si exceptionnelle)**<br>- = Non exceptionnelle |
|---|---|---|---|
| `N` | NM | * | - |
| `load` | NM et M | * | `mipp_load_type_mLMUL(const type* addr)` |
| `loadu` | NM et M | * | `mipp_loadu_type_mLMUL(const type* addr)` |
| `cmask` | NM et M | `uint_*` | `mipp_cmask_type_mLMUL(const type* addr)` |
| `store` | NM et MR | * | - |
| `set` | NM | * | `mipp_store_type_mLMUL(type scalar)` |
| `set1 (registre)` | NM | * | `mipp_set1_type_mLMUL(type scalar)` |
| `set1 (masque)` | NM | * | `mipp_set1m_type_mLMUL(uint_8 scalar)` |
| `set0 (registre)` | NM | * | `mipp_set0_type_mLMUL()` |
| `set0 (masque)` | NM | * | `mipp_set0m_type_mLMUL()` |
| `add` | * | * | - |
| `sub` | * | * | - |
| `mul` | * | * | - |
| `div` | * | * | - |
| `min` | * | * | - |
| `max` | * | * | - |
| `andb (registre)` | * | `int_*` et `uint_*` | - |
| `andb (masque)` | * | `int_*` et `uint_*` | - |
| `orb (registre)` | * | `int_*` et `uint_*` | - |
| `orb (masque)` | * | `int_*` et `uint_*` | - |
| `xorb (registre)` | * | `int_*` et `uint_*` | - |
| `xorb (masque)` | * | `int_*` et `uint_*` | - |
| `andnb (registre)` | * | `int_*` et `uint_*` | - |
| `andnb (masque)` | * | `int_*` et `uint_*` | - |
| `lshiftr` | * | `int_*` et `uint_*` | - |
| `rshiftr` | * | `int_*` et `uint_*` | - |
| `lshift` | * | `int_*` et `uint_*` | - |
| `rshift` | * | `int_*` et `uint_*` | - |
| `blend` | NM | * | - |
| `sat` | * | * | - |
| `fmadd` | NM et M | `float_*` | - |
| `fnmadd` | NM et M | `float_*` | - |
| `fmsac` | NM et M | `float_*` | - |
| `fnmsac` | NM et M | `float_*` | - |
| `fmacc` | NM et M | `float_*` | - |
| `fnmacc` | NM et M | `float_*` | - |

| | | | |
|---|---|---|---|
| sqrt | * | float_* | - |
| rsqrt | * | float_* | - |
| notb (registre) | * | int_* et uint_* | - |
| notb (masque) | * | int_* et uint_* | - |
| neg | * | int_* | - |
| cmpeq | * | * | - |
| cmpne | * | * | - |
| cmple | * | * | - |
| cmplt | * | * | - |
| cmpge | * | * | - |
| cmpgt | * | * | - |
| sign | * | * | - |
| popc | NM et M | * | - |
| div2 | NM | int_* et uint_* | - |
| div4 | NM | int_* et uint_* | - |
| testz (registre) | NM | int_* et uint_* | mipp_testz_type_mLMUL(...) |
| testz (masque) | NM | int_* et uint_* | mipp_testzm_type_mLMUL(...) |
| sum | NM et M | * | - |
| hadd | NM et M | * | - |
| hmul | NM et M | * | - |
| hmin | NM et M | * | - |
| lrot | NM | * | - |
| rrot | NM | * | - |
| shuff | NM | * | - |
| slideup | NM | * | - |
| slidedown | NM | * | - |
| getfirst | NM | * | - |
| cvt | NM | int32/64⇔float32/64<br>uint32/64⇔float32/64 | mipp_cvt_type1_mLMUL_type2_mLMUL(...) |
| reduction | NM | * | macro:<br>MIPP_REDUCTION_TYPE_MLMUL(<br>variable, registre, opération)<br>variable = reduction_opération(regitre) |

TABLE 2 – Compatibilités des fonction

Enfin, la dernière chose à savoir sur l'utilisation de ces fonctions est la place du registre accumulateur : pour les fonctions retournant un registre vectoriel sur lequel a été effectuée une opération (par exemple `add`, `xorb`, `min` ou encore `fmadd`), le compilateur traduit en code assembleur en appliquant l'opération voulue sur **le premier** registre vectoriel passé en paramètre, c'est le registre accumulateur. Puis, si dans le code C/C++ c'est un autre registre vectoriel qui prend la valeur de retour, alors le compilateur va s'arranger pour sauvegarder avant (si nécessaire) le registre sur lequel l'opération est faite, ce qui introduit des instructions assembleurs supplémentaire. S'il n'y a pas besoin de sauvegarder l'ancienne valeur du registre accumulateur, alors ces instructions peuvent être évités en renvoyant la valeur de retour de la fonction sur le registre accumulateur. D'où l'utilité des subtiles différences entre les `fmadd`,`fmsac`, `fmacc`, `fnmadd`,`fnmsac` et `fnmacc` : grâce à ces fonctions, on peut jouer avec le registre accumulateur pour en utiliser le moins possible.

## 2.3 Gestion des types vectoriels à taille agnostique

Le problème est ici de faire un type commun du nom de `rvd` qui doit réunir tous les types de registre vectoriel. La solution utilisée dans MIPPv1 pour les autres jeux d'instructions était d'utiliser un type quelconque de registre vectoriel et de mettre dedans les données que l'on veut indépendamment du type en appliquant des `static cast` entre les vecteurs utilisés dans les intrinsèques et le registre de type commun `rvd`, ce qui fonctionne car les registres vectoriels faisaient dans tous les cas des tailles identiques. Or, avec l'apparition du LMUL dans l'extension vectorielle de RISC-V, les registres vectoriels en C n'ont pas tous la même taille, ce qui rend impossible cette stratégie.

Afin de remédier à ce problème, la stratégie fut d'utiliser les templates C++ sur les types et non sur les fonctions. C'est ici une différence fondamentale entre MIPPv1 et MIPPv2 : dans MIPPv2 la déclaration d'un registre vectoriel s'opère avec une configuration via template, comme l'a expliqué la section 2.2.1.

Cependant, même avec cette stratégie, quelques obstacles sont toujours présents : les types de registres vectoriels étant tous distincts, l'implémentation concrète du type configurable via template se profile bien en utilisant la spécialisation de template sur une structure contenant le registre vectoriel : il suffirait de rentrer pour chaque couple `<type, LMUL>` de rentrer dans la structure le type vectoriel correspondant, mais il faut rappeler que en C, les registres vectoriels sont des `sizeless struct`, donc on ne peut les placer en tant qu'attribut d'une structure. Ce qui rend cette approche impossible. La spécialisation de template n'étant pas applicable ni aux `typedef` (C) ni aux `using` (C++), un simple renommage de type est lui aussi compromis.

La solution utilisée fut donc de passer par une fonctionnalité du C++ qui est de pouvoir rentrer un type (pas l'objet typé, mais bien le type) comme une forme d'attribut d'une structure à l'aide du mot clé `using`. C'est à dire qu'il est possible de faire ceci (si l'on prend un exemple avec le type `int` en attribut de la structure) :

```
struct nomStructure{
   using type = int;
};
```

Avec cette stratégie, il est désormais possible de passer le type des registres vectoriel dans des structures, et de faire de la spécialisation de template sur ces structures. C'est à dire de répéter ces lignes de code pour tous coupe `<type, LMUL>` :

```
template <> struct vector_t<type, LMUL>{
   using type = typeRegsitreVectorielC;
};
```

Par exemple, pour la spécialisation de template pour le type vectoriel flottant configuré avec un LMUL=8, on obtient cette déclaration :

```
template <> struct vector_t<float, 8>{
   using type = rvd_float64_m8_t;
};
```

Ce n'est toujours pas fini car cette méthode n'offre pas encore un type vectoriel accessibles sous une même syntaxe. Mais maintenant que les types vectoriels ont été rentré dans des structures accessible par passage d'argument template, les templates peuvent être appliqués à des `using`. En effet, si la spécialisation de template n'est pas possible sur le mot clé `using`, l'utilisation de template sans spécialisation direct l'est. Il ne reste donc qu'à accéder à ces types en déclarant le template de cette manière :

```
template <typename T, int LMUL=1> using rvd = typename vector_t <T,LMUL>::type;
```

Ainsi on obtient bien le type `rvd` qui réunit tous les types vectoriels, configurable avec ses arguments template lors de sa déclaration. Il en est de même pour la gestion des registres masqués :

Spécialisation :

```
template <> struct mask_t<type, LMUL>{
   using type = typeRegsitreMasquéC;
};
```

Déclaration avec template :

```
template <typename T, int LMUL=1> using rvm = typename mask_t <T,LMUL>::type;
```

Et on retrouve comme prévu la syntaxe décrite en section 2.2.1.

## 2.4 Génération des fonctions

Comme les sections précédentes ont pu le montrer, avec le LMUL, tous les types et le masquage, il y a un nombre colossal de fonction à construire (issu du produit cartésien entre la liste des noms des fonctions et ces derniers paramètres). De manière exacte, il y a en tout **1212** opérations à déclarer et implémenter simplement pour la version C (il y en a donc au moins le double pour la version C++). Dans MIPPv1, les fonctions étaient déclarées en brut dans le header C++, ce qui serait ici bien trop long en raison du nombre de fonctions, sans parler des déclarations de types et des prototypes C++. De plus, un changement ou un ajout provoquerait un nombre de changement à faire dans le code trop élevé.

C'est ici qu'interviennent les MACROs. Avec cette fonctionnalité du langage C, il est possible d'automatiser la déclaration et définition des prototypes des fonctions MIPP. La stratégie est de construire des MACROs qui, en passant en paramètre le nom d'une opération vectorielle, son type et son LMUL, construisent la déclaration de la fonction MIPPv2 associée, puis de les appliquer pour chaque types et valeurs de LMUL. Cette méthode est réalisable grâce à la concaténation syntaxique offerte par les MACROs : comme les intrinsèques respectent une syntaxe relativement régulière (décrite en section 1.3.2), on peut facilement générer leurs noms. Voici un exemple de génération de nom de fonction intrinsèque qui prend deux registre vectoriels en argument :

```
#define INTRINSIC_NAME(nom_opération , lettreType, nBits, LMUL)\
  v ## nom_opération ## _vv_ ## lettreType ## nBits ## m ## LMUL
```

De même pour les types (syntaxe décrite en section 1.3.1) :

```
#define VECTOR_TYPE(type, nBits, LMUL) v ## type ## nBits ## m ## LMUL ## _t
```

Cependant, il existe des intrinsèques qui ne respectent pas la syntaxe décrite précédemment, il faut donc pour chacune créer une MACRO qui permet de générer son nom. Puis, il faut ensuite générer la fonction MIPPv2. L'objectif est de généraliser le plus possible l'écriture de la fonction à l'aide des MACROs. Pour illustrer la stratégie, on peut partir d'une opération précise, que l'on va peut à peut généraliser. Considérons dans un premier temps l'opération d'addition de deux vecteurs d'entier sur 64 bits, avec un LMUL égal à 2. La fonction se déclare comme ceci :

```
#define GENERATE_ADD_INT64_LMUL2()                                          \
  rvd_int64_m2_t mipp_add_int64_m2(rvd_int64_m2_t a1, rvd_int64_m2_t a2) {  \
     return vadd_vv_i64m2(a1, a2, MAX_VL);                                  \
  }                                                                         \
```

Ensuite, cette MACRO peut être généralisée pour tous les types et valeur de LMUL (avec indiqué en pointillés le nom de la fonction mipp et de l'intrinsèque utilisé) :

```
#define GENERATE_ADD(type, nBits, LMUL, lettreType)                                  \
  VECTOR_TYPE(type, nBits, LMUL) mipp_add_ ## type ## nBits ## LMUL ## _m ## LMUL(   \
VECTOR_TYPE(type, nBits, LMUL) a1, VECTOR_TYPE(type, nBits, LMUL) a2) {              \
    return vadd_vv_ ## lettreType ## nBits ## m ## LMUL (a1, a2, MAX_VL);           \
  }                                                                                  \
```

Puis, la syntaxe des opérations mathématiques à deux vecteurs étant très régulière, la généralisation de l'opération est donc possible, du moment qu'elle prend deux registres vectoriels en entrée et qu'elle en renvoie un (les changement par rapport à la dernière MACRO sont surlignés en rouge) :

```
#define GENERATE_2_ARGS_VECTOR_OPERATION(type, nBits, LMUL, lettreType, nom_op)                \
  VECTOR_TYPE(type, nBits, LMUL) mipp_ ## nom_op ## _ ## type ## nBits ## LMUL ## _m ## LMUL(  \
VECTOR_TYPE(type, nBits, LMUL) a1, VECTOR_TYPE(type, nBits, LMUL) a2) {                        \
    return v ## nom_op ## _vv_ ## lettreType ## nBits ## m ## LMUL (a1, a2, MAX_VL);           \
  }                                                                                            \
```

Il existe aussi des opérations semblables mais sur des masques, ainsi, une généralisation sur le type de l'opération est possible :

```
#define GENERATE_2_ARGS_OPERATION(type, nBits, LMUL, lettreType, nom_op, TYPE, argTypes)  \
  TYPE(type, nBits, LMUL) mipp_ ## nom_op ## _ ## type ## nBits ## LMUL ## _m ## LMUL(   \
TYPE(type, nBits, LMUL) a1, TYPE(type, nBits, LMUL) a2) {                                \
    return v##nom_op##_##argTypes##_##lettreType##nBits##m##LMUL (a1, a2, MAX_VL);       \
  }                                                                                      \
```

Et ainsi de suite, en généralisant le type de retour (registre vectoriel, `void`, scalaire...), le nombre d'arguments et la structure, n'importe quelle fonction MIPP peut totalement être construite à partir de quelques MACROs, en passant les bons arguments de construction et de spécialisation. Pour illustrer la spécialisation, dans l'exemple précédent, on a l'égalité entre ces appels de MACROs :

```
GENERATE_ADD_INT64_LMUL2()
==
GENERATE_ADD(int, 64, 2, i)
==
GENERATE_2_ARGS_VECTOR_OPERATION(int, 64, 2, i, add)
==
GENERATE_2_ARGS_OPERATION(int, 64, 2, i, add, VECTOR_TYPE, vv)
```

En codant uniquement les MACROs les plus généralisées (dans la figure ci dessous, la plus généralisée est la dernière), les fonctions MIPP peuvent juste être déclarée en précisant leur arguments de construction. Et même mieux : il est possible pour une opération d'automatiser sa construction pour tous les types et valeurs de LMUL. Il suffit que ses 4 premiers arguments soient le type, le nombre de bits, le LMUL, et la lettre représentant le type (issue de la syntaxe RISC-V) : une fois cette condition respectée, il suffit de créer une MACRO qui prend une MACRO de génération en argument, et qui l'appelle pour tous les types et valeurs de LMUL. Voici un exemple pour faire varier le nombre de bits de lequel est codé le type :

```
#define SWITCH_NBITS(MACRO, type, LMUL, lettretType, ...)  \
  MACRO(type,  8, LMUL, lettreType, __VA_ARGS__)           \
  MACRO(type, 16, LMUL, lettreType, __VA_ARGS__)           \
  MACRO(type, 32, LMUL, lettreType, __VA_ARGS__)           \
  MACRO(type, 64, LMUL, lettreType, __VA_ARGS__)           \
```

De cette manière, l'appel de :

```
SWITCH_NBITS(GENERATE_2_ARGS_OPERATION, int, 2, i, add, VECTOR_TYPE, vv)
```

entraînera la création de la fonction MIPP d'addition de vecteurs pour tous les types d'entier signés. Il ne reste qu'ensuite à créer les MACROs `SWITCH_LMUL` et `SWITCH_TYPE`, et l'opération voulue sera crée pour tous types de registres vectoriels.

Finalement, la tâche de génération des fonctions revient avec cette méthode à rentrer tous les noms d'opérations MIPP avec les bons arguments de construction dans une MACRO de génération automatique.

## 2.5 Tests

### 2.5.1 Compilation

L'extension vectorielle de RISC-V étant très récente, les compilateurs C/C++ ont quelques difficultés à gérer la production de code assembleur à partir des intrinsèques vectorielles. Par exemple, g++ produit un code assembleur très peu optimisé, rempli d'instructions inutiles et parasites. La Figure 2 illustre ce problème sur un extrait de code d'une succession de simple opérations vectorielles.

```
16      vfmul.vf    v12,v2,ft3,v0.t
17      vmv1r.v     v13,v12
18      vsetvli     a3,a5,e32,m1
19      vfadd.vv    v13,v13,v16,v0.t
20      vsetvli     a3,a5,e32,m1
21      vsetvli     a3,a5,e32,m1
22      vsetvli     a3,a5,e32,m1
23      vmv1r.v     v2,v4
24      vfmul.vv    v2,v3,v3,v0.t
```

FIGURE 2 – Exemple de code assembleur généré par g++

L'opération `vsetvli a3,a5,e32,m1` est chargée de la configuration d'un registre vectoriel, qui n'est nécessaire d'exécuter qu'une seule fois. Et on voit ici qu'elle est répétée inutilement 3 fois d'affilée. Ce problème est récurrent à chaque utilisation de registres vectoriel dans le code C/C++. D'autres défaut de production de code assembleur sont par ailleurs présent, comme l'ajout d'opérations inutiles du type `x=x` suite à l'utilisation d'intrinsèques vectorielles.

Cependant, ces défaut de compilation ne concernent que g++. En effet, clang++ fonctionne mieux et produit un code assembleur bien plus propre.

### 2.5.2 Exécution

Afin de tester l'extension vectorielle sur une machine non RISC-V, il existe plusieurs émulateurs. Premièrement, il existe `spike pk` qui est à jour sur l'extension. Cependant il présente des simplifications qui biaise les tests, c'est à dire qu'il ne simule uniquement le code assembleur et son jeu d'instruction, sans se préoccuper de l'architecture du processeur. Or, l'intérêt du code vectoriel est de comparer les performances face à un code séquentiel, à l'aide de la mesure du nombre de cycles horloge de la machine. Ce qui est impossible à l'aide de `spike pk`, car cette mesure serait non conforme à un réel processeur RISC-V.

Il existe aussi le simulateur `GEM5` qui, contrairement à `spike pk`, simule l'architecture du processeur. Grâce à cela, il est possible d'avoir une réelle idée du nombre de cycles horloges et temps

d'exécution d'un programme. Malheureusement, ce simulateur n'est pas à jour sur l'extension vectorielle du RISC-V. Il utilise ancienne version, dans laquelle il n'y a par exemple pas l'argument `vl` dans les arguments des intrinsèques vectorielles. Cependant, `GEM5` étant très configurable, il est possible de le ramener à la version de l'extension actuelle.

### 2.5.3 Algorithmes

Programmer de manière vectorielle et séquentiel requiert des stratégies de code différentes. Un exemple de comparaison de code séquentiel avec le vectoriel est présenté dans le tableau 3. Il s'agit du code de génération du fractale de Mandelbrot dans la boucle qui parcourt tous les pixels. Un algorithme permet d'obtenir la couleur du pixel sélectionné en fonction de sa position. Cette couleur est calculée avec le nombre d'itérations parcourue par l'algorithme sur ce pixel, où dans chaque itération une norme est calculée à partir de la précédente, et l'itération suivante est lancée tant que cette norme ne dépasse pas une certaine valeur (`BAIL_OUT`). Le code vectoriel a donc pour principe de traiter les pixels groupe par groupe :

Juste avant le code visible, il y a une boucle qui parcourt tous les pixels de l'image voulue. En version séquentielle les pixels sont traités un par un, et groupe par groupe dans la version vectorielle. Les variables cr, ci, zr et zi sont calculées précédemment dans la boucle.

**Séquentiel**

```
for (n=0; n<=maxiter
     && m<BAIL_OUT*BAIL_OUT; n++)
{
   float a=zr*zr-zi*zi+cr;
   float b=zr*(zi+zi)+ci;
   zr=a;
   zi=b;
   m=a*a+b*b;
}
pixels[y*WIDTH + x] = n;
```

**Vectoriel**

```
for (n = 0; n <= maxiter && popc(vmask)>0; n
{
   vtemp=mul(vzr, vzr,vmask);// zr^2
   va=mul(vzi, vzi, vmask, va);//zi^2
   va=sub(vtemp, va,vmask, va);//zr^2-zi^2
   va=add(va, vcr,vmask, va);//zr^2-zi^2+cr
   //FLOAT a=zr*zr-zi*zi+cr;

   vb=mul(vzr, vzi, vmask, vb);// zr*zi
   vb=mul(vb, 2.0, vmask, vb);// 2*zr*zi
   vb=add(vb, vci, vmask, vb);// 2*zr*zi+ci
   //FLOAT b=zr*(zi+zi)+ci;

   mov(vzr,va);//zr=a;
   mov(vzi,vb);//zi=b;
   //zr=a;
   //zi=b;

   vtemp=mul(va, va, vmask, vm);// a^2
   vm=mul(vb, vb, vmask, vm);//b^2
   vm=add(vm, vtemp, vmask, vm);//a^2+b^2
   //m=a*a+b*b;

   vn=add(vn, 1, vmask, vn);
   vmask=cmplt(vm, BAIL_OUT32);
}
store(Tn, vn);
```

TABLE 3 – Algorithme de génération du Mandelbrot en version séquentielle et vectorielle

L'idée du code vectoriel est d'utiliser un masque qui permet de retenir les pixels dans le groupe de pixels qui ont terminé ou non leur algorithme en comparant le vecteur des normes des pixels à un vecteur contenant la norme à ne pas dépasser. Un autre registre vectoriel retient le nombre d'itérations effectuées.

### 2.5.4 Efficacité