# 付録1. 関数呼び出しのバリエーションと高度な機能

## Table of Contents



これまでの実装により、関数呼び出しから基本的な算術演算、メモリアクセスなどMYRISCVXの基本的な命令をサポートし、簡単なプログラムを生成できるようになりました。LLVMにはこれまでに紹介したIR以外にも、さまざまなIRが用意されています。また、より本格的なプログラムをコンパイルするためにはさらなる実装が必要になります。機能のサポートだけでなく、より最適化されたアセンブリ命令を生成するための機能を追加していきたいです。MYRISCVXのLLVM実装に新たな機能を追加して、より多くの構文をサポートし、より最適化された命令を生成するための方法について学んでいきましょう。

# [LowLayer] 関数の引数として構造体を使用する

今までの関数処理の中で、引数は基本的にポインタか値渡し、そして値も何らかの型の値を1つずつ渡していくという形式でした。しかし、C言語では構造体などの複数の要素をまとめた型を渡すことができます。また、C言語では構造体の値をそのまま値渡しで引数として渡すことができます。

下記のプログラムでは、 `func` に構造体 `struct S elem` を値渡ししています。つまり、 `func` 内で `struct S elem` の変更しても、その変更の結果は関数の呼び出し側にの値に影響を与えません。 ポインタではないのでアセンブリ的には、Figure 1 のように引数の要素をひとつひとつコピーして、関数呼び出され側に値を渡してやる必要があります。

- program/appendix_1/func_struct_call.c

```
#define STRUCT_SIZE (17)

struct S
{
  int x[STRUCT_SIZE];
};

int func (struct S elem) {
  int total = 0;
  for(int i = 0; i < STRUCT_SIZE; i++) {
    total += elem.x[i];
  }
  return total;
}

int call_func () {
  struct S elem;
  for (int i = 0; i < STRUCT_SIZE; i++) {
    elem.x[i] = i;
  }
  return func (elem);
}
```

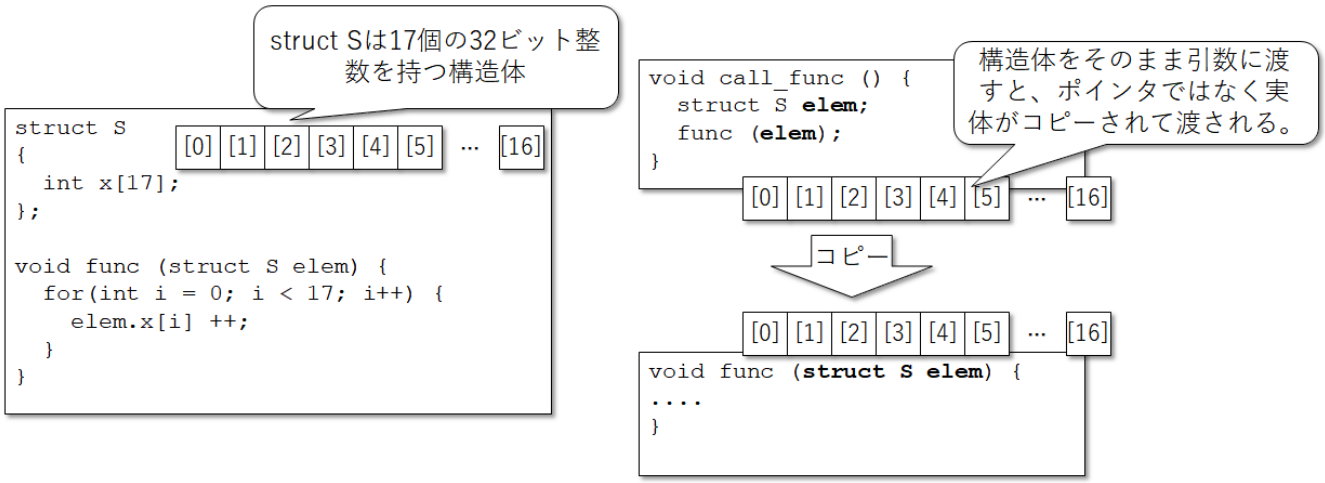



*Figure 1.* 関数の引数として構造体を値渡しする仕組み。

上記の例では、実際に渡さなければならない値は `x[0]~x[16]` の17個です。 しかしMYRISCVXのABIでは引数渡しで使用できるレジスタの数はa0-a7の8個までです。 この場合、どのようにして構造体の値を渡していけばよいでしょうか。

