

付録2. 浮動小数点命令

Table of Contents

[ISA] 浮動小数点数について	2
[RISCV] RISC-Vの浮動小数点命令サポートについて	2
[ISA] 浮動小数点数フォーマット	2
例 10進数3.14を単精度浮動小数点数で表現する	3
[ISA] 浮動小数点数の正規化と桁合わせ	4
[ISA] 浮動小数点数の加減算	5
[ISA] 浮動小数点数の乗算	6
[RISCV] RISC-Vの浮動小数点命令について	7
[RISCV] 浮動小数点命令のためのレジスタ定義	7
[RISCV] 浮動小数点算術演算命令	8
浮動小数点四則演算命令	8
浮動小数点符号操作命令	9
浮動小数点積和演算命令	9
浮動小数点平方根演算命令	10
浮動小数点最大値・最小値取得命令	10
浮動小数点変換命令	10
浮動小数点比較・分類命令	11
浮動小数点レジスタのためのロードストア命令	11
[RISCV] 浮動小数点命令のエンコーディング	12
浮動小数点命令をLLVMに実装する	13
[LLVM] 浮動小数点レジスタを定義	14
[LLVM] 浮動小数点命令の追加：ロード・ストア命令	17
[LLVM] 浮動小数点算術演算命令の追加	19
[LLVM] 浮動小数点レジスタ向けのCalling Conventionの追加	24
[LLVM] さまざまな浮動小数点命令の追加	27
浮動小数点比較命令	27
浮動小数点数のその他の命令	30
まとめとレッスン：浮動小数点演算を用いたマンデルブロ集合の描画	34
簡単に画像を出力するためのフォーマット: PGM	36
マンデルブロ曲線描画プログラムをLLVMでコンパイルしてシミュレーション	36
さらに一步先へ：浮動小数点演算を用いたレイトレースプログラムの実行	38

これまで整数データおよび整数演算に焦点を当てて命令セットアーキテクチャを調査してきました。しかしコンピュータは整数だけでなく、小数点を含む実数も扱う必要があります。これらを扱うために、現代のコンピュータでは浮動小数点数 (**floating point number**) が一般的に使われています。浮動小数点数を扱うため

に、各命令セットアーキテクチャではさまざまな仕様が定義されています。浮動小数点数を格納するためのレジスタ、演算命令、メモリにアクセス命令、また整数レジスターとの間で情報交換をするための命令など… 本章では、これらの浮動小数点数の機能について概観しながら、LLVMに浮動小数点命令を追加するための方法について見ていきます。

[ISA] 浮動小数点数について

コンピュータは整数だけでなく、小数点の入ったデータを取り扱う必要があります。一般的にこのような小数点の入ったデータをコンピュータで扱うには2つの表現形式があります。1つは固定小数点数 (**fixed point number**) で、もう一つは浮動小数点数 (**floating point number**) です。名前から想像できる通り、「小数点」（いわゆる3.14などの小数点で「.」）の位置が固定なのか変化するのかで仕様の違いがあります。固定小数点数は浮動小数点数よりも処理が簡単ですが、表現できる数値の範囲は浮動小数点数よりも小さくなります。浮動小数点数は固定小数点数ほどフォーマットが簡単ではありませんが、より多くの値をより高い精度で保持できます。

本章では浮動小数点数に焦点を当てて解説しますが、浮動小数点数の使用自体は非常に複雑です。IEEE 754という規格によって明確に定義されており、サポートするビット長に応じて以下の種類が定義されています。

- 半精度 (**half precision**) : 16ビット長
- 単精度 (**single precision**) : 32ビット長
- 倍精度 (**double precision**) : 64ビット長
- 4倍精度 (**Quadruple precision**) : 128ビット長

さらにIEEE 754では10進数浮動小数点数という形式も定義されていますが、こちらは上記の4種類の形式に比べてあまり一般的ではないのでここでは省略します。

[RISCV] RISC-Vの浮動小数点命令サポートについて

さて、多くの命令セットアーキテクチャでは、この浮動小数点数を扱うための仕様が定義されています。RISC-Vにも浮動小数点数を扱うための仕様が定義されています。

- F拡張：単精度浮動小数点数をサポートするための命令グループ
- D拡張：倍精度浮動小数点数をサポートするための命令グループ
- Q拡張：4倍精度浮動小数点数をサポートするための命令グループ

たとえばRISC-Vのサポート仕様について、`RV64IMAFD`という記述ならば、単精度浮動小数点命令と倍精度浮動小数点命令がサポートされていることを意味します^[1]。

[ISA] 浮動小数点数フォーマット

IEEE 754の浮動小数点数フォーマットには16ビット長から128ビット長までの仕様が定義されていますが、どの仕様も基本的なデータフォーマットは変わりません。Figure 1に、半精度から4倍精度までの浮動小数点数の形式を示します。どの形式も

- 符号ビット (S)
- 指数部ビット (E)
- 仮数部ビット (F)

で構成されます。符号は+/-を表現するだけなので1ビットで良いのですが、指数部と仮数部のビット長は形式によって異なります。しかしそのすべてのフォーマットにおいてこのビット列で表す浮動小数点数は以下で表現されます。

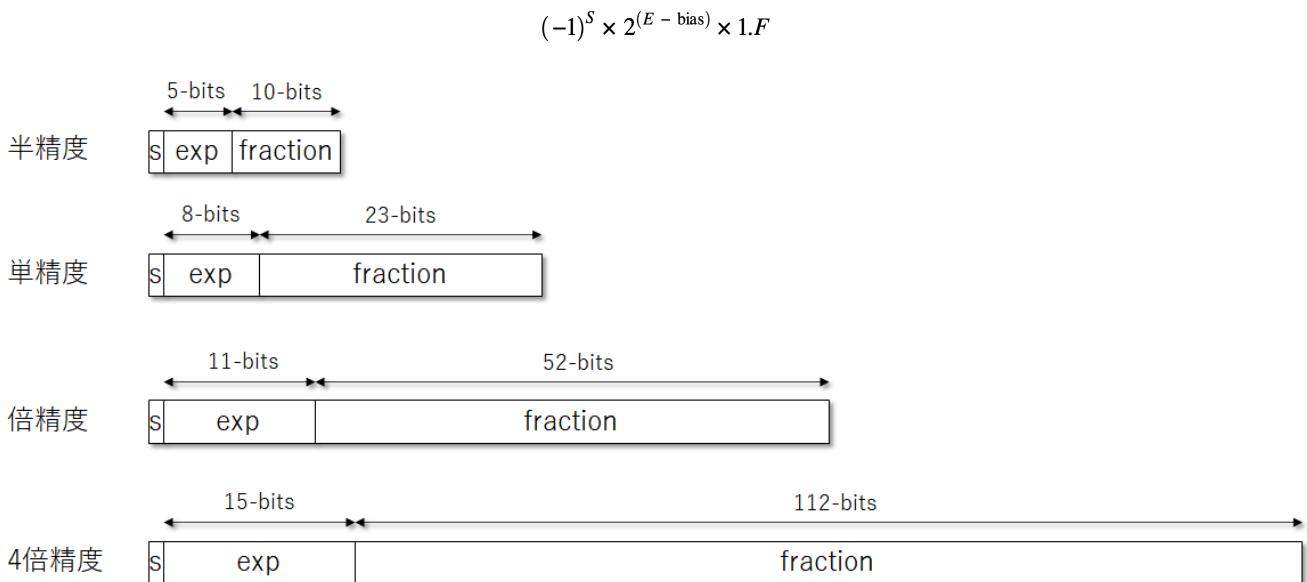


Figure 1. IEEE 754で定める浮動小数点数のフォーマット。半精度から4倍精度までのフォーマットが定義されている。Sは符号部、expは指数部、fractionは仮数部である。ビット長が長くなるほど表現できる実数の範囲が広く、精度が高い。

最初の $(-1)^S$ は、S=0であれば正の数、S=1であれば負の数であることを示します。次の $2^{(E - \text{bias})}$ は指数部 Eを使って表現されます。 bias は各フォーマットで異なる値が定義されており、たとえば单精度の場合は127、倍精度の場合は1023となります。つまり、Eは常に正の整数として扱われ、たとえば单精度の場合は

- $E < 127$: $2^{(E - \text{bias})}$ は1よりも小さな値となり、1よりも小さな値を表現する際に使用されます。
- $E \geq 127$: $2^{(E - \text{bias})}$ は1以上となり、1以上の値を表現する際に使用されます。

最後のFは仮数部と呼ばれ、浮動小数点数の細かな値を表現します。仮数部ビット長は、单精度より倍精度、倍精度よりも4倍精度の方が大きなビット幅を確保されており、より細かな値を表現できるというわけです。

例 10進数3.14を单精度浮動小数点数で表現する

10進数の3.14を上記の浮動小数点フォーマットに変換すると、 $(-1)^0 \times 2^{(128 - 127)} \times 1.57$ となります。したがって

- 符号部：0
- 指数部：128、これは2進数ビット列で表現すると1000 0000となります。
- 仮数部：0.57、これを2進数でどのように表現するかということですが、これまでの2進数の表現では、

- 0桁目 : $2^0 = 1$
- 1桁目 : $2^1 = 2$
- 2桁目 : $2^2 = 4$
- として表現しています。同様に、小数点以下を表現するために、
- -1桁目 : $2^{-1} = 0.5$
- -2桁目 : $2^{-2} = 0.25$
- -3桁目 : $2^{-3} = 0.125$
- と考えることができます。これを0.57という数値に対して適用すると
と $0.10010001111010111000011\dots$ となります。このように、0.57という数値は2進数の有限ビット列で表現することは出来ません。単精度のビット長で表現できる限界まで打ち切るので、2進数で3.14を表現した場合は誤差が発生しています。

よって、単精度フォーマットに適用すると以下のようになります。

0	1000 0000	100 1000 1111 0101 1100 0011
---	-----------	------------------------------

これを16進数で表記すると、0x4048f5c3となります。

[ISA] 浮動小数点数の正規化と桁合わせ

上記の浮動小数点数フォーマットで見たように、浮動小数点数の表現形式は $(-1)^S \times 2^{(E - \text{bias})} \times 1.F$ で表現されます。
ここで、仮数部に当たる1.Fの部分は、必ず1で始まるように決められており、これを正規化(normalize)されていると言います。なぜこのような規則になっているのかというと、上記の3.14を浮動小数点数で表現する場合、正規化されていないと以下のような表現も可能になります。

- $(-1)^0 \times 2^{(129 - 127)} \times 0.157$
- $(-1)^0 \times 2^{(130 - 127)} \times 0.0157$

指数部の値を増やして仮数部の先頭に0を並べることで、同じ3.14でもいくつかの表現が可能になります。IEEE 754の規則では、このような表現は許されません。レジスタに格納されている数値や、浮動小数点数の算術演算結果は常に正規化されてレジスタに格納される必要があります。そうでなければ仮数部に無駄なゼロが並んでしまい表現範囲が狭くなってしまうため、浮動小数点数の表現としては常に正規化されることが求められています^[2]。

一方で、以降で説明する算術演算については、正規化した状態では直接計算できません。たとえば浮動小数点数において $0.1 + 100.0$ を実行しようとすると、それぞれの単精度浮動小数点数での正規化された表現は、

- $0.1 = (-1)^0 \times 2^{(123 - 127)} \times 1.600 = 2^{-4} \times 1.600$
- $100.0 = (-1)^0 \times 2^{(133 - 127)} \times 1.5625 = 2^6 \times 1.5625$

ですが、これを単純に加算しようとすると、 $0.1 + 100.0 = 2^{-4} \times 1.600 + 2^6 \times 1.5625$ となり指数部が異なるためそのまま加算できません。したがって $2^{-4} \times 1.600 + 2^6 \times 1.5625 = 2^6 \times 0.0015625 + 2^6 \times 1.5625 = 2^6 \times 1.5640625 = 100.1$ と、指数部を合わせる代わりに仮数部をずらして計算します。一時的にではあります但このように浮動小

数点数を正規化数から変換し指数部の桁合わせを行うことで、浮動小数点数の加減算を実現しています。浮動小数点数の加減算・乗算の具体的な方法については、以下で説明します。

0.1	100.0
0 01111011 10011001100110011001101	+ 0 10000101 10010000000000000000000000000000
0 123-127=-4 (1.)600	+ 0 133-127=6 (1.)5625
0 6 0.0015625	+ 0 6 (1.)5625

Figure 2. 浮動小数点数の正規化表現と桁合わせ。浮動小数点数は通常時は正規化されて格納されているが、計算時には指数部同士を桁合わせして計算する。

[ISA] 浮動小数点数の加減算

浮動小数点数フォーマットについて定義を紹介しましたが、このフォーマットを使ってどのように演算するかを簡単に紹介します。算術演算の基本として四則演算、加減算と乗算除算について簡単に見ていきましょう。本書は命令セットアーキテクチャとコンパイラについて解説する本ですので、ハードウェアを使って浮動小数点数の算術演算をどのように実装するかについてまでは言及しません。コンパイラから見れば、以降の章で紹介する浮動小数点数の演算命令を実行すれば加減算はハードウェアが勝手に行ってくれるので詳細を知っておくことは必ずしも必要ありません。しかし考え方自体は情報科学の基本的な知識に当たりますので、簡単に紹介しておくことにします。

前節で紹介したとおり、浮動小数点数フォーマットは以下の数式を使って表現できます。

$(-1)^S \times 2^{(E - \text{bias})} \times 1.F$ (ただし S=符号部、E=指数部、F=仮数部) 単精度浮動小数点数の場合 bias は 127、そしてそれ以外の要素 S, E, F によって値が一意に決定されます。

では、 S_1, E_1, F_1 と S_2, E_2, F_2 の要素から構成される 2 つの浮動小数点数同士を加算するためには、どのようにすればよいでしょうか。

- $\text{Val}_1 = (-1)^{S_1} \times 2^{(E_1 - \text{bias})} \times 1.F_1$
- $\text{Val}_2 = (-1)^{S_2} \times 2^{(E_2 - \text{bias})} \times 1.F_2$

まず符号部ですが、 S_1 と S_2 に応じて仮数部・指数部に対する演算が変わってきます。

- Val_1 と Val_2 が同じ符号であれば、加算後も符号は同じですので単純に加算します。
- Val_1 と Val_2 の符号が異なる場合、絶対値を減算する必要があります、お互いの値の大きさによって計算後の符号は変わります。

問題を簡単にするためここでは Val_1 と Val_2 が同一の符号であることを考えます。

まず最初に考えなければならないのは、 Val_1 と Val_2 は指数部の値です。

E_1 と E_2 は異なる可能性があり、桁合わせができていません。 小学校で学ぶ筆算でもそうですが、桁が合っていないと計算ができないので、最初に指数部の桁合わせを行います。 ここでは符号が同一 $S_1 = S_2 = 0$ で、 $E_2 > E_1$ とすると、

$$\text{Val}_1 + \text{Val}_2$$

$$= (-1)^{S_1} \times 2^{(E_1 - \text{bias})} \times 1.F_1 + (-1)^{S_2} \times 2^{(E_2 - \text{bias})} \times 1.F_2^*$$

$$= 2^{(E_1 - \text{bias})} \times 1.F_1 + 2^{(E_2 - \text{bias})} \times 1.F_2^*$$

$$= 2^{(E_1 - \text{bias})} \times 1.F_1 + 2^{((E_2 - E_1) + (E_1 - \text{bias}))} \times 1.F_2^*$$

$$= 2^{(E_1 - \text{bias})} \times 1.F_1 + 2^{(E_2 - E_1)} \times 2^{E_1 - \text{bias}} \times 1.F_2^*$$

$$= (1.F_1 + 2^{E_2 - E_1} \times 1.F_2) \times 2^{(E_1 - \text{bias})}$$

このように

$2^{(E_1 - \text{bias})}$ でまとめ上げることができ、 $2^{E_2 - E_1} \times 1.F_2$ は $1.F_2$ を $E_2 - E_1$ 分左シフトすることを意味しますから、左シフト後に $1.F_1$ 加算します。最後に計算後の値を正規化して演算は終了です。

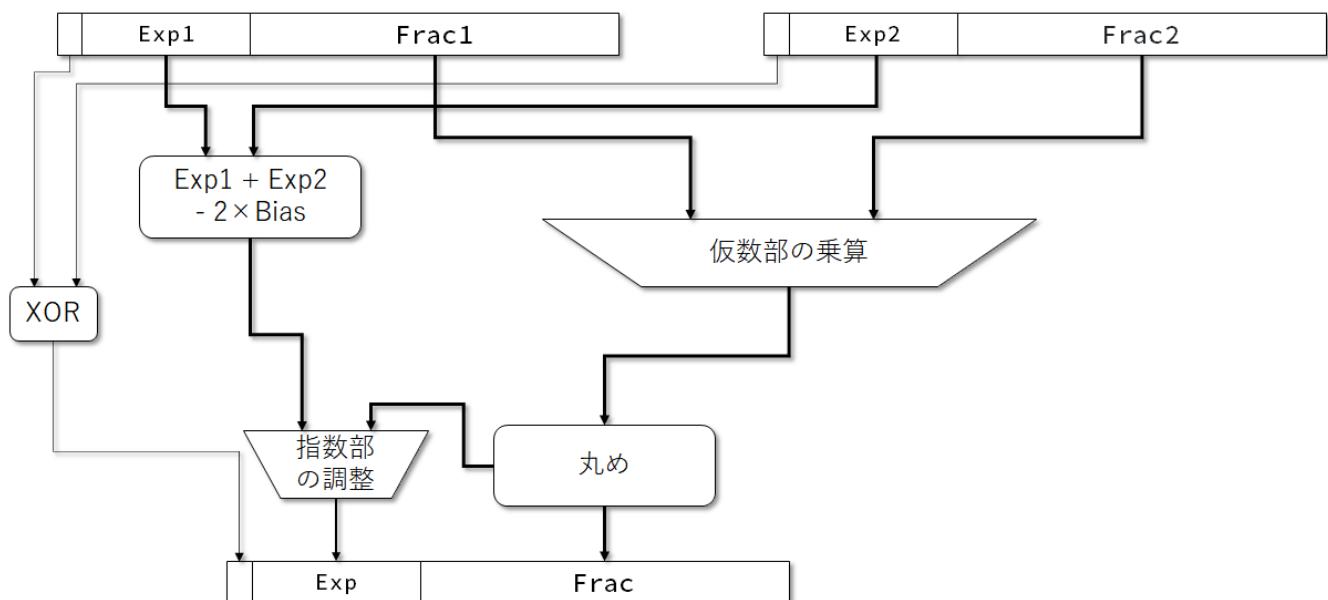


Figure 3. 浮動小数点加算回路の構成概略。指数部同士の調整を行い桁合わせを行い加算部の加算を実行する

[ISA] 浮動小数点数の乗算

次は浮動小数点数での乗算について考えます。加算に比べて乗算は少し単純です。浮動小数点数のフォーマットは

$$(-1)^S \times 2^{(E - \text{bias})} \times 1.F$$

なので、以下の2つの値 Val_1 と Val_2 を乗算する場合、

- $\text{Val}_1 = (-1)^{S_1} \times 2^{(E_1 - \text{bias})} \times 1.F_1$

- $\text{Val}_2 = (-1)^{S_2} \times 2^{(E_2 - \text{bias})} \times 1.F_2$

計算結果は以下のようになります。

$$\text{Val}_1 \times \text{Val}_2 = ((-1)^{S_1} \times (-1)^{S_2}) \times 2^{(E_1 - \text{bias})} \times 2^{(E_2 - \text{bias})} \times 1.F_1 \times 1.F_2^*$$

$$= ((-1)^{S_1 + S_2}) \times 2^{(E_1 - \text{bias} + E_2 - \text{bias})} \times 1.F_1 \times 1.F_2^*$$

$$= ((-1)^{S_1 + S_2}) \times 2^{(E_1 + E_2 - 2 \times \text{bias})} \times 1.F_1 \times 1.F_2$$

と計算できます。つまり、指数部同士を加算して、最上位ビットを付加した仮数部同士の乗算を行えばよいことになります。このように、加算とは異なり乗算では指数部同士の桁合わせが必要ありません。

仮数部同士の乗算の結果、この値が正規化されているとは限らないので、正規化処理を行ったうえで値を表現可能なビット数に収めるための丸め処理を行います。

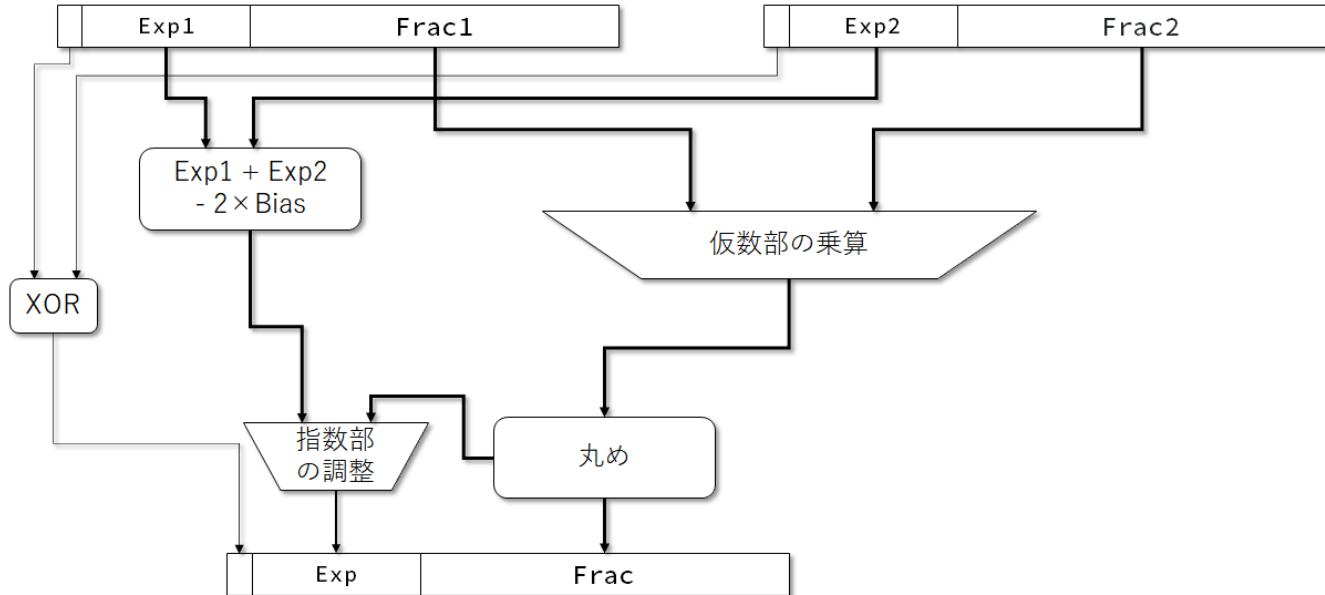


Figure 4. 浮動小数点乗算回路の構成概略。指数部はバイアスを除いて乗算を行い、仮数部同士も乗算を行う。最後に丸めを行い正規化する。

[RISCV] RISC-Vの浮動小数点命令について

多くの命令セットアーキテクチャと同様に、RISC-Vにも浮動小数点演算の命令が定義されています。RISC-Vには「単精度浮動小数点命令（F）」「倍精度浮動小数点命令（D）」「4倍精度浮動小数点命令（Q）」が定義されています。4倍精度まで命令として定義されているのは非常に珍しく、先を見越したアーキテクチャだということができます。ここでは単精度浮動小数点命令のためのF拡張と、倍精度浮動小数点命令D拡張について簡単に紹介します。

[RISCV] 浮動小数点命令のためのレジスタ定義

RISC-Vでは、整数命令のためのレジスタとは別に浮動小数点値数を格納するためのレジスタが定義されています。浮動小数点命令のオペランドとして、接頭語”f”を付加し、整数レジスタと同様にインデックスで表現します。レジスタは32本定義されており、f0からf31までのレジスタを使用できます。

RISC-Vでは単精度と倍精度は別の拡張として取り扱われていますが、レジスタは共有です。アーキテクチャ仕様がRV64IMAFDCである場合、浮動小数点命令はF拡張のみなのでレジスタは32ビットですが、RV64IMAFDCとなると倍精度の仕様も追加されるので浮動小数点レジスタのサイズは64ビットまで拡張されます。単精度命令の場合は下位の32ビット、倍精度命令の場合は64ビット全体を使用することになります。どちらの場合も、オペランドはf0～f31の名称で指定され、オペコードの演算タイプによって倍精度か単精度を決定するようになっています。

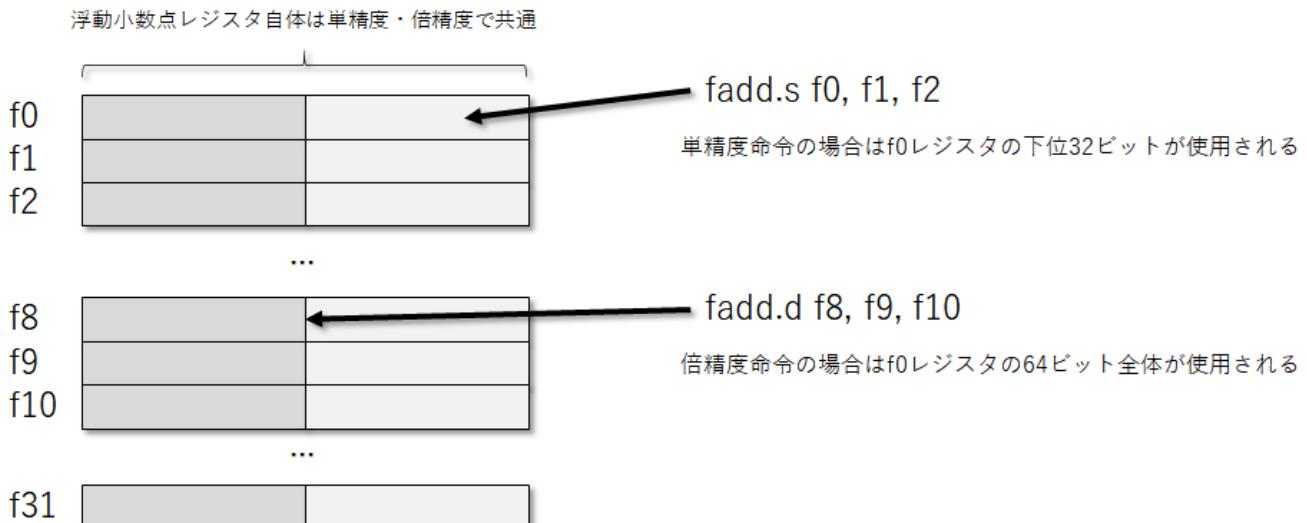


Figure 5. RISC-Vの浮動小数点レジスタの定義。単精度命令・倍精度命令で使用するレジスタは共通。単精度命令ではレジスタの下位32ビットを使用し、倍精度命令ではレジスタの64ビット全体を使用する。

[RISCV] 浮動小数点算術演算命令

RISC-Vでは、単精度・倍精度において以下の浮動小数点命令が定義されています。

- 四則演算（加算・減算・乗算・除算）
- 符号操作のための命令
- 平方根
- 最大値・最小値
- 整数值との変換、整数レジスタとの移動
- 比較・クラス分類
- （倍精度のみ）単精度値との変換

浮動小数点四則演算命令

RISC-Vでは单精度・倍精度の浮動小数点加算・減算・乗算・除算命令が定義されています。浮動小数点レジスタ2つを入力オペランドとして受け取り、演算後その結果を浮動小数点レジスタに格納します。

```

/// 四則演算命令
// 単精度命令
FADD.S    fd,fs1,fs2  // freq[fd] = freq[fs1] + freq[fs2]
FSUB.S    fd,fs1,fs2  // freq[fd] = freq[fs1] - freq[fs2]
FMUL.S    fd,fs1,fs2  // freq[fd] = freq[fs1] * freq[fs2]
FDIV.S    fd,fs1,fs2  // freq[fd] = freq[fs1] / freq[fs2]
// 倍精度命令
FADD.D    fd,fs1,fs2  // freq[fd] = freq[fs1] + freq[fs2]
FSUB.D    fd,fs1,fs2  // freq[fd] = freq[fs1] - freq[fs2]
FMUL.D    fd,fs1,fs2  // freq[fd] = freq[fs1] * freq[fs2]
FDIV.D    fd,fs1,fs2  // freq[fd] = freq[fs1] / freq[fs2]

```

浮動小数点符号操作命令

RISC-Vの浮動小数点命令には、符号を操作するための命令が用意されています。これらの命令は、主に浮動小数点データの符号反転や絶対値などを取得するために利用されます。 **FSGNJ** および **FSGNJP** 命令は、符号ビット以外は **fs1** レジスタから取得しますが、符号ビットのみ **fs2** から取得します。 **FSGNJPX** も **fs1** レジスタから取得しますが、符号ビットは **fs1** と **fs2** の排他的論理和となります。

```
/// 符号操作のための命令
// 単精度命令
FSGNJ.S    fd,fs1,fs2    // freq[fd] = {符号部: freq[fs2], 指数部・仮数部: freq[fs1]}
FSGNJP.S   fd,fs1,fs2    // freq[fd] = {符号部: !freq[fs2], 指数部・仮数部: freq[fs1]}
FSGNJPX.S  fd,fs1,fs2   // freq[fd] = {符号部: freq[fs1]^freq[fs2],
                                         指数部・仮数部: freq[fs1]}
// 倍精度命令
FSGNJ.D    fd,fs1,fs2    // freq[fd] = {符号部: freq[fs2], 指数部・仮数部: freq[fs1]}
FSGNJP.D   fd,fs1,fs2    // freq[fd] = {符号部: !freq[fs2], 指数部・仮数部: freq[fs1]}
FSGNJPX.D  fd,fs1,fs2   // freq[fd] = {符号部: freq[fs1]^freq[fs2],
                                         指数部・仮数部: freq[fs1]}
```

多くの場合、これらの符号操作命令は疑似命令として利用されます。

```
FMV.{S,D}   fd,fs      // FSGNJ.{S,D}  fd,fs,fs 浮動小数点レジスタ移動命令
FABS.{S,D}   fd,fs      // FSGNJPX.{S,D} fd,fs,fs 浮動小数点絶対値取得命令
FNEG.{S,D}   fd,fs      // FSGNJPN.{S,D} fd,fs,fs 浮動小数点符号反転命令
```

浮動小数点積和演算命令

浮動小数点積和演算命令は、3つの浮動小数点レジスタA,B,Cを受け取り $A \times B + C$ をはじめとするバリエーションを計算する命令です。乗算命令と加算命令を組み合わせる場合と比較して、命令数を削減できることと、乗算の結果を丸めることなく加算するので精度が落ちないというメリットがあります。

この積和演算はさまざまなアプリケーションで使用され、たとえば行列積の計算などでも積和演算は多用されます。このため、この命令の性能が浮動小数点演算の性能に大きく影響します。

```
/// 融合積和演算 Fused Multiply-Add
// 単精度命令
FMADD.S    fd,fs1,fs2,fs3  // freq[fd] = (freq[fs1] * freq[fs2]) + freq[fs3]
FMSUB.S    fd,fs1,fs2,fs3  // freq[fd] = (freq[fs1] * freq[fs2]) - freq[fs3]
FNMSUB.S   fd,fs1,fs2,fs3 // freq[fd] = -(freq[fs1] * freq[fs2]) + freq[fs3]
FNMADD.S   fd,fs1,fs2,fs3 // freq[fd] = -(freq[fs1] * freq[fs2]) - freq[fs3]
// 倍精度命令
FMADD.D    fd,fs1,fs2,fs3  // freq[fd] = (freq[fs1] * freq[fs2]) + freq[fs3]
FMSUB.D    fd,fs1,fs2,fs3  // freq[fd] = (freq[fs1] * freq[fs2]) - freq[fs3]
FNMSUB.D   fd,fs1,fs2,fs3 // freq[fd] = -(freq[fs1] * freq[fs2]) + freq[fs3]
FNMADD.D   fd,fs1,fs2,fs3 // freq[fd] = -(freq[fs1] * freq[fs2]) - freq[fs3]
```

浮動小数点平方根演算命令

浮動小数点レジスタ1つを受け取り、その平方根を計算してその結果を浮動小数点レジスタに格納します。

```
/// 平方根
// 単精度命令
FSQRT.S    fd,fs1           // freq[fd] = sqrtf(freq[fs1])
// 倍精度命令
FSQRT.D    fd,fs1           // freq[fd] = sqrnd(freq[fs1])
```

浮動小数点最大値・最小値取得命令

2つの浮動小数点レジスタから値を読み込み、比較して最大値・最小値を取得し浮動小数点レジスタに格納します。

```
/// 最大値・最小値
// 単精度命令
FMIN.S    fd,fs1,fs2        // freq[fd] = min(freq[fs1], freq[fs2])
FMAX.S    fd,fs1,fs2        // freq[fd] = max(freq[fs1], freq[fs2])
// 倍精度命令
FMIN.D    fd,fs1,fs2        // freq[fd] = min(freq[fs1], freq[fs2])
FMAX.D    fd,fs1,fs2        // freq[fd] = max(freq[fs1], freq[fs2])
```

浮動小数点変換命令

浮動小数点データと整数データ、あるいは浮動小数点数同士を変換します。

- 整数データ（整数レジスタ） \leftrightarrow 浮動小数点データ（浮動小数点レジスタ）
- 浮動小数点データ（浮動小数点レジスタ） \leftrightarrow 浮動小数点データ（浮動小数点レジスタ）

の2種類があります。命令のフォーマットは一貫しており、

FCVT.{変換後の型}.{変換前の型} というフォーマットです。型の表記としては **W**（32ビット整数）、**WU**（32ビット符号なし整数）、**L**（64ビット整数）、**LU**（64ビット符号なし整数）、**S**（単精度浮動小数点）、**D**（倍精度浮動小数点）の表記があります。たとえば、**FCVT.W.S** は **S**（単精度浮動小数点）から **W**（32ビット整数）へと変換することを意味します。このとき、入力データは浮動小数点レジスタから取得し、出力データは整数レジスタに格納されます。

これ以外に、型を変換せずに値を移動するだけの命令も存在します。**FMV** 命令は浮動小数点レジスタと整数レジスタの間で、型変換をせずにデータの移動のみ行います。

```
/// 整数値との変換・整数レジスタとの移動
// 単精度命令
FCVT.W.S  rd,fs1           // reg[rd] = Convert_Single_to_Int32(freq[fs1])
FCVT.WU.S  rd,fs1          // reg[rd] = Convert_Single_to_UInt32(freq[fs1])
FCVT.S.W   fd,rs1           // freq[fd] = Convert_Int32_to_Single(rs1)
```

```

FCVT.S.WU fd,rs1      // freq[fd] = Convert_UInt32_to_Single(rs1)

FMV.X.W   rd,fs1      // reg[rd] = freq[fs1]
FMV.W.X   fd,rs1      // freq[fd] = reg[rs1]

// 倍精度命令
FCVT.W.D  rd,fs1      // reg[rd] = Convert_Double_to_Int32(freq[fs1])
FCVT.W.U.D rd,fs1      // reg[rd] = Convert_Double_to_UInt32(freq[fs1])
FCVT.D.W   fd,rs1      // freq[fd] = Convert_Int32_to_Double(rs1)
FCVT.D.WU  fd,rs1      // freq[fd] = Convert_UInt32_to_Double(rs1)

FMV.X.D   rd,fs1      // reg[rd] = freq[fs1]
FMV.D.X   fd,rs1      // freq[fd] = reg[rs1]

FCVT.L.D   rd,fs1      // reg[rd] = Convert_Double_to_Int64(freq[fs1])
FCVT.L.U.D rd,fs1      // reg[rd] = Convert_Double_to_UInt64(freq[fs1])
FCVT.D.L   fs1,rd      // freq[fd] = Convert_Int64_to_Double(rs1)
FCVT.D.LU  fs1,rd      // freq[fd] = Convert_UInt64_to_Double(rs1)

FCVT.S.D   fd,fs1      // freq[fd] = Convert_Double_to_Single(fs1)
FCVT.D.S   fd,fs1      // freq[fd] = Convert_Single_to_Double(fs1)

```

浮動小数点比較・分類命令

RISC-Vの浮動小数点比較命令では、2つの浮動小数点レジスタを受け取り、大小比較などの比較演算を行いその結果を整数レジスタに格納します。また、**FCLASS** 命令は浮動小数点オペランドを受け取り、その値がどのような値であるのか（-無限大、-サブノーマル数、+-正規化数、ゼロ、など）を判定してそれに対応した値を整数レジスタに格納します。

```

/// 比較・クラス分類
// 単精度命令
FEQ.S    rd,fs1,fs2  // reg[rd] = freq[fs1] == freq[fs2]
FLT.S    rd,fs1,fs2  // reg[rd] = freq[fs1] < freq[fs2]
FLE.S    rd,fs1,fs2  // reg[rd] = freq[fs1] <= freq[fs2]
FCLASS.S rd,fs1      // reg[fd] = class(freq[fs1])

// 倍精度命令
FEQ.D    rd,fs1,fs2  // reg[rd] = freq[fs1] == freq[fs2]
FLT.D    rd,fs1,fs2  // reg[rd] = freq[fs1] < freq[fs2]
FLE.D    rd,fs1,fs2  // reg[rd] = freq[fs1] <= freq[fs2]
FCLASS.D rd,fs1      // reg[fd] = class(freq[fs1])

```

浮動小数点レジスタのためのロードストア命令

浮動小数点レジスタの値とメモリの間でデータ転送するためのロードストア命令が定義されています。これらは整数レジスタで定義されているメモリアクセス命令とほぼ同一です。メモリアドレスは整数レジスタの値とオフセットの加算で算出され、ロード命令はメモリの内容を浮動小数点レジスタに格納し、ストア命令は浮動小数点レジスタの値をメモリに書き込みます。

```

/// メモリアクセス命令
// 単精度メモリアクセス命令
FLW    fd,offset(rs1)      // freq[fd] = Memory(reg[rs1] + offset)
FSW    fs2,offset(rs1)      // Memory(reg[rs1] + offset) = freq[fs2]
// 倍精度メモリアクセス命令
FLD    fd,offset(rs1)      // freq[fd] = Memory(reg[rs1] + offset)
FSD    fs2,offset(rs1)      // Memory(reg[rs1] + offset) = freq[fs2]

```

[RISCV] 浮動小数点命令のエンコーディング

これらの浮動小数点演算命令のエンコーディングは表[tb:RVF_table]、表link:RISC-V倍精度浮動小数点命令（RV32D%20/%20RV64D）の命令エンコーディング Figure 6, Figure 7 のようになっています。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FLW																		rs1	0	1	0		rd	0	0	0	0	1	1	1		
FSW																		rs1	0	1	0		imm[4:0]	0	1	0	0	1	1	1		
FMADD.S						rs3	0	0										rs2		rs1		rm	rd	1	0	0	0	0	1	1	1	
FMSUB.S						rs3	0	0										rs2		rs1		rm	rd	1	0	0	0	1	1	1	1	
FMMSUB.S						rs3	0	0										rs2		rs1		rm	rd	1	0	0	1	0	1	1	1	
FMADD.D						rs3	0	0										rs2		rs1		rm	rd	1	0	0	1	1	1	1	1	
FADD.S	0	0	0	0	0	0	0	0										rs2		rs1		rm	rd	1	0	1	0	0	1	1	1	
FSUB.S	0	0	0	0	1	0	0	0										rs2		rs1		rm	rd	1	0	1	0	0	1	1	1	
FMUL.S	0	0	0	1	0	0	0	0										rs2		rs1		rm	rd	1	0	1	0	0	1	1	1	
FDIV.S	0	0	0	1	1	0	0											rs2		rs1		rm	rd	1	0	1	0	0	1	1	1	
FSQRT.S	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1		
FSGNJ.S	0	0	1	0	0	0	0	0										rs2		rs1	0	0	0	rd	1	0	1	0	0	0	1	1
FSGNJN.S	0	0	1	0	0	0	0	0										rs2		rs1	0	0	1	rd	1	0	1	0	0	0	1	1
FSGNXJ.S	0	0	1	0	0	0	0	0										rs2		rs1	0	1	0	rd	1	0	1	0	0	0	1	1
FMIN.S	0	0	1	0	1	0	0											rs2		rs1	0	0	0	rd	1	0	1	0	0	0	1	1
FMAX.S	0	0	1	0	1	0	0											rs2		rs1	0	0	1	rd	1	0	1	0	0	0	1	1
FCVT.W.S	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1		
FCVT.WU.S	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	1	1		
FMV.X.W	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1		
FEQ.S	1	0	1	0	0	0	0										rs2		rs1	0	1	0	rd	1	0	1	0	0	0	1	1	
FLT.S	1	0	1	0	0	0	0										rs2		rs1	0	0	1	rd	1	0	1	0	0	0	1	1	
FLE.S	1	0	1	0	0	0	0										rs2		rs1	0	0	0	rd	1	0	1	0	0	0	1	1	
FCLASS.S	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1		
FCVT.S.W	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1		
FCVT.S.WU	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	1		
FMV.W.X	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1		
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.L.S	1	1	0	0	0	0	0	0	0	0	0	1	0				rs1				rm	rd	1	0	1	0	0	1	1			
FCVT.LU.S	1	1	0	0	0	0	0	0	0	0	0	1	1				rs1				rm	rd	1	0	1	0	0	1	1			
FCVT.S.L	1	1	0	1	0	0	0	0	0	0	0	1	0				rs1				rm	rd	1	0	1	0	0	0	1	1		
FCVT.S.LU	1	1	0	1	0	0	0	0	0	0	0	1	1				rs1				rm	rd	1	0	1	0	0	0	1	1		

Figure 6. RISC-V単精度浮動小数点命令（RV32F / RV64F）の命令エンコーディング

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FLD																		rs1	0	1	0		rd	0	0	0	0	1	1	1		
FSD																		rs1	0	1	0		imm[4:0]	0	1	0	0	1	1	1		
FMADD.D						rs3	0	1									rs2		rs1		rm		rd	1	0	0	0	0	1	1	1	
FMSUB.D						rs3	0	1									rs2		rs1		rm		rd	1	0	0	0	1	1	1	1	
FMSUB.D						rs3	0	1								rs2		rs1		rm		rd	1	0	0	1	0	1	1	1		
FMADD.D						rs3	0	1								rs2		rs1		rm		rd	1	0	0	1	1	1	1	1		
FADD.D	0	0	0	0	0	0	1									rs2		rs1		rm		rd	1	0	1	0	0	1	1	1		
FSUB.D	0	0	0	0	1	0	1									rs2		rs1		rm		rd	1	0	1	0	0	1	1	1		
FMUL.D	0	0	0	1	0	0	1									rs2		rs1		rm		rd	1	0	1	0	0	1	1	1		
FDIV.D	0	0	0	1	1	0	1									rs2		rs1		rm		rd	1	0	1	0	0	1	1	1		
FSQRT.D	0	1	0	1	1	0	1	0	0	0	0	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FSGNJ.D	0	0	1	0	0	0	1									rs2		rs1	0	0	0	rd	1	0	1	0	0	1	1	1		
FSGNJN.D	0	0	1	0	0	0	1									rs2		rs1	0	0	1	rd	1	0	1	0	0	1	1	1		
FSGNX.J.D	0	0	1	0	0	0	1									rs2		rs1	0	1	0	rd	1	0	1	0	0	1	1	1		
FMIN.D	0	0	1	0	1	0	1									rs2		rs1	0	0	0	rd	1	0	1	0	0	1	1	1		
FMAX.D	0	0	1	0	1	0	1									rs2		rs1	0	0	1	rd	1	0	1	0	0	1	1	1		
FCVT.S.D	1	1	0	0	0	0	1	0	0	0	0	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FCVT.D.S	1	1	0	0	0	0	1	0	0	0	0	1				rs1			rm		rd	1	0	1	0	0	1	1	1			
FEQ.D	1	0	1	0	0	0	1									rs2		rs1	0	1	0	rd	1	0	1	0	0	1	1	1		
FLT.D	1	0	1	0	0	0	1									rs2		rs1	0	0	1	rd	1	0	1	0	0	1	1	1		
FLE.D	1	0	1	0	0	0	1									rs2		rs1	0	0	0	rd	1	0	1	0	0	1	1	1		
FCLASS.D	1	1	1	0	0	0	1	0	0	0	0	0				rs1			0	0	1	rd	1	0	1	0	0	1	1	1		
FCVT.W.D	1	1	0	0	0	0	1	0	0	0	0	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FCVT.WU.D	1	1	0	0	0	0	1	0	0	0	0	1				rs1			rm		rd	1	0	1	0	0	1	1	1			
FCVT.D.W	1	1	0	1	0	0	1	0	0	0	0	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FCVT.D.WU	1	1	0	1	0	0	1	0	0	0	0	1				rs1			rm		rd	1	0	1	0	0	1	1	1			
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.L.D	1	1	0	0	0	0	1	0	0	0	1	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FCVT.LU.D	1	1	0	0	0	0	1	0	0	0	1	1				rs1			rm		rd	1	0	1	0	0	1	1	1			
FMV.X.D	1	1	1	0	0	0	1	0	0	0	0	0				rs1			0	0	0	rd	1	0	1	0	0	1	1	1		
FCVT.D.L	1	1	0	1	0	0	1	0	0	0	1	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FCVT.D.LU	1	1	0	1	0	0	1	0	0	0	1	1				rs1			rm		rd	1	0	1	0	0	1	1	1			
FMV.D.X	1	1	1	1	0	0	1	0	0	0	0	0				rs1			rm		rd	1	0	1	0	0	1	1	1			
FMV.D.X	1	1	1	1	1	0	0	1	0	0	0	0				rs1			0	0	0	rd	1	0	1	0	0	1	1	1		

Figure 7. RISC-V単精度浮動小数点命令（RV32D / RV64D）の命令エンコーディング2

浮動小数点命令をLLVMに実装する

前章まででMYRISCVXの整数命令に関してLLVMバックエンドへの実装を進めてきました。これだけでも随分と完成度が上がってきましたが、まだ足りていないものがあります。その一つとして、浮動小数点命令のサポートがあります。

RISC-Vは現状では2種類の浮動小数点命令が定義されています。

- F拡張：単精度浮動小数点命令
 - D拡張：倍精度浮動小数点命令
- それぞれ、接尾語に `.S` と `.D` が付けられており、たとえば、
- `fadd.s rd,rs1,rs2,rs3` : 単精度FMA演算命令 : $freg[rd] = freg[rs1] \times freg[rs2] + freg[rs3]$
 - `fadd.d rd,rs1,rs2,rs3` : 倍精度FMA演算命令 : $freg[rd] = freg[rs1] \times freg[rs2] + freg[rs3]$
- などが定義されています。ここでは、これらの浮動小数点命令を追加してみましょう。

なお、RISC-Vの命令セットの規定によると、単精度浮動小数点命令は「F拡張」、倍精度浮動小数点命令は「D拡張」として定義されています。コンパイラにおいてもそれらのスイッチを用意するべきですが、今回は簡易化のため、MYRISCVX32、MYRISCVX64のどちらでも単精度・倍精度の浮動小数点演算命令が実行可能としています。

[LLVM] 浮動小数点レジスタを定義

まずは、浮動小数点レジスタを定義しないと始まりません。RISC-Vの浮動小数点レジスタは、実体としては単精度版と倍精度版が共有されています。つまり、倍精度をサポートしているRV64Dのアーキテクチャならば64ビットのレジスタが32本あり、下位の32ビットが単精度レジスタと共有されています。一方、単精度をサポートしているRV32Fのみのサポートで、RV64Dをサポートしていないようなアーキテクチャならば、倍精度命令はそのままでは実行できません。エミュレートなり、別の方針を模索する必要があります。

ここではRV32F用の単精度レジスタ、RV64D用の倍精度レジスタを定義します。しかし、これは共有関係にあるので、以下のような実装方針を取ります。

1. 単精度浮動小数点レジスタ32本をこれまで通り定義する。
2. 倍精度浮動小数点レジスタ32本を定義するが、64ビットのうち下位の32ビットを単精度レジスタと共有するようにサブレジスタを指定する。

このようなサブルジスタによる共有は、冷静に考えてみると必須の機能です。X86ではAX、EAXレジスタなどのレジスタフィールドが共有されているレジスタは沢山ありますので、このようなレジスタ共有の機能があるのは当然です。

というわけで、まずは単精度浮動小数点レジスタを定義します。[MYRISCVXRegisterInfo.td](#) に以下の定義を追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXRegisterInfo.td](#)

```
// Floating point registers
let Namespace = "MYRISCVX" in {
    def F0_S : MYRISCVXGPRReg< 0, "f0", ["ft0"]>, DwarfRegNum<[32]>;
    def F1_S : MYRISCVXGPRReg< 1, "f1", ["ft1"]>, DwarfRegNum<[33]>;
    def F2_S : MYRISCVXGPRReg< 2, "f2", ["ft2"]>, DwarfRegNum<[34]>;
    def F3_S : MYRISCVXGPRReg< 3, "f3", ["ft3"]>, DwarfRegNum<[35]>;
    ...
    def F29_S : MYRISCVXGPRReg<29, "f29", ["ft9"]>, DwarfRegNum<[61]>;
    def F30_S : MYRISCVXGPRReg<30, "f30", ["ft10"]>, DwarfRegNum<[62]>;
    def F31_S : MYRISCVXGPRReg<31, "f31", ["ft11"]>, DwarfRegNum<[63]>;
}
```

- [llvm/lib/Target/MYRISCVX/MYRISCVXRegisterInfo.td](#)

```
// Single Floating Point Register Class
def FPR_S : RegisterClass<"MYRISCVX", [f32], 32, (add
    F0_S, F1_S, F2_S, F3_S, F4_S, F5_S, F6_S, F7_S,
    F8_S, F9_S, F10_S, F11_S, F12_S, F13_S, F14_S, F15_S,
    F16_S, F17_S, F18_S, F19_S, F20_S, F21_S, F22_S, F23_S,
    F24_S, F25_S, F26_S, F27_S, F28_S, F29_S, F30_S, F31_S
)>;
```

[MYRISCVXGPRReg](#) クラスを使用して単精度レジスタを32本定義します。整数レジスタの定義とあまり変わ

りませんが、`DwarfRegNum` の定義のみ少し注意が必要です。`DwarfRegNum` はGCCやGDBがレジスタを参照するための一意に決められたレジスタ番号なので、整数レジスタと番号が被ってはいけません。したがって、单精度レジスタ32本では32-63までの番号を使っています。

次に、倍精度浮動小数点レジスタを定義します。上記で説明した通り、サブレジスタ指定を使って单精度レジスタを包含します。このために、`MYRISCVXFPR_D` レジスタクラスを作成します。

- `llvm/lib/Target/MYRISCVX/MYRISCVXRegisterInfo.td`

```
def sub_32 : SubRegIndex<32>;
// MYRISCVX Double Floating-Point Registers
class MYRISCVXFPR_D<bits<5> Enc, list<string> alt,
    MYRISCVXGPRReg subreg, list<SubRegIndex> subreg_indices> :
Register<""> {
    let HWEncoding{4-0} = Enc;
    let SubRegs = [subreg];
    let SubRegIndices = [sub_32];
    let AsmName = subreg.AsmName;
    let AltNames = subreg.AltNames;
}
```

要点としては以下です。

- `MYRISCVXFPR_D` は `Register` クラスを継承します。ただし、その名前 `AsmRegister` はとりあえず空白 "" とします。
- ハードウェエンコーディング `HWEncoding` は指定したものをそのまま使用します。
- サブレジスタとして `SubRegs` を指定します。これは配列を指定できるので、`[subreg]` として配列を指定します。`subreg` は倍精度浮動小数点レジスタ定義時に指定します。
- レジスタ名 `AsmName` と代替レジスタ名 `AltNames` をここで指定します。これは、`subreg` のものをそのまま使用します。

`MYRISCVXFPR_D` クラスを利用して、倍精度レジスタを定義します。

- `llvm/lib/Target/MYRISCVX/MYRISCVXRegisterInfo.td`

```
def F0_D : MYRISCVXFPR_D< 0, ["ft0" ], F0_S , [sub_32]>, DwarfRegNum<[64]>;
def F1_D : MYRISCVXFPR_D< 1, ["ft1" ], F1_S , [sub_32]>, DwarfRegNum<[65]>;
def F2_D : MYRISCVXFPR_D< 2, ["ft2" ], F2_S , [sub_32]>, DwarfRegNum<[66]>;
def F3_D : MYRISCVXFPR_D< 3, ["ft3" ], F3_S , [sub_32]>, DwarfRegNum<[67]>;
...
def F29_D : MYRISCVXFPR_D<29, ["ft9" ], F29_S, [sub_32]>, DwarfRegNum<[93]>;
def F30_D : MYRISCVXFPR_D<30, ["ft10"], F30_S, [sub_32]>, DwarfRegNum<[94]>;
def F31_D : MYRISCVXFPR_D<31, ["ft11"], F31_S, [sub_32]>, DwarfRegNum<[95]>;
```

32本分の倍精度レジスタを定義し、それぞれサブレジスタとして单精度レジスタを定義しました。

ただし、これだけだとコンパイル時にエラーとなります。これは、現状の設定だとレジスタ名 `AltName` が

重複する定義を許さないためです。 現状では単精度レジスタは `f0-f31` 、倍精度レジスタも `f0-f31` という名前で定義されており、レジスタ名が重複しています。

```
LLVM ERROR: Had duplicate keys to match on
```

そこで、`MYRISCVX.td` にレジスタの重複を許可する設定を追加します。

- `llvm/lib/Target/MYRISCVX/MYRISCVX.td`

```
def MYRISCVXAsmParser : AsmParser {  
    // AsmParserのParse時にAltNameで指定したレジスタ名も受け付ける  
    let ShouldEmitMatchRegisterAltName = 1;  
    let AllowDuplicateRegisterNames = 1;  
}
```

`AllowDuplicateRegisterNames` を1に設定することで、レジスタ名が重複してもエラーを出しません。 これで最後までコンパイルが通るようになります。

そしてこれらのレジスタをMYRISCVXの浮動小数点レジスタとして宣言します。

- `llvm/lib/Target/MYRISCVX/MYRISCVXISelLowering.cpp`

```
MYRISCVXTargetLowering::MYRISCVXTargetLowering(const MYRISCVXTargetMachine &TM,  
                                                const MYRISCVXSubtarget &STI)  
...  
    // レジスタクラスをセットアップする  
    addRegisterClass(XLenVT, &MYRISCVX::GPRRegClass);  
  
    addRegisterClass(MVT::f32, &MYRISCVX::FPR_SRegClass);  
    addRegisterClass(MVT::f64, &MYRISCVX::FPR_DRegClass);
```

最後に、浮動小数点レジスタ用のディスアセンブルコードを追加します。これは `MYRISCVXGenDisassemblerTables.cpp` から呼ばれているため必要なデコーダです。

- `llvm/lib/Target/MYRISCVX/Disassembler/MYRISCVXDisassembler.cpp`

```
static DecodeStatus DecodeFPR_SRegisterClass(MCInst &Inst,  
                                            unsigned RegNo,  
                                            uint64_t Address,  
                                            const void *Decoder) {  
    if (RegNo > 32)  
        return MCDisassembler::Fail;  
  
    Inst.addOperand(MCOperand::createReg(MYRISCVX::F0_D + RegNo));  
    return MCDisassembler::Success;  
}
```

```

static DecodeStatus DecodeFPR_DRegisterClass(MCInst &Inst,
                                             unsigned RegNo,
                                             uint64_t Address,
                                             const void *Decoder) {
    if (RegNo > 32)
        return MCDisassembler::Fail;

    Inst.addOperand(MCOperand::createReg(MYRISCVX::F0_S + RegNo));
    return MCDisassembler::Success;
}

```

[LLVM] 浮動小数点命令の追加：ロード・ストア命令

では、まずは簡単な浮動小数点命令の実装から入りましょう。演算命令を定義するところから始めてもよいですが、そのまえに必ずロードストア命令が必要になるので、そこから入ります。

RISC-Vでは、単精度浮動小数点レジスタ用の **FLW / FSW**、倍精度浮動小数点レジスタの **FLD / FSD** 命令が定義されています。これを定義しましょう。整数レジスタへのロードストア命令についてですが、テンプレートとして **LoadMemory** クラス、**StoreMemory** クラスを定義していました。これらは対象となるレジスタクラスを指定できるようにしていったので、レジスタクラスを浮動小数点命令のものに変えれば上手く行きそうです。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfo.td](#)

```

// メモリロードストアのためのテンプレートを作成
let canFoldAsLoad = 1 in
class LoadMemory<bits<7> opcode, bits<3> funct3, string instr_asm, RegisterClass RC>:
    MYRISCVX_I<opcode, funct3, (outs RC:$rd), (ins GPR:$rs1, simm12:$simm12),
    !strconcat(instr_asm, "\t$rd, ${simm12}(${rs1})"),
    [], IILoad>;

class StoreMemory<bits<7> opcode, bits<3> funct3, string instr_asm, RegisterClass RC>:
    MYRISCVX_S<opcode, funct3, (outs), (ins RC:$rs2, GPR:$rs1, simm12:$simm12),
    !strconcat(instr_asm, "\t$rs2, ${simm12}(${rs1})"),
    [], IIStore>;

```

これに追加する形で、**FLW / FSW / FLD / FSD** の定義を追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

def FLW : LoadMemory <0b0000111, 0b010, "flw", FPR_S>;
def FSW : StoreMemory<0b0100111, 0b010, "fsw", FPR_S>;

def FLD : LoadMemory <0b0000111, 0b011, "fld", FPR_D>;
def FSD : StoreMemory<0b0100111, 0b011, "fsd", FPR_D>;

```

`LoadPattern` と `StorePattern` を使用して浮動小数点レジスタのロードストア命令のパターンを定義しています。しかしここれまでの `StoreMemory` は汎用レジスタ `GPR` のみを扱うパターンになってしまっているため、修正して浮動小数点命令にも対応させます。

- `llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td`

```
defm : LoadPattern <load_a, FLW>;
defm : StorePattern<store, FSW, FPR_S>;  
  
defm : LoadPattern <load_a, FLD>;
defm : StorePattern<store , FSD, FPR_D>;
```

では、ここまで来たら LLVM のビルドを行い、テストプログラムを動かしてみます。以下のような簡単なコードを考えます。

- `/program/appendix_2/fp_mem.c`

```
void fp_memoy(float *dst, float *src)
{
    *dst = *src;
}
void dp_memoy(double *dst, double *src)
{
    *dst = *src;
}
```

これをコンパイルしてみましょう。

```
$ ${BUILD}/bin/clang --target=riscv32-unknown-elf fp_mem.c -c -emit-llvm \
-o fp_mem.riscv32.static.bc
$ ${BUILD}/bin/llc -march=myriscvx32           -debug -disable-tail-calls \
-relocation-model=static -filetype=asm fp_mem.riscv32.static.bc \
-o fp_mem.myriscvx32.static.S

$ ${BUILD}/bin/clang --target=riscv64-unknown-elf fp_mem.c -c -emit-llvm \
-o fp_mem.riscv64.static.bc
$ ${BUILD}/bin/llc -march=myriscvx64           -debug -disable-tail-calls \
-relocation-model=static -filetype=asm fp_mem.riscv64.static.bc \
-o fp_mem.myriscvx64.static.S
```

単精度側のアセンブリです。正しく `flw` と `fsw` が生成されていることが分かります。

- `fp_mem.myriscvx32.static.S`

```
fp_memoy:                                # @fp_memoy
# 最適化オプションを指定していないので無駄な整数ロードストア命令が生成されている。
# %bb.0:                                # %entry
```

```

addi    x2, x2, -8
sw      x10, 4(x2)          # 第一引数dstを関数フレームに格納
sw      x11, 0(x2)          # 第二引数srcを関数フレームに格納
lw      x10, 0(x2)
flw    f0, 0(x10)          # srcの場所から値をロード
lw      x10, 4(x2)
fsw    f0, 0(x10)          # dstの場所に値をストア
addi    x2, x2, 8
ret

```

倍精度側のアセンブリです。正しく `fld` と `fsd` が生成されていることが分かります。

- `fp_mem.myriscvx32.static.S`

```

dp_memoy:                                # @dp_memoy
# 最適化オプションを指定していないので無駄な整数ロードストア命令が生成されている。
# %bb.0:                                # %entry
    addi    x2, x2, -8
    sw      x10, 4(x2)          # 第一引数dstを関数フレームに格納
    sw      x11, 0(x2)          # 第二引数srcを関数フレームに格納
    lw      x10, 0(x2)
    fld    f0, 0(x10)          # srcの場所から値をロード
    lw      x10, 4(x2)
    fsd    f0, 0(x10)          # dstの場所に値をストア
    addi    x2, x2, 8
    ret

```

[LLVM] 浮動小数点算術演算命令の追加

浮動小数点レジスタのロードストア命令が生成できるようになったので、次は簡単な算術演算ができるようになります。とりあえずは、加減乗除、そして積和演算ができるようになりたいです。

私たちはすでに命令定義のクラスとテンプレートを持っていますので、そこに当てはめるだけで良いです。

- `llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td`

```

def FADD_S : ArithLogicR<0b1010011, 0b000, 0b0000000, "fadd.s", fadd, FPR_S>;
def FSUB_S : ArithLogicR<0b1010011, 0b000, 0b0000100, "fsub.s", fsub, FPR_S>;
def FMUL_S : ArithLogicR<0b1010011, 0b000, 0b0001000, "fmul.s", fmul, FPR_S>;
def FDIV_S : ArithLogicR<0b1010011, 0b000, 0b0001100, "fdiv.s", fdiv, FPR_S>;

def FADD_D : ArithLogicR<0b1010011, 0b000, 0b0000001, "fadd.d", fadd, FPR_D>;
def FSUB_D : ArithLogicR<0b1010011, 0b000, 0b0000101, "fsub.d", fsub, FPR_D>;
def FMUL_D : ArithLogicR<0b1010011, 0b000, 0b0001001, "fmul.d", fmul, FPR_D>;
def FDIV_D : ArithLogicR<0b1010011, 0b000, 0b0001101, "fdiv.d", fdiv, FPR_D>;

```

单精度版では、使用するレジスタとして `FPR_S`、倍精度版では `FPR_D` を指定しました。それぞれ、生成パ

タンに使用する演算としては `fadd`, `fsub`, `fmul`, `fdiv` を指定しています。 これだけで、`fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fadd.d`, `fsub.d`, `fmul.d`, `fdiv.d` が生成できます。

これでサンプルプログラムをコンパイルしてみましょう。以下のようなプログラムをコンパイルしてみます。

- `/program/appendix_2/fp_ops_simple.c`

```
void fp_math_simple(float *a, float *b,
                     float *res_add, float *res_sub, float *res_mul, float *res_div)
{
    *res_add = *a + *b;
    *res_sub = *a - *b;
    *res_mul = *a * *b;
    *res_div = *a / *b;
}

void fp_math_double(double *a, double *b,
                    double *res_add, double *res_sub, double *res_mul, double
*res_div)
{
    *res_add = *a + *b;
    *res_sub = *a - *b;
    *res_mul = *a * *b;
    *res_div = *a / *b;
}
```

少しややこしいテストになっています。引数を `float *a` などとポインタ指定にしています。 これは現在まだ浮動小数点命令向けのABIを定義していないため `float a` と浮動小数点データをそのまま引数で渡すとエラーになります。 そこでポインタを渡し、そのポインタ位置から浮動小数点ロード命令を使って引数データをロードするという作戦を取っています。 計算結果についても戻り値として直接渡すのではなく引数に結果を取得するためのポインタ `float *result` を渡しています。

```
 ${BUILD}/bin/clang -O3 fp_ops_simple.c -c -emit-llvm \
 -o fp_ops_simple.riscv64.static.bc

 ${BUILD}/bin/llc -march=myriscvx64           -debug -disable-tail-calls -relocation
 -model=static \
 -filetype=asm fp_ops_simple.riscv64.static.bc \
 -o fp_ops_simple.myriscvx64.static.S
```

実行結果は以下のようになりました。浮動小数点算術演算命令が生成されていることが分かります。

```
# 単精度版のコンパイル結果は省略
```

```
test_dp_math:                                # @test_dp_math
# %bb.0:                                     # %entry
```

```

fld      f0, 0(x12)
fld      f1, 0(x11)
fld      f2, 0(x10)
fadd.d  f3, f2, f1
fadd.d  f3, f3, f0
fsub.d  f4, f2, f1
fsub.d  f4, f4, f0
fadd.d  f3, f4, f3
fmul.d  f4, f2, f1
fmul.d  f4, f4, f0
fadd.d  f3, f4, f3
fdiv.d  f1, f2, f1
fdiv.d  f0, f1, f0
fadd.d  f0, f0, f3
fsd      f0, 0(x13)
ret

```

さらに、積和演算も定義します。積和演算命令は以下のようなフォーマットになっています。

31	27 26	25 24	20 19	15 14	12 11	7 6	0
rs3	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB	

このフォーマットで、積和演算用の命令クラスを作成します。[MYRISCVX_FMA](#) クラスとします。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrFormats.td](#)

```

//=====
// FMA-Type instruction class in MYRISCVX : <|opcode|rd|rs1|rs2|rs3|>
//=====

class MYRISCVX_FMA<bits<7> opcode, bits<2> fmt, bits<3> rm,
                  dag outs, dag ins, string asmstr, list<dag> pattern,
                  InstrItinClass itin>:
    MYRISCVXInst<outs, ins, asmstr, pattern, itin, FrmR>
{
    bits<5> rs3;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;

    let Inst{31-27} = rs3;
    let Inst{26-25} = fmt;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = rm;
    let Inst{11-7}   = rd;
    let Inst{6-0}    = opcode;
}

```

```
}
```

さらに`MYRISCVX_FMA`クラスを使いやすくした、**FPMultAdd** クラスを作成しておきます。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
// Floating-Point FMA instructions with 3 register operands.  
class FPMultAdd<bits<7> opcode, bits<2> fmt, bits<3> rm,  
    string instr_asm,  
    RegisterClass RC> :  
    MYRISCVX_FMA<opcode, fmt, rm, (outs RC:$rd), (ins RC:$rs1, RC:$rs2, RC:$rs3),  
        !strconcat(instr_asm, "\t$rd, $rs1, $rs2, $rs3"),  
        [], IIAlu> {  
    let isReMaterializable = 1;  
}
```

このテンプレートを元に、単精度用と倍精度用の積和演算命令を追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
//  
// Single Floating Point Operations  
//  
def FMADD_S : FPMultAdd<0b1010011, 0b00, 0b000, "fmadd.s", FPR_S>;  
def FMSUB_S : FPMultAdd<0b1000111, 0b00, 0b000, "fmsub.s", FPR_S>;  
def FNMSUB_S : FPMultAdd<0b1001011, 0b00, 0b000, "fnmsub.s", FPR_S>;  
def FNMADD_S : FPMultAdd<0b1001111, 0b00, 0b000, "fnmadd.s", FPR_S>;  
//  
// Double Floating Point Operations  
//  
def FMADD_D : FPMultAdd<0b1010011, 0b01, 0b000, "fmadd.d", FPR_D>;  
def FMSUB_D : FPMultAdd<0b1000111, 0b01, 0b000, "fmsub.d", FPR_D>;  
def FNMSUB_D : FPMultAdd<0b1001011, 0b01, 0b000, "fnmsub.d", FPR_D>;  
def FNMADD_D : FPMultAdd<0b1001111, 0b01, 0b000, "fnmadd.d", FPR_D>;
```

さらに、これらの命令の生成パターンを追加します。**FMADD** に関しては **fma** というパターンがあるのですが、それ以外についてもデフォルトで用意されていないので、手動で追加します。以下では単精度の生成パターンを追加していますが、倍精度も同様に追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
// Single Floating-Point FMA pattern registration  
def : Pat<(fma FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3),  
        (FMADD_S $rs1, $rs2, $rs3)>;  
def : Pat<(fma FPR_S:$rs1, FPR_S:$rs2, (fneg FPR_S:$rs3)),  
        (FMSUB_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;  
def : Pat<(fma (fneg FPR_S:$rs1), FPR_S:$rs2, FPR_S:$rs3),  
        (FNMSUB_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
```

```

def : Pat<(fma (fneg FPR_S:$rs1), FPR_S:$rs2, (fneg FPR_S:$rs3)),
        (FNMADD_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
def : Pat<(fadd (fmul      FPR_S:$rs1, FPR_S:$rs2), FPR_S:$rs3),
        (FMADD_S $rs1, $rs2, $rs3)>;
def : Pat<(fsub (fmul      FPR_S:$rs1, FPR_S:$rs2), FPR_S:$rs3),
        (FMSUB_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
def : Pat<(fadd (fmul (fneg FPR_S:$rs1), FPR_S:$rs2), FPR_S:$rs3),
        (FNMSUB_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
def : Pat<(fsub (fmul (fneg FPR_S:$rs1), FPR_S:$rs2), FPR_S:$rs3),
        (FNMADD_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
def : Pat<(fadd (fneg (fmul FPR_S:$rs1, FPR_S:$rs2)), FPR_S:$rs3),
        (FNMSUB_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
def : Pat<(fsub (fneg (fmul FPR_S:$rs1, FPR_S:$rs2)), FPR_S:$rs3),
        (FNMADD_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;
def : Pat<(fsub FPR_S:$rs3, (fmul FPR_S:$rs1, FPR_S:$rs2)),
        (FNMSUB_S FPR_S:$rs1, FPR_S:$rs2, FPR_S:$rs3)>;

```

ここまで出来たら、LLVMをビルドしてテストコードを流してみます。以下のようなテストコードを用意しました。

- [/program/appendix_2/fp_ops.c](#)

```

#include <math.h>

void test_fp_math(float *a, float *b, float *c, float *result)
{
    float res_fmad = *a * *b + *c;
    float res_fsub = *a * *c - res_fmad;
    float res_fnmadd = (-*b) * *c + res_fsub;
    float res_fnmsub = (-*a) * *c - res_fnmadd;
    *result = res_fnmsub;
}

void test_dp_math(double *a, double *b, double *c, double *result)
{
    double res_fmad = *a * *b + *c;
    double res_fsub = *a * *b - res_fmad;
    double res_fnmadd = (-*b) * *c + res_fsub;
    double res_fnmsub = (-*a) * *c - res_fnmadd;
    *result = res_fnmsub;
}

```

```

$ ${BUILD}/bin/clang -O3 fp_ops.c -c -emit-llvm -o fp_ops.riscv64.static.bc
$ ${BUILD}/bin/llc -march=myriscvx64      -debug -disable-tail-calls \
  -relocation-model=static -filetype=asm fp_ops.riscv64.static.bc \

```

```
-o fp_ops.myriscvx64.static.S
```

浮動小数点数に関するABIを定義していないことから、このテストコードも値は引数から取得していますが、値そのものはポインタを使用しています。生成されたアセンブリ命令を見てみます。

```
# 単精度版のコンパイル結果は省略
test_dp_math:                                # @test_dp_math

# %bb.0:                                     # %entry
    fld      f0, 0(x12)
    fld      f1, 0(x11)
    fld      f2, 0(x10)
    fmadd.d f3, f2, f1, f0
    fmsub.d f3, f2, f1, f3
    fnmsub.d      f1, f1, f0, f3
    fnmadd.d      f0, f2, f0, f1
    fsd      f0, 0(x13)
    ret
```

無事に浮動小数点演算命令を生成できました。ここでは省略していますが、倍精度命令のテストも正しく命令を生成できています。

[LLVM] 浮動小数点レジスタ向けのCalling Conventionの追加

浮動小数点演算命令を使った算術演算は生成できるようになりましたが、まだ関数の引数として浮動小数点型を使用できません。これは、MYRISCVXのCalling Conventionに、浮動小数点のサポートを追加していないからです。

RISC-VのCalling Conventionでは、浮動小数点数の関数の引数渡しは整数と同様に定義されています。

Name	ABI Mnemonic	Meaning	Preserved across calls?
f0-f7	ft0-ft7	Temporary registers	No
f8-f9	fs0-fs1	Callee-saved registers	Yes*
f10-f17	fa0-fa7	Argument registers	No
f18-f27	fs2-fs11	Callee-saved registers	Yes*
f28-f31	ft8-ft11	Temporary registers	No

上記の通り、浮動小数点レジスタ経由の引数渡しでは、`f10-f17`が用意されています。戻り値は`f10`経由で戻せばよいでしょう。 というわけで、整数レジスタのCalling Conventionと同様に、2種類のCalling Conventionを用意します。

- **lp32** : 最初の8個までの引数は浮動小数点レジスタ `f10 - f17` で渡す。それから先の引数はスタック経由で渡す。
- **stack32** : すべての浮動小数点型の引数をスタック経由で渡す。

このため、[MYRISCVXCallingConv.td](#) に記述を追加しています。LP32について、f32とf64の型についての引数の情報を指定しました。[f10-f17](#) のレジスタを使用するように指定し([CCAssignToReg](#))、残りはスタックを使用するようにします([CCAssignToStack](#))。

- [llvm/lib/Target/MYRISCVX/MYRISCVXCallingConv.td](#)

```
// RV32の引数渡しConventionを定義する
def CC_LP32 : CallingConv<[
    // i1/i8/i16の型はi32に変換する
    CCIftype<[i1, i8, i16], CCPromoteToType<i32>>,
    // i32型の引数はA0 - A7のレジスタに割り当てる
    CCIftype<[i32], CCAssignToReg<[A0, A1, A2, A3, A4, A5, A6, A7]>>,
    // Single Floating-Point arguments are passed in FP registers
    CCIftype<[f32], CCAssignToReg<[F10_S, F11_S, F12_S, F13_S, F14_S, F15_S, F16_S, F17_S]>>,
    // Double Floating-Point arguments are passed in FP registers
    CCIftype<[f64], CCAssignToReg<[F10_D, F11_D, F12_D, F13_D, F14_D, F15_D, F16_D, F17_D]>>,
    // 引数をスタックに割り当てる場合は4バイトサイズ、4バイトアラインで配置する
    CCIftype<[i32], CCAssignToStack<4, 4>>,
    // 単精度浮動小数点数は4バイトのメモリを割り当て、4バイトアラインで配置する
    CCIftype<[f32], CCAssignToStack<4, 4>>,
    // 倍精度浮動小数点数は8バイトのメモリを割り当て、8バイトアラインで配置する
    CCIftype<[f64], CCAssignToStack<8, 8>>
];
```

戻り値については、浮動小数点レジスタ [f10](#) を経由して戻します。このための記述を追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXCallingConv.td](#)

```
def RetCC_LP32 : CallingConv<[
    CCIftype<[i32], CCAssignToReg<[A0, A1]>>,
    // Floating-Point are return in registers FA0, FA1
    CCIftype<[f32], CCAssignToReg<[F10_S]>>,
    CCIftype<[f64], CCAssignToReg<[F10_D]>>
];
```

これだけです。LLVMをビルドして、サンプルプログラムを動作させてみましょう。以下のようなコードを考えます。

- [/program/appendix_2/fp_args.c](#)

```
float test_fp_arg(float a, float b, float c)
{
    return a + b + c;
```

```

}

double test_dp_arg(double a, double b, double c)
{
    return a + b + c;
}

float test_fp_longarg(float f0, float f1, float f2, float f3, float f4,
                      float f5, float f6, float f7, float f8, float f9)
{
    return f0 + f1 + f2 + f3 + f4 + f5 + f6 + f7 + f8 + f9;
}

double test_dp_longarg(double f0, double f1, double f2, double f3, double f4,
                      double f5, double f6, double f7, double f8, double f9)
{
    return f0 + f1 + f2 + f3 + f4 + f5 + f6 + f7 + f8 + f9;
}

```

```

$ ${BUILD}/bin/clang -O3 fp_args.c -c -emit-llvm -o fp_args.riscv32.static.bc
$ ${BUILD}/bin/llc -march=myriscvx32      -debug -disable-tail-calls \
  -relocation-model=static -filetype=asm fp_args.riscv32.static.bc \
  -o fp_args.myriscvx32.static.S

```

- `test_fp_arg()` のコンパイル結果。`float` の場合は以下のようなコードとなります。

```

test_fp_arg:                                # @test_fp_arg
# %bb.0:                                     # %entry
    fadd.s  f0, f10, f11
    fadd.s  f10, f0, f12
    ret

```

単純に引数渡しで演算を行い、レジスタ経由で戻り値を渡します。

- `test_dp_longarg()` のコンパイル結果。`double` の場合は以下のようなコードとなります。

```

test_dp_longarg:                            # @test_dp_longarg
# %bb.0:                                     # %entry
    fadd.d  f0, f10, f11
    fadd.d  f0, f0, f12
    fadd.d  f0, f0, f13
    fadd.d  f0, f0, f14
    fadd.d  f0, f0, f15
    fadd.d  f0, f0, f16
    fadd.d  f0, f0, f17
    fld     f1, 0(x2)
    fadd.d  f0, f0, f1

```

```

fld      f1, 8(x2)
fadd.d  f10, f0, f1
ret

```

[LLVM] さまざまな浮動小数点命令の追加

浮動小数点比較命令

RISC-Vには、[Figure 6](#)に示すように浮動小数の比較命令として以下が定義されています。

```

/// 最大値・最小値
// 単精度命令
FMIN.S    fd,fs1,fs2      // freq[fd] = min(freq[fs1], freq[fs2])
FMAX.S    fd,fs1,fs2      // freq[fd] = max(freq[fs1], freq[fs2])
// 倍精度命令
FMIN.D    fd,fs1,fs2      // freq[fd] = min(freq[fs1], freq[fs2])
FMAX.D    fd,fs1,fs2      // freq[fd] = max(freq[fs1], freq[fs2])

```

```

/// 比較・クラス分類
// 単精度命令
FEQ.S     rd,fs1,fs2    // reg[rd] = freq[fs1] == freq[fs2]
FLT.S     rd,fs1,fs2    // reg[rd] = freq[fs1] < freq[fs2]
FLE.S     rd,fs1,fs2    // reg[rd] = freq[fs1] <= freq[fs2]
// 倍精度命令
FEQ.D     rd,fs1,fs2    // reg[rd] = freq[fs1] == freq[fs2]
FLT.D     rd,fs1,fs2    // reg[rd] = freq[fs1] < freq[fs2]
FLE.D     rd,fs1,fs2    // reg[rd] = freq[fs1] <= freq[fs2]

```

これらの命令について、パタンを登録するだけです。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

// Arithmetic and logical instructions with 2 register operands.
class FPCompDestR<bits<7> opcode, bits<3> funct3, bits<7>funct7,
           string instr_asm, SDPatternOperator OpNode,
           RegisterClass RC> :
  MYRISCVX_R<opcode, funct3, funct7, (outs GPR:$rd), (ins RC:$rs1, RC:$rs2),
  !strconcat(instr_asm, "\t$rd, $rs1, $rs2"),
  [(set GPR:$rd, (OpNode RC:$rs1, RC:$rs2))], IIAlu> {
    let isReMaterializable = 1;
}

```

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
def FMAX_S : ArithLogicR<0b1010011, 0b001, 0b0010100, "fmax.s", fmaxnum, FPR_S>;
```

```

def FMIN_S : ArithLogicR<0b1010011, 0b000, 0b0010100, "fmin.s", fminnum, FPR_S>;
def FEQ_S : FPCompDestR<0b1010011, 0b010, 0b1010000, "feq.s", seteq, FPR_S>;
def FLT_S : FPCompDestR<0b1010011, 0b001, 0b1010000, "flt.s", setlt, FPR_S>;
def FLE_S : FPCompDestR<0b1010011, 0b000, 0b1010000, "fle.s", settle, FPR_S>;

```

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

def FMAX_D : ArithLogicR<0b1010011, 0b001, 0b0010101, "fmax.s", fmaxnum, FPR_D>;
def FMIN_D : ArithLogicR<0b1010011, 0b000, 0b0010101, "fmin.s", fminnum, FPR_D>;
def FEQ_D : FPCompDestR<0b1010011, 0b010, 0b1010001, "feq.d", seteq, FPR_D>;
def FLT_D : FPCompDestR<0b1010011, 0b001, 0b1010001, "flt.d", setlt, FPR_D>;
def FLE_D : FPCompDestR<0b1010011, 0b000, 0b1010001, "fle.d", settle, FPR_D>;

```

上記の10命令を登録しました。また、比較命令のためのパターンとして [FPCompDestR](#) を定義しました。その名の通り、浮動小数点レジスタ通しの値を比較して、その結果を汎用レジスタに格納するパターンです。

[FPCompDestR](#) クラスでは、命令の定義と同時に命令生成のパターンも登録しています。この命令定義では [seteq](#), [setlt](#), [setle](#) を命令のパターンとして登録していますが、LLVMにはもう一つ比較のパターンが存在します。[setoeq](#), [setolt](#), [setole](#) です。これらは浮動小数点命令特有の演算で、Ordered Opsと言われています。これらのパターンでも命令を生成できるように、パターンを追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

def : Pat<(setueq FPR_S:$rs1, FPR_S:$rs2), (FEQ_S $rs1, $rs2)>;
def : Pat<(setult FPR_S:$rs1, FPR_S:$rs2), (FLT_S $rs1, $rs2)>;
def : Pat<(setule FPR_S:$rs1, FPR_S:$rs2), (FLE_S $rs1, $rs2)>;
def : Pat<(setugt FPR_S:$rs1, FPR_S:$rs2), (FLE_S $rs2, $rs1)>;
def : Pat<(setuge FPR_S:$rs1, FPR_S:$rs2), (FLT_S $rs2, $rs1)>;

def : Pat<(setgt FPR_S:$rs1, FPR_S:$rs2), (FLE_S $rs2, $rs1)>;
def : Pat<(setge FPR_S:$rs1, FPR_S:$rs2), (FLT_S $rs2, $rs1)>;

```

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

def : Pat<(setueq FPR_D:$rs1, FPR_D:$rs2), (FEQ_D $rs1, $rs2)>;
def : Pat<(setult FPR_D:$rs1, FPR_D:$rs2), (FLT_D $rs1, $rs2)>;
def : Pat<(setule FPR_D:$rs1, FPR_D:$rs2), (FLE_D $rs1, $rs2)>;
def : Pat<(setugt FPR_D:$rs1, FPR_D:$rs2), (FLE_D $rs2, $rs1)>;
def : Pat<(setuge FPR_D:$rs1, FPR_D:$rs2), (FLT_D $rs2, $rs1)>;

def : Pat<(setgt FPR_D:$rs1, FPR_D:$rs2), (FLE_D $rs2, $rs1)>;
def : Pat<(setge FPR_D:$rs1, FPR_D:$rs2), (FLT_D $rs2, $rs1)>;

```

さらに注意です。[setle](#), [setlt](#), [setole](#), [setolt](#) だけでなく、その逆もあります。[setgt](#), [setge](#), [setogt](#), [setoge](#) です。これらは [le](#), [lt](#) のオペランドを逆にすればよいので、これらのパターンも追加しています。

また、命令のエイリアスとして `fge.s`, `fgt.s`, `fge.d`, `fgt.d` を追加しました。これはアセンブラーに対応させるためです。

- `llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td`

```
def : InstAlias<"fge.s $rd, $rs1, $rs2",
                 (FLT_S GPR:$rd, FPR_S:$rs2, FPR_S:$rs1), 0>;
def : InstAlias<"fgt.s $rd, $rs1, $rs2",
                 (FLE_S GPR:$rd, FPR_S:$rs2, FPR_S:$rs1), 0>;
```

- `llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td`

```
def : InstAlias<"fge.d $rd, $rs1, $rs2",
                 (FLT_D GPR:$rd, FPR_D:$rs2, FPR_D:$rs1), 0>;
def : InstAlias<"fgt.d $rd, $rs1, $rs2",
                 (FLE_D GPR:$rd, FPR_D:$rs2, FPR_D:$rs1), 0>;
```

ここまでで、LLVMをビルドしてテストを流してみましょう。以下のようなC言語のコードをテストします。

- `/program/appendix_2/fp_cmp.c`

```
int fp_lt_cmp(float a, float b) { return a < b; }
int fp_le_cmp(float a, float b) { return a <= b; }
int fp_gt_cmp(float a, float b) { return a > b; }
int fp_ge_cmp(float a, float b) { return a >= b; }

int dp_lt_cmp(double a, double b) { return a < b; }
int dp_le_cmp(double a, double b) { return a <= b; }
int dp_gt_cmp(double a, double b) { return a > b; }
int dp_ge_cmp(double a, double b) { return a >= b; }
```

```
`${BUILD}/bin/clang -O3 fp_cmp.c -c -emit-llvm \
-o fp_cmp.riscv64.static.bc

`${BUILD}/bin/llc -march=myriscvx64           -debug -disable-tail-calls -relocation
-model=static \
-filetype=asm fp_cmp.riscv64.static.bc \
-o fp_cmp.myriscvx64.static.S
```

結果は以下のようになりました。正しく命令が生成できていることが分かります。

```
_Z9fp_lt_cmpff:
    flt.s  x10, f10, f11
    ret
_Z9fp_le_cmpff:
```

```

fle.s  x10, f10, f11
ret
_Z9fp_gt_cmpff:
fle.s  x10, f11, f10
ret
_Z9fp_ge_cmpff:
flt.s  x10, f11, f10
ret
_Z9dp_lt_cmpdd:
flt.d  x10, f10, f11
ret
_Z9dp_le_cmpdd:
fle.d  x10, f10, f11
ret
_Z9dp_gt_cmpdd:
fle.d  x10, f11, f10
ret
_Z9dp_ge_cmpdd:
flt.d  x10, f11, f10
ret

```

浮動小数点数のその他の命令

比較命令以外にも、浮動小数点数のさまざまな演算をサポートしましょう。たとえば、以下のようなものが挙げられます。

- 平方根
- 浮動小数点符号反転
- 浮動小数点絶対値
- 符号付整数から浮動小数点数への変換、符号なし整数から浮動小数点数への変換
- 浮動小数点数から符号付整数への変換、浮動小数点数から符号なし整数への変換

これらの操作について、RISC-Vの命令を定義しています。[MYRISCVXInstrInfoFD.td](#) に以下を追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

// Floating-Point instructions with 1 register operands.
class FPTwoOp<bits<7> opcode, bits<3> rm, bits<7> funct7,
              string instr_asm,
              RegisterClass RC> :
    MYRISCVX_R<opcode, rm, funct7, (outs RC:$rd), (ins RC:$rs1, RC:$rs2),
        !strconcat(instr_asm, "\t$rd, $rs1, $rs2"),
        [], IIAlu> {
    let isReMaterializable = 1;
}

```

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
// Floating-Point instructions with 1 register operands.
class FPSSingleOp<bits<7> opcode, bits<3> rm, bits<7> funct7, bits<5> rs2_op,
    string instr_asm,
    RegisterClass DstRC, RegisterClass SrcRC> :
    MYRISCVX_R<opcode, rm, funct7, (outs DstRC:$rd), (ins SrcRC:$rs1),
        !strconcat(instr_asm, "\t$rd, $rs1"),
        [], IIAlu> {
    let isReMaterializable = 1;
    let rs2 = rs2_op;
}
```

まずは、1オペランド用の命令と2オペランド用の命令テンプレートを用意しました。それぞれ `FPSSingleOp`, `FPTwoOp` というテンプレートクラスを作成します。`FPSSingleOp` テンプレートでは、型の変換命令が入るため入力オペランドのレジスタタイプ(`RegisterClass SrcRC`)と出力オペランドのレジスタタイプ(`RegisterClass DstRC`)が個別に指定できます。これに基づいて単精度浮動小数点命令を定義します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
def FSQRT_S : FPSSingleOp<0b1010011, 0b000, 0b0101100, 0b00000, "fsqrt.s", FPR_S,
FPR_S>;
def FSIGNJ_S : FPTwoOp<0b1010011, 0b000, 0b0010000, "fsgnj.s", FPR_S>;
def FSIGNJN_S : FPTwoOp<0b1010011, 0b001, 0b0010000, "fsgnjn.s", FPR_S>;
def FSIGNJX_S : FPTwoOp<0b1010011, 0b010, 0b0010000, "fsgnjx.s", FPR_S>;
def FCVT_W_S : FPSSingleOp<0b1010011, 0b000, 0b1100000, 0b00000, "fcvt.w.s", GPR,
FPR_S>;
def FCVT_WU_S : FPSSingleOp<0b1010011, 0b000, 0b1100000, 0b00001, "fcvt.wu.s", GPR,
FPR_S>;
def FMV_X_W : FPSSingleOp<0b1010011, 0b000, 0b1110000, 0b00000, "fmv.x.w", GPR,
FPR_S>;
def FCLASS_S : FPSSingleOp<0b1010011, 0b001, 0b1110000, 0b00000, "fclass.s", FPR_S,
FPR_S>;
def FCVT_S_W : FPSSingleOp<0b1010011, 0b000, 0b1101000, 0b00000, "fcvt.s.w", FPR_S,
GPR>;
def FCVT_S_WU : FPSSingleOp<0b1010011, 0b000, 0b1101000, 0b00001, "fcvt.s.wu", FPR_S,
GPR>;
def FMV_W_X : FPSSingleOp<0b1010011, 0b000, 0b1111000, 0b00000, "fmv.w.x", FPR_S,
GPR>;
```

次に、これらの命令を生成するためのSelectionDAGのパターンを追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```
def : Pat<(fsqrt FPR_S:$rs1), (FSQRT_S $rs1)>;
def : Pat<(fneg FPR_S:$rs1), (FSIGNJN_S $rs1, $rs1)>;
def : Pat<(fabs FPR_S:$rs1), (FSIGNJX_S $rs1, $rs1)>;
def : InstAlias<"fmv.s $rd, $rs1", (FSIGNJ_S FPR_S:$rd, FPR_S:$rs1, FPR_S:$rs1), 0>;
```

```

def : InstAlias<"fneg.s $rd, $rs1", (FSIGNJN_S FPR_S:$rd, FPR_S:$rs1, FPR_S:$rs1), 0>;
def : InstAlias<"fabs.s $rd, $rs1", (FSIGNJX_S FPR_S:$rd, FPR_S:$rs1, FPR_S:$rs1), 0>;
def : Pat<(fp_to_sint FPR_S:$rs1), (FCVT_W_S $rs1)>;
def : Pat<(fp_to_uint FPR_S:$rs1), (FCVT_WU_S $rs1)>;
def : Pat<(sint_to_fp GPR:$rs1), (FCVT_S_W $rs1)>;
def : Pat<(uint_to_fp GPR:$rs1), (FCVT_S_WU $rs1)>;

```

エイリアスとして `fmv.s`, `fneg.s`, `fabs.s` を追加しました。アセンブリでは、これらの記法も有効になります。

同様に、倍精度浮動小数点数についても命令の定義とパターンを追加します。

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

def FSQRT_D    : FPSingleOp<0b1010011, 0b000, 0b0101101, 0b00000, "fsqrt.d",   FPR_D,
FPR_D>;
def FSIGNJ_D   : FPTwoOp<0b1010011, 0b000, 0b0010001, "fsgnj.d" , FPR_D>;
def FSIGNJN_D  : FPTwoOp<0b1010011, 0b001, 0b0010001, "fsgnjn.d", FPR_D>;
def FSIGNJX_D  : FPTwoOp<0b1010011, 0b010, 0b0010001, "fsgnjx.d", FPR_D>;
def FCVT_W_D   : FPSingleOp<0b1010011, 0b000, 0b1100001, 0b00000, "fcvt.w.d",   GPR,
FPR_D>;
def FCVT_WU_D  : FPSingleOp<0b1010011, 0b000, 0b1100001, 0b00001, "fcvt.wu.d",  GPR,
FPR_D>;
// def FMV_X_W   : FPSingleOp<0b1010011, 0b000, 0b1110001, 0b00000, "fmv.x.w",   GPR,
FPR_D>;
def FCLASS_D   : FPSingleOp<0b1010011, 0b001, 0b1110001, 0b00000, "fclass.d",  FPR_D,
FPR_D>;
def FCVT_D_W   : FPSingleOp<0b1010011, 0b000, 0b1101001, 0b00000, "fcvt.d.w",  FPR_D,
GPR>;
def FCVT_D_WU  : FPSingleOp<0b1010011, 0b000, 0b1101001, 0b00001, "fcvt.d.wu", FPR_D,
GPR>;
// def FMV_W_X   : FPSingleOp<0b1010011, 0b000, 0b1111001, 0b00000, "fmv.w.x",
FPR_D, GPR>;

```

- [llvm/lib/Target/MYRISCVX/MYRISCVXInstrInfoFD.td](#)

```

def : Pat<(fsqrt FPR_D:$rs1), (FSQRT_D $rs1)>;
def : Pat<(fneg FPR_D:$rs1), (FSIGNJN_D $rs1, $rs1)>;
def : Pat<(fabs FPR_D:$rs1), (FSIGNJX_D $rs1, $rs1)>;
def : InstAlias<"fmv.d $rd, $rs1", (FSIGNJ_D FPR_D:$rd, FPR_D:$rs1, FPR_D:$rs1), 0>;
def : InstAlias<"fneg.d $rd, $rs1", (FSIGNJN_D FPR_D:$rd, FPR_D:$rs1, FPR_D:$rs1), 0>;
def : InstAlias<"fabs.d $rd, $rs1", (FSIGNJX_D FPR_D:$rd, FPR_D:$rs1, FPR_D:$rs1), 0>;
def : Pat<(fp_to_sint FPR_D:$rs1), (FCVT_W_D $rs1)>;
def : Pat<(fp_to_uint FPR_D:$rs1), (FCVT_WU_D $rs1)>;
def : Pat<(sint_to_fp GPR:$rs1), (FCVT_D_W $rs1)>;
def : Pat<(uint_to_fp GPR:$rs1), (FCVT_D_WU $rs1)>;

```

それでは、テストパターンを作成してコンパイルしてみましょう。

- /program/appendix_2/fp_others.c

```
#include <math.h>
#include <stdint.h>

// float f_sqrt (float in) { return sqrtf(in); }
float f_abs (float in) { return fabsf(in); }
float f_neg (float in) { return -in; }

// double d_sqrt (double in) { return sqrt(in); }
double d_abs (double in) { return fabs(in); }
double d_neg (double in) { return -in; }

int32_t cvt_fp_to_sint (float in) { return (int32_t)(in); }
uint32_t cvt_fp_to_uint (float in) { return (uint32_t)(in); }
int32_t cvt_dp_to_sint (double in) { return (int32_t)(in); }
uint32_t cvt_dp_to_uint (double in) { return (uint32_t)(in); }

float cvt_sint_to_fp (int32_t in) { return (float)(in); }
float cvt_uint_to_fp (uint32_t in) { return (float)(in); }
double cvt_sint_to_dp (int32_t in) { return (double)(in); }
double cvt_uint_to_dp (uint32_t in) { return (double)(in); }

float f_sqrt(float k) { return sqrt(k); }
```

単純な関数群を並べました。型の変換や、1オペランドの演算を並べます。これらがどのようにコンパイルされるのかをテストします。

```
 ${BUILD}/bin/clang -O3 fp_others.c -c -emit-llvm \
 -o fp_others.riscv64.static.bc

 ${BUILD}/bin/llc -march=myriscvx64           -debug -disable-tail-calls -relocation
 -model=static \
 -filetype=asm fp_others.riscv64.static.bc \
 -o fp_others.myriscvx64.static.S
```

```
_Z5f_absf:
    fsgnjx.s      f10, f10, f10
    ret
_Z5f_negf:
    fsgnjn.s      f10, f10, f10
    ret
_Z5d_absd:
    fsgnjx.d      f10, f10, f10
    ret
_Z5d_negd:
    fsgnjn.d      f10, f10, f10
    ret
```

```

_Z14cvt_fp_to_sintf:
    fcvt.w.s      x10, f10
    ret
_Z14cvt_fp_to_uintf:
    fcvt.wu.s     x10, f10
    ret
_Z14cvt_dp_to_sintd:
    fcvt.w.d      x10, f10
    ret
_Z14cvt_dp_to_uintd:
    fcvt.wu.d     x10, f10
    ret
_Z14cvt_sint_to_fpi:
    fcvt.s.w      f10, x10
    ret
_Z14cvt_uint_to_fpj:
    fcvt.s.wu     f10, x10
    ret
_Z14cvt_sint_to_dpi:
    fcvt.d.w      f10, x10
    ret
_Z14cvt_uint_to_dpj:
    fcvt.d.wu     f10, x10
    ret

```

正しく命令が生成できていそうです。

まとめとレッスン：浮動小数点演算を用いたマンデルブロ集合の描画

浮動小数点演算命令を追加することで、かなり広範囲なプログラムをコンパイルして実行できるようになりました。ここでは少し複雑で計算量の多いプログラムをコンパイルして、私たちの開発したLLVMバックエンドが正しく動作することを見てみましょう。

今回作成するプログラムは「マンデルブロ集合」という集合によって描かれる図形を出力するプログラムです。マンデルブロ集合は以下のような図形として表現されることが多く、情報科学の勉強をしたことのある読者であれば一度は見たことがあると思います。

このマンデルブロ集合の原理的な部分はとりあえずおいておいて、マンデルブロ集合は複素数の浮動小数点データを用いて計算されることが多いです。ある複素変数 z （ここでは実数部を x 、虚数部を y とする）のペアで表現される平面上の中で、以下のよう漸化式を計算していきます。

$z_0 = 0, z_{n+1} = z^2 + c$ (c は平面上の x, y 座標とする) とし、この z が無限大に発散しなければマンデルブロ集合に含まれる、そうでなければ含まれないという定義です。平面上で、マンデルブロ集合に含まれる点をプロットすると、Figure 8のような図形を描画できます。

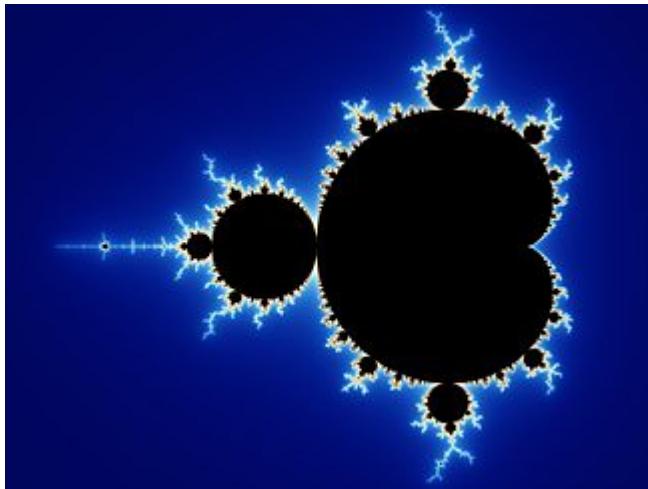


Figure 8. マンデルブロ集合の例。Wikipedia(<https://ja.wikipedia.org/wiki/マンデルブロ集合>) より引用

普通このような複素数のデータをC++言語で表現するのにはライブラリを使用するのですが、ここでは簡単にするため、C言語で2つの変数 x と y で表現することにします。この時、上記の漸化式は以下のように書き換えることができます。

$$x_{n+1} = x_n^2 - y_n^2 + a, \quad y_{n+1} = 2x_n y_n + b$$

そしてマンデルブロ集合の定義「無限大に発散する」というのを確認するためには、ある非常に大きな回数だけこの漸化式を実行し、その結果あるスレッシュホールド値を上回っていれば発散している、そうでなければ発散していないと考えるようにします。

この定義に則り、ある平面座標 (a, b) がマンデルブロ集合に含まれるかどうかを判定するプログラムは以下のように記述します。

- [/program/appendix_2/lesson_mandelbrot/mandelbrot.c](#)

```
// #include <math.h>

#include "mandelbrot.h"

double mandelbrot(double a, double b)
{
    double x = 0.0;
    double y = 0.0;

    for (int i = 1; i <= NMAX; i++) {
        double zr = x * x - y * y + a;
        double zi = 2 * x * y + b;
        if (zr * zr + zi * zi > 4.0) {
            // *ret_r = zr;
            // *ret_i = zi;
            return log((double)i);
        }
        x = zr;
        y = zi;
    }
    return 0.0;
}
```

```
}
```

上記のプログラムでは20000回漸化式を繰り返す中で、漸化式を計算してスレッショルドを超えるかどうかを判定し、スレッショルドを超えていればマンデルブロ集合に含まれるということで0を返します。そうでなければ発散するまでにかかったイタレーション数を返します。

この計算をすべての平面上で計算します。これでマンデルブロ曲線を描画できます。

簡単に画像を出力するためのフォーマット: PGM

計算したマンデルブロ集合を描画するためには画像フォーマットとして出力する必要があります。画像ファイルとして出力するためにはいくつか方法がありますが、JPEGやPNGなどの画像ファイルはフォーマットが非常に複雑で出力するのが難しいため、もっとシンプルなフォーマットを使用します。ここではPPMというフォーマットを使用します。

PPMというのはPNM（Portable aNyMap）形式のフォーマットの一種で、画像の圧縮などは行わずテキスト形式として表現されます。このため画像サイズは大きくなりますが取り扱いが非常に簡単です。PNMフォーマット形式には3種類存在し、以下の3種類が定義されています。

- PBM（Portable Bit Map）：画像を白黒（ビットマップ）として表示します。
- PGM（Portable GrayMap）：画像をグレイスケールとして表示します。
- PPM（Portable PixMap）：画像をフルカラー形式として表示します。

どれも基本的な形式は一緒で、ファイルの中身は以下のような構成を取ります。

1行目：画像フォーマット。PBM=P1, PGM=P2, PPM=P3

2行目：画像の横幅と縦幅

3行目：ピクセルデータの最大値

4行目以降：ピクセルデータ

今回はマンデルブロ曲線を描くためにPGMを使用し、画像サイズは320x240とします。

また

`mandelbrot()` が返す値の最大値は `log(NMAX)` なのでこれをピクセルデータの最大値とします。

```
P2  
320 240  
9  
// ここから先はデータ
```

PGMファイルはLinuxの環境では簡単に表示できます。画像の編集・表示ツールであるImageMagickの`display`コマンドを使用して表示します。

マンデルブロ曲線描画プログラムをLLVMでコンパイルしてシミュレーション

では、マンデルブロ曲線描画プログラムをコンパイルしてみましょう。`mandelbrot.c` はMYRISCVX

のLLVMでコンパイルしてアセンブリファイルを出力し、それを呼び出すための `mandelbrot_main.c` はGCCでコンパイルするという方式を取り、最後に `ld` でまとめてリンクしたいと思います。また、PGM ファイルの出力は `printf()` を直接呼び出し、標準出力に書き出したものをリダイレクトしてファイルに格納します。

```
$ ${BUILD}/bin/clang -O3 --target=riscv64-unknown-elf mandelbrot.c -c -emit-llvm -o mandelbrot.riscv64.bc
$ ${BUILD}/bin/llvm-dis mandelbrot.riscv64.bc -o mandelbrot.riscv64.bc.ll
$ ${BUILD}/bin/llc -march=myriscvx64 -debug -disable-tail-calls \
  -relocation-model=static \
  -filetype=asm mandelbrot.riscv64.bc \
  -o mandelbrot.myriscvx64.static.S
```

生成された `mandelbrot.myriscvx64.static.S` を見てみましょう。

```
mandelbrot:                                # @mandelbrot

# %bb.0:                                     # %entry
...
$BB0_1:                                     # %for.body
                                             # =>This Inner Loop Header: Depth=1
    fadd.d  f2, f0, f0
    fmadd.d f2, f3, f2, f11
    fmul.d  f3, f3, f3
    fmsub.d f0, f0, f0, f3
    fadd.d  f0, f0, f10
    fmul.d  f3, f0, f2
    fmadd.d f3, f0, f2, f3
    fle.d   x13, f1, f3
    bne     x13, x0, $BB0_3
    j       $BB0_2

$BB0_2:                                     # %for.inc
                                             #   in Loop: Header=$BB0_1 Depth=1
    slli   x13, x11, 32
...
```

メインのループは浮動小数点演算のかたまりです。ループを脱出する条件として `fle.d` による比較を行い、結果を `x13` に格納してそれに基づく条件分岐命令を実行しています。

実行ファイルを生成するために、アセンブリファイルと `main()` の入った `mandelbrot_main.c` をリンクしましょう。`mandelbrot_main.c` のコンパイルにはGCC、リンクにはldの力を借ります^[3]

```
# mandelbrot_main.cをコンパイルしてmandelbrot_main.riscv.oを生成
$ riscv64-unknown-elf-gcc mandelbrot_main.c \
  -o mandelbrot_main.riscv.o -c \
  -mcmode=medany -static -std=gnu99 -O2 -ffast-math -static -lm -lgcc

# mandelbrot.riscv.oとmandelbrot.myriscvx64.static.Sをリンクして実行ファイルを作成
```

```
# 実行ファイル名はmandelbrot.llvm.riscv
$ riscv64-unknown-elf-gcc \
-o mandelbrot.llvm.riscv \
mandelbrot.myriscvx64.static.S \
mandelbrot_main.riscv.o \
-mcmodel=medany -static -std=gnu99 -O2 -ffast-math -static -lm -lgcc
```

このプログラムは実行時間が長いのでさすがにRTLシミュレーションでは実行できません。spikeを使って動作を確認することにします。

```
$ spike pk mandelbrot.llvm.riscv > mandelbrot_myriscvx.pgm
```

シミュレーション結果により、Figure 9のような画像が出力されました。マンデルブロ曲線の描画に成功です。私たちのLLVM実装は正しく浮動小数点演算を演算できています。

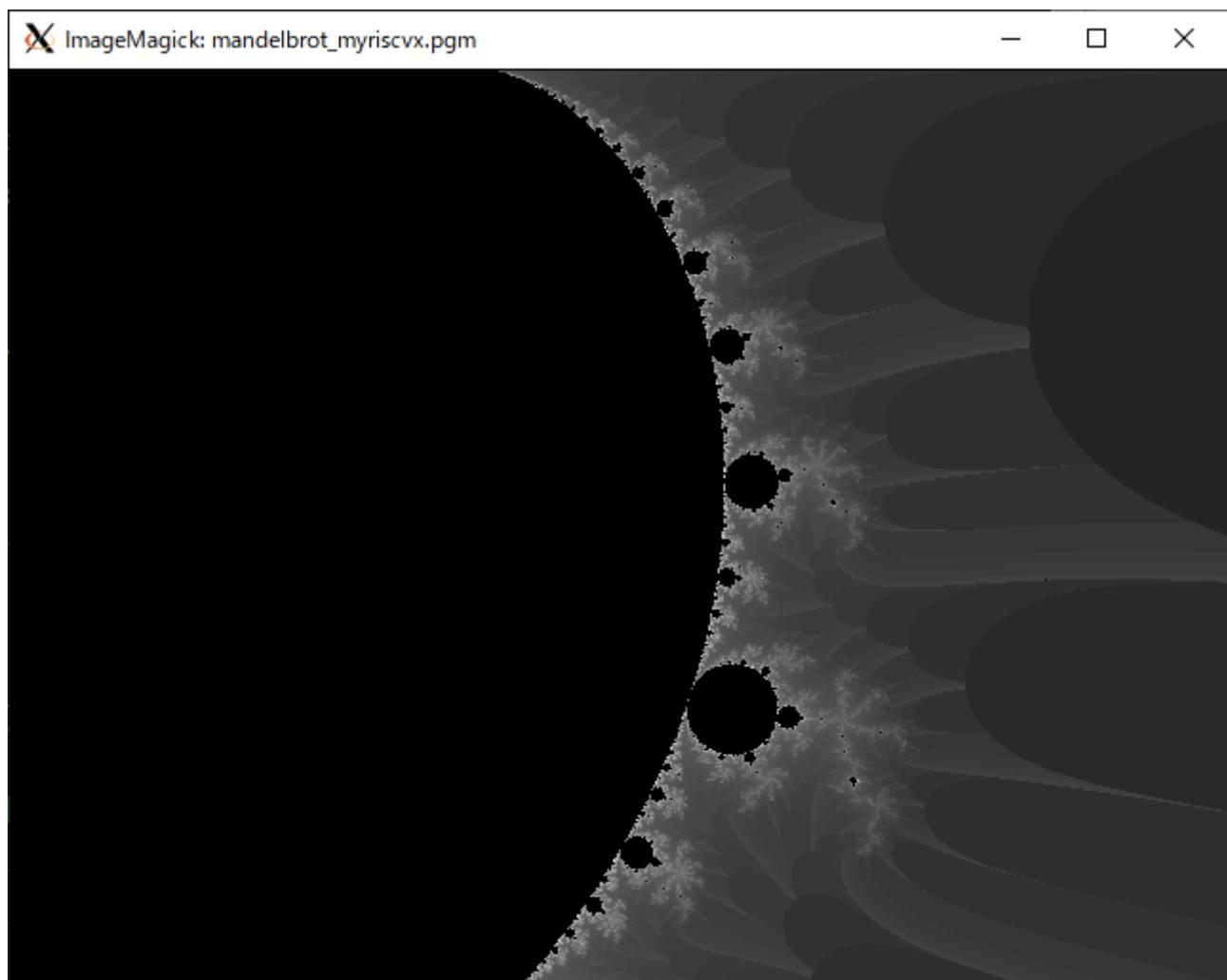


Figure 9. MYRISCVX版LLVMでコンパイルしたマンデルブロ曲線プログラムをシミュレーションして生成した画像

さらに一歩先へ：浮動小数点演算を用いたレイトレースプログラムの実行

浮動小数点演算命令を実装したので、さらに複雑なプログラムのコンパイルとシミュレーションに挑戦します。前節ではマンデルブロ曲線でしたが、さらに発展させてC++で記述されたレイトレースプログラムをコンパイルしシミュレーションに挑戦します。

レイトレースとは、3次元上に配置された物体を2次元上に投影する方式の一種です。レイ（光源）をトレースすることにより、3次元上に配置された物体を問う家するだけでなく、その物体の性質（透過性、屈折率）などに応じて色や明るさなどを再計算することで、光沢や透過性などを表現できるアルゴリズムです。非常に高精度な画像を生成できる一方で、計算量が多く、処理に時間がかかるアルゴリズムとして知られています。

少しだけレイトレースのアルゴリズムについて触れておきましょう（その方がデバッグ時に方針が立てやすいです）。ある3次元空間に、表示すべき物体、視点（人間の目に当たる）、そして光源を配置します。光源からは光が投射され、物体を照らしてその表面がどのような色になるのかが分かり、視点から物体を見たときにどのようにどのような色になるのかが決定されます（Figure 10）。

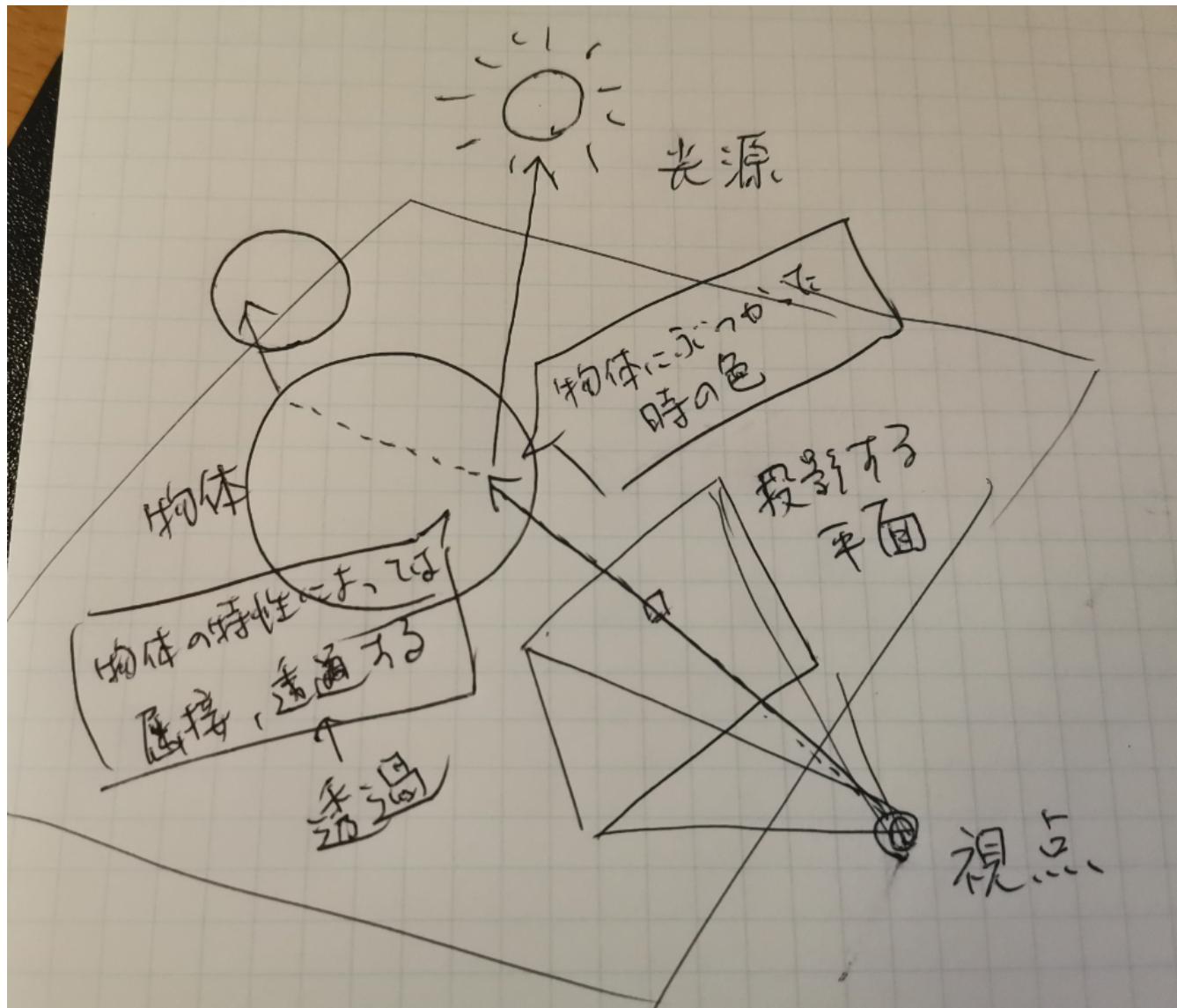


Figure 10. レイトレースアルゴリズムの概略（著者注：すみません綺麗な3D図が描けなかったので手書きしました。。。）

レイトレースでは、2次元平面上に空間上の物体を配置するために、2次元上の各ピクセルについて色情報を計算することになります。あるピクセルの色を決定するために、

1. 視点から対象となるピクセルに対して光線を発射します。

2. 光線を発射してから最初にぶつかる物体を検出し、ぶつかる交点を計算します（これを位置 x とします）。
3. 交点から光源（明かり）まで光線を発射します。もしこの光線がさらに別の物体にぶつかれば、位置 x は影になっていることが分かります。光源までの情報に基づいて位置 x の色を計算します。
4. さらに、もしぶつかった物体の性質上、反射する（鏡のような物体）ような性質であったり、屈折する（ガラスのような物体）ような性質であった場合を考えます。物体の性質に応じて物体の性質に基づき次の光線を生成し、同じようにして位置 x から光線を発射して色情報を再計算します。これは再帰的処理として表現されます。
5. 4.を再帰的に実行し続けるといつまでたっても終わらないので、ある程度十分に実行すると再帰的処理を中断して、反射と屈折に基づく交点 x の色情報を計算します。
6. 最後に、反射、屈折に基づく色情報を混ぜ合わせて最終的な交点の色情報を計算し、平面上のピクセルの値とします。

これを一から実装するのは大変なので、 <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/> に掲載されているレイトレースのプログラムを参考に実装しました。ただしLLVMでコンパイルを通すため、少しだけ修正を加えています^[4]。

このプログラムをビルドしてみます。今回はC++のソースファイルからアセンブリファイルの出力までをすべてLLVMで実行しています。リンクのみ `ld` で行いました。

```
$ ${BUILD}/bin/clang -O3 raytracer.cpp -c -emit-llvm -o raytracer.riscv64.bc
$ ${BUILD}/bin/llvm-dis raytracer.riscv64.bc -o raytracer.riscv64.bc.ll
$ ${BUILD}/bin/llc -march=myriscvx64           -debug -disable-tail-calls \
  -relocation-model=static -filetype=asm raytracer.riscv64.bc \
  -o raytracer.myriscvx64.static.S
# リンク処理
$ riscv64-unknown-elf-g++ -o raytracer.llvm.riscv raytracer.myriscvx64.static.S \
  -mcmodel=medany -static -std=gnu99 -O2 -ffast-math -static -lm -lgcc
```

`raytracer.myriscvx64.static.S` は非常に大きなアセンブリファイルですが、一応要点だけ眺めておきましょう。

- `raytracer.myriscvx64.static.S`

```
main:                                # @main

# %bb.0:                                # %if.else.i.i157
    addi    x2, x2, -144
    sd     x1, 136(x2)          # 8-byte Folded Spill
    sd     x2, 128(x2)          # 8-byte Folded Spill
...
$BB3_48:                                # %invoke.cont118
    sd     x10, 0(x11)
    addi   x10, x2, 16
    call   _Z6renderRKSt6vectorI6SphereSaIS0_EE
# %bb.49:                                # %invoke.cont127
    ld     x10, 16(x2)
```

```
beq x10, x0, $BB3_51
```

`_Z6renderRKSt6vectorI6SphereSaIS0_EE` が `render()` に相当します。C++ではオーバーロードが許されているため、アセンブリレベルでは引数の種類などに応じてこのように関数のラベルに対して接頭語や接尾語が付加されています。

`render()` は引数として物体の情報、光源の情報などを渡して実行されます。

```
_Z6renderRKSt6vectorI6SphereSaIS0_EE: # @_Z6renderRKSt6vectorI6SphereSaIS0_EE

# %bb.0:                                # %entry
    addi    x2, x2, -192
    sd     x1, 184(x2)                  # 8-byte Folded Spill
    sd     x2, 176(x2)                  # 8-byte Folded Spill
...
    addi    x14, x24, 0
    call    _Z5traceRK4Vec3IfES2_RKSt6vectorI6SphereSaIS4_EERKiPS0_
    flw   f2, 48(x2)                  # 4-byte Folded Reload
    addi    x24, x24, 12
```

`trace()` は画像のサイズ分だけ呼び出されます。たとえばこのプログラムだと 640×480 だけ `trace()` が実行されます。

```
.type   _Z5traceRK4Vec3IfES2_RKSt6vectorI6SphereSaIS4_EERKiPS0_,@function
_Z5traceRK4Vec3IfES2_RKSt6vectorI6SphereSaIS4_EERKiPS0_: #
 @_Z5traceRK4Vec3IfES2_RKSt6vectorI6SphereSaIS4_EERKiPS0_

# %bb.0:                                # %entry
    addi    x2, x2, -272
    sd     x1, 264(x2)                  # 8-byte Folded Spill
    sd     x2, 256(x2)                  # 8-byte Folded Spill
...
    addi    x13, x23, 0
    call    _Z5traceRK4Vec3IfES2_RKSt6vectorI6SphereSaIS4_EERKiPS0_
    sw     x19, 128(x2)
    sd     x19, 120(x2)
...
    addi    x14, x22, 0
    call    _Z5traceRK4Vec3IfES2_RKSt6vectorI6SphereSaIS4_EERKiPS0_
    flw   f2, 0(x26)
    flw   f1, 120(x2)
...
```

アルゴリズムの概要で説明したとおり、`trace()` は再帰的に呼び出されます。これにより最終的な対象ピクセルの色情報を計算して、`printf()` で出力します。出力形式はマンデルブロ曲線の時にも使用したPNM形式の一つであるPPMを使用しています。PPM形式はフルカラーで出力できます。

ではSpikeでシミュレーションしてみましょう。

```
$ spike pk raytracer.llvm.riscv > raytracer.llvm.riscv.ppm
```

出力結果はFigure 11のようになりました。上手く行ったようです。私たちのLLVM実装は、レイトレースを正しく実行できました！

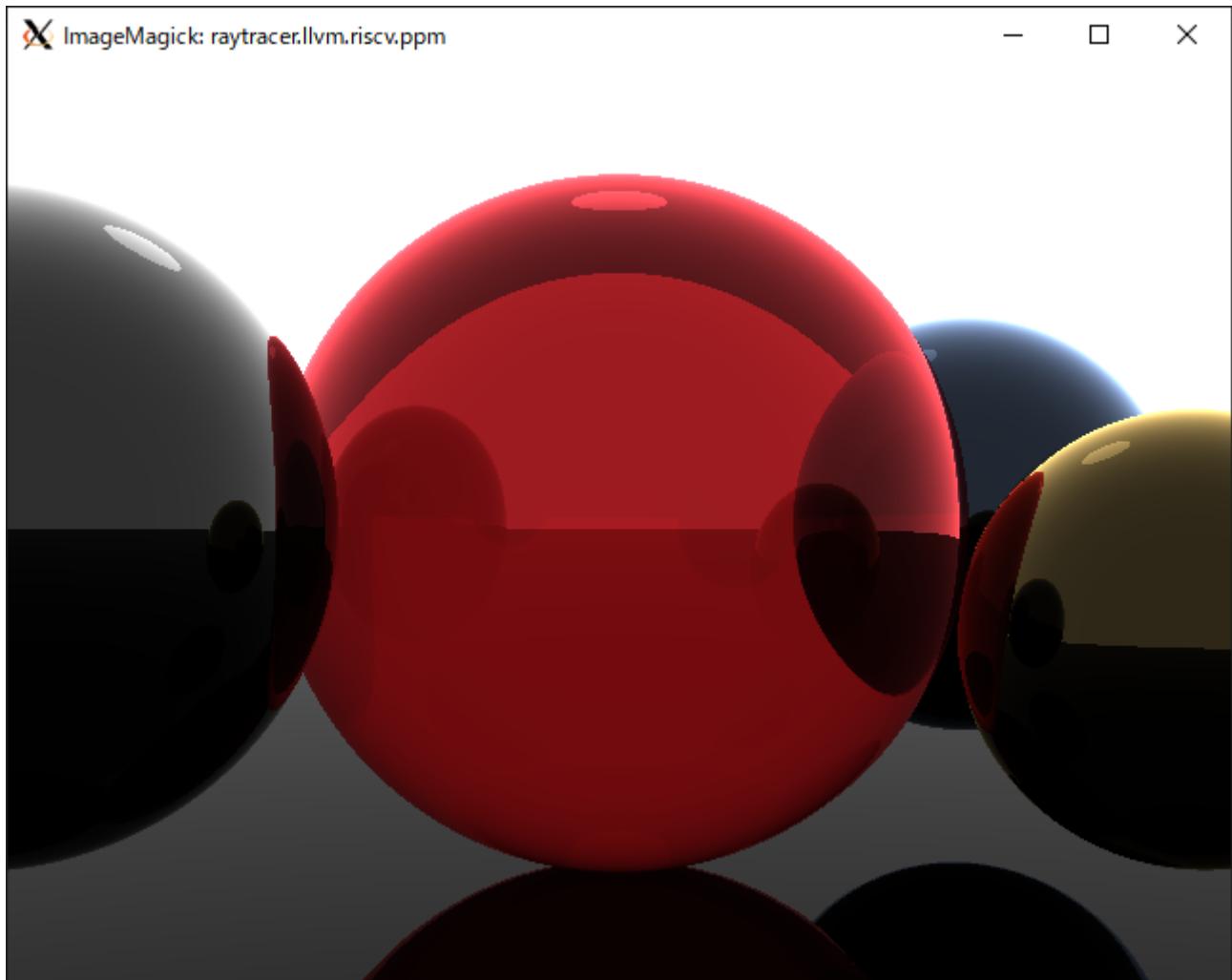


Figure 11. LLVMでコンパイルしたライトレースプログラムをSpikeでシミュレーションし生成したライトレース画像

[1] RISC-Vの浮動小数点命令では半精度浮動小数点命令は定義されていません。しかしRISC-Vのベクトル拡張命令では、半精度浮動小数点演算をサポートできます。

[2] ただし $E = 0$ の場合は非正規化数として正規化しない表現が許されます

[3] LLVMですべてをコンパイルしても良いですが、`mandelbrot()` が正しくコンパイルされていることを確認するために `mandelbrot.c` のみをLLVMでコンパイルして実行しました。

[4] このプログラムでは戻り値として3次元座標を示す3つのデータを返すようにしていますが、私たちの実装したMYRISCVXでは最大で2個までの値しか戻り値で返すことができないため、引数のポインタで渡すようにプログラムの修正を加えています。