# [RISCV] 値渡しの仕組み

RISC-Vの構造体の値渡しの仕組みについては、まずは仕様書を参照してみましょう。 RISC-VのABIについては”RISC-V ELF psABI specification”を見てみます。 この仕様書はGitHubに公開されています。

仕様書の中で、`Procedure Calling Convention'' -> Integer Calling Convention"`を参照すると、以下のような説明があります。

> Aggregates whose total size is no more than XLEN bits are passed in a register, with the fields laid out as though they were passed in memory. If no register is available, the aggregate is passed on the stack. Aggregates whose total size is no more than 2□XLEN bits are passed in a pair of registers; if only one register is available, the first half is passed in a register and the second half is passed on the stack. If no registers are available, the aggregate is passed on the stack. Bits unused due to padding, and bits past the end of an aggregate whose size in bits is not divisible by XLEN, are undefined.

aggregatesは集合体、つまり構造体もaggregatesの1つと考えることができます。これによると、XLENよりも小さな構造体は1つの引数レジスタを経由して渡されます。もし使用可能なレジスタが無い場合にはメモリを経由して渡されます。　またlatexmath:[$2\times$]XLENビットを超えないサイズの構造体も2つのレジスタを経由して渡されます。　　こちらも、レジスタが足りない場合はスタックを経由して渡されます。少しややこしいので纏めておきましょう。

- RV32（XLEN=32）：64ビット以下のサイズの構造体はレジスタを経由して渡されます。それ以外はメモリを経由して渡されます。
- RV64（XLEN=64）：128ビット以下のサイズの構造体はレジスタを経由して渡されます。それ以外はメモリを経由して渡されます。

| 構造体の合計ビットサイズ | ⇐ 32ビット | ⇐ 64ビット | ⇐ 128ビット | > 128ビット |
|---|---|---|---|---|
| RV32 | 1つの引数レジスタを使用する | 2つの引数レジスタを使用する | メモリを経由する | メモリを経由する |
| RV64 | 1つの引数レジスタを使用する | 1つの引数レジスタを使用する | 2つの引数レジスタを使用する | メモリを経由する |

これを確かめるために以下のようなプログラムを作って、GCCでコンパイルしてみましょう。以下のようなプログラムを作ります。

- program/appendix_1/func_struct_simple/func_struct_simple.h

```
#include <stdint.h>

struct S32
{
  uint8_t  a, b;
  uint16_t c;
};

struct S64
{
  struct S32 s32;
  uint32_t   d;
};

struct S128
{
  struct S64 s64;
  int64_t   e;
};

struct S256
{
  struct S128 s128;
  uint32_t    f[4];
};
```

それぞれサイズが32ビット・64ビット・128ビット・256ビットの構造体を用意しました。構造体の中には8ビットから64ビットのデータを格納しています（ Figure 2 ）。

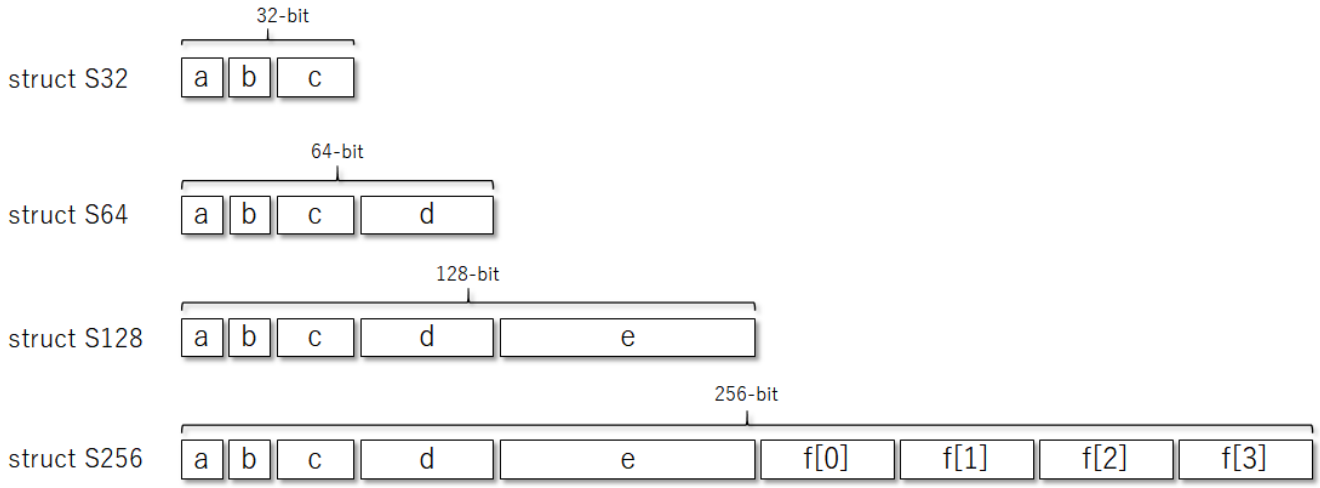



*Figure 2.* 構造体の値渡しについて見るために*4*種類の構造体についてテストする。*S32*はメンバ変数の合計ビット数が*32*ビットの構造体、*S256*はメンバ変数の合計ビット数が*256*ビットの構造体

# [LowLayer] 構造体の値渡しを受け取る関数の仕組み

さて、これらの構造体の合計値を計算するプログラムを用意します。つまりこれらの構造体を値渡しの引数として受け取るプログラムを用意します。

- program/appendix_1/func_struct_simple/func_struct_simple.c

```
#include "func_struct_simple.h"

uint64_t func_S32 (struct S32 elem) {
  return elem.a + elem.b + elem.c;
}

uint64_t func_S64 (struct S64 elem) {
  return func_S32(elem.s32) + elem.d;
}

uint64_t func_S128 (struct S128 elem) {
  return func_S64(elem.s64) + elem.e;
}

uint64_t func_S256 (struct S256 elem) {
  return func_S128(elem.s128) + elem.f[0] + elem.f[1] + elem.f[2] + elem.f[3];
}
```

まずは生成されるLLVM　IRを見てみましょう。RV32とRV64でコンパイルして、違いは発生するでしょうか。

```
# RV32でのclangによるLLVM IR生成
$ ${BUILD}/bin/clang  -O1 --target=riscv32-unknown-elf func_struct_simple.c -c -emit
-llvm \
    -o func_struct_simple.riscv32.static.bc
$ ${BUILD}/bin/llvm-dis func_struct_simple.riscv32.static.bc -o \
    func_struct_simple.riscv32.static.bc.ll

# RV64でのclangによるLLVM IR生成
$ ${BUILD}/bin/clang  -O1 --target=riscv64-unknown-elf func_struct_simple.c -c -emit
-llvm \
    -o func_struct_simple.riscv64.static.bc
$ ${BUILD}/bin/llvm-dis func_struct_simple.riscv64.static.bc -o \
    func_struct_simple.riscv64.static.bc.ll
```

- func_struct_simple.riscv32.static.bc.ll

```
define dso_local i64 @func_S32(i32 %elem.coerce) local_unnamed_addr #0 {
entry:
  %elem.sroa.2.0.extract.shift = lshr i32 %elem.coerce, 8
  %elem.sroa.3.0.extract.shift = lshr i32 %elem.coerce, 16
...

define dso_local i64 @func_S64([2 x i32] %elem.coerce) local_unnamed_addr #0 {
entry:
  %elem.coerce.fca.0.extract = extractvalue [2 x i32] %elem.coerce, 0
  %elem.coerce.fca.1.extract = extractvalue [2 x i32] %elem.coerce, 1
...

define dso_local i64 @func_S128(%struct.S128* nocapture readonly %elem)
local_unnamed_addr #1 {
entry:
  %.elt = bitcast %struct.S128* %elem to i32*
  %.unpack = load i32, i32* %.elt, align 8
  %0 = insertvalue [2 x i32] undef, i32 %.unpack, 0
  %1 = getelementptr inbounds %struct.S128, %struct.S128* %elem, i32 0, i32 0, i32 1
...

define dso_local i64 @func_S256(%struct.S256* nocapture readonly %elem)
local_unnamed_addr #2 {
entry:
  %byval-temp = alloca %struct.S128, align 8
  %0 = getelementptr inbounds %struct.S128, %struct.S128* %byval-temp, i32 0, i32 0,
i32 0, i32 0
  call void @llvm.lifetime.start.p0i8(i64 16, i8* nonnull %0) #4
  %1 = getelementptr inbounds %struct.S256, %struct.S256* %elem, i32 0, i32 0, i32 0,
i32 0, i32 0
...
```

少し長いLLVM IRが出てきましたが、ここで見なければならないのは引数の受け取り方です。つまり、

- `func_S32()`, `func_S64()` は引数をレジスタから受け取ってそのまま演算するためのIRが挿入されている。
- `func_S128()`, `func_S256()` は引数をポインタ経由で受け取り、ポインタからデータをロードして演算をしている。

ということです。これはRV32のABIに従っていると言って良いでしょう。RV32（XLEN=32）の場合はXLEN=32までのデータはレジスタを経由して渡されています。

次はRV64でLLVM IRを確認してみます。結果に違いは現れるでしょうか。

- `func_struct_simple.riscv64.static.bc.ll`

```
define dso_local signext i32 @func_S32(i64 %elem.coerce) local_unnamed_addr #0 {
entry:
  %tmp.0.extract.trunc = trunc i64 %elem.coerce to i32
  %elem.sroa.2.0.extract.shift = lshr i32 %tmp.0.extract.trunc, 8
  %elem.sroa.3.0.extract.shift = lshr i32 %tmp.0.extract.trunc, 16
...

define dso_local signext i32 @func_S64(i64 %elem.coerce) local_unnamed_addr #0 {
entry:
  %elem.sroa.2.0.extract.shift = lshr i64 %elem.coerce, 32
  %elem.sroa.2.0.extract.trunc = trunc i64 %elem.sroa.2.0.extract.shift to i32

define dso_local signext i32 @func_S128([2 x i64] %elem.coerce) local_unnamed_addr #0
{
entry:
  %elem.coerce.fca.0.extract = extractvalue [2 x i64] %elem.coerce, 0
  %elem.coerce.fca.1.extract = extractvalue [2 x i64] %elem.coerce, 1

...
define dso_local signext i32 @func_S256(%struct.S256* nocapture readonly %elem)
local_unnamed_addr #1 {
entry:
  %.elt = bitcast %struct.S256* %elem to i64*
  %.unpack = load i64, i64* %.elt, align 8
  %0 = insertvalue [2 x i64] undef, i64 %.unpack, 0
  %1 = getelementptr inbounds %struct.S256, %struct.S256* %elem, i64 0, i32 0, i32 1
  %.unpack11 = load i64, i64* %1, align 8
  %2 = insertvalue [2 x i64] %0, i64 %.unpack11, 1
```

大きな違いは、 `func_S128()` のIRです。 RV32の時は引数にポインタを使ってデータを渡していましたが、今回はポインタが登場せず値をそのまま渡しています。 XLEN=64なので、128ビットまでのデータはレジスタを経由して値渡しをして良いのです。

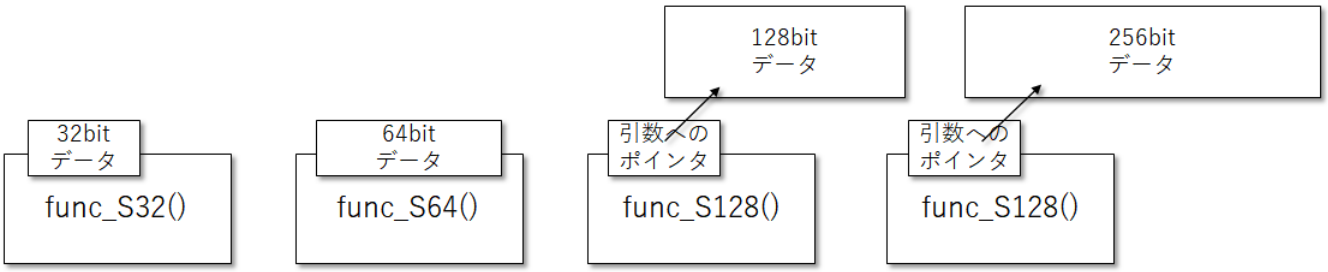



*Figure 3. RV32*における関数の値渡し。*64*ビットまでのデータサイズはそのまま値を渡すが、それ以上のデータサイズを渡す時は、引数データへのポインタを渡す。

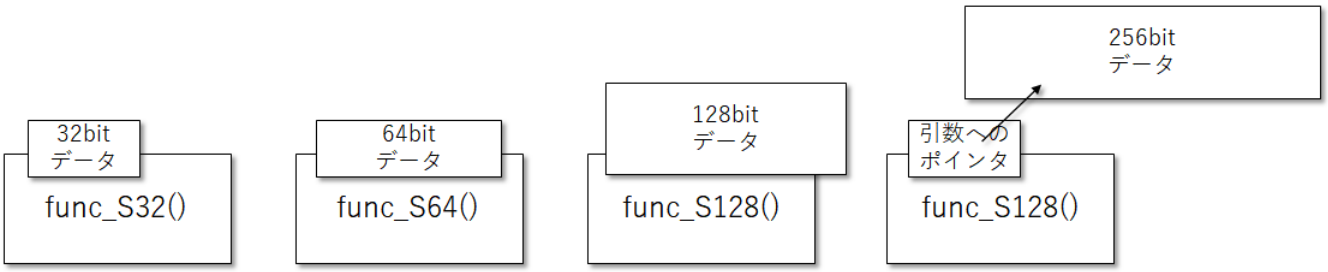



*Figure 4. RV64*における関数の値渡し。*128*ビットまでのデータサイズはそのまま値を渡すが、それ以上のデータサイズを渡す時は、引数データへのポインタを渡す。

# [LowLayer] 構造体を値渡しする関数の仕組み

さて、構造体を受け取る側の関数の動作については分かってきました。 では、つぎに構造体を値渡しして関数コールを行う、呼び出し側はどのような仕組みで関数を呼べば良いでしょうか。 呼び出され側のルールは理解したので、簡単に想像ができます。

- RV32の場合：64ビットまでの値はレジスタやスタックを経由してそのまま値を渡す。それよりも大きなデータについてはメモリ中に構造体データを格納し、引数にはその構造体データの位置を指したポインタを渡す。
- RV64の場合：128ビットまでの値はレジスタやスタックを経由してそのまま値を渡す。それよりも大きなデータについてはメモリ中に構造体データを格納し、引数にはその構造体データの位置を指したポインタを渡す。

受け取り側と同じく、これをLLVM　IRで確認してみましょう。受け取り側で使った　`func_S64()`　、`func_S128()`、 `func_S256()` にはそれぞれ1つ小さな構造体を計算するための呼び出し関数処理が含まれているので、それを見てみます。　もうひとつ、S256を値渡しの引数として呼び出す関数を用意しておきます。

- `program/appendix_1/func_struct_simple/func_struct_simple.c`

```
int func_S256_caller()
{
  struct S256 elem;
  elem.s128.s64.s32.a = 100;
  elem.s128.s64.s32.b = 200;
  elem.s128.s64.s32.c = 300;
  elem.s128.s64.d     = 400;
  elem.s128.e         = 0xdeadbeef;
  elem.f[0]             = 600;
  elem.f[1]             = 700;
  elem.f[2]             = 800;
  elem.f[3]             = 900;

  return func_S256(elem);
}
```

まずはRV32のLLVM IRを見てみます。

- func_struct_simple.riscv32.static.bc.ll

```
define dso_local i64 @func_S64([2 x i32] %elem.coerce) local_unnamed_addr #0 {
entry:
  %elem.coerce.fca.0.extract = extractvalue [2 x i32] %elem.coerce, 0
  %elem.coerce.fca.1.extract = extractvalue [2 x i32] %elem.coerce, 1
  ;; func_S32()の呼び出し部分。引数としてi32型の値をそのまま渡している。
  %call = call i32 @func_S32(i32 %elem.coerce.fca.0.extract)

define dso_local i64 @func_S128(%struct.S128* nocapture readonly %elem)
local_unnamed_addr #1 {
entry:
...
  %2 = insertvalue [2 x i32] %0, i32 %.unpack3, 1
  ;; func_S64()の呼び出し部分。引数としてi32型の値を2つそのまま渡している。
  %call = call i32 @func_S64([2 x i32] %2)

define dso_local i64 @func_S256(%struct.S256* nocapture readonly %elem)
local_unnamed_addr #2 {
entry:
  %byval-temp = alloca %struct.S256, align 8
...
  %1 = getelementptr inbounds %struct.S256, %struct.S256* %elem, i32 0, i32 0, i32 0,
i32 0, i32 0
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* nonnull align 8 dereferenceable(16) %0, i8*
nonnull align 8 dereferenceable(16) %1, i32 16, i1 false), !tbaa.struct !12
  ;; func_S128()の呼び出し部分。引数としてstruct.S128のポインタを渡している。
  %call = call i32 @func_S128(%struct.S128* nonnull %byval-temp)
...

define dso_local i64 @func_S256_caller() local_unnamed_addr #2 {
entry:
  %byval-temp = alloca %struct.S256, align 8
...
  store i32 800, i32* %elem.sroa.10.0..sroa_idx26, align 8, !tbaa.struct !17
  %elem.sroa.11.0..sroa_idx28 = getelementptr inbounds %struct.S256, %struct.S256*
%byval-temp, i32 0, i32 1, i32 3
  store i32 900, i32* %elem.sroa.11.0..sroa_idx28, align 4, !tbaa.struct !17
  ;; func_S256()の呼び出し部分。引数としてstruct.S256のポインタを渡している。
  %call = call i32 @func_S256(%struct.S256* nonnull %byval-temp)
  call void @llvm.lifetime.end.p0i8(i64 32, i8* nonnull %0) #4
```

まず、`func_S32()`, `func_S64()` の呼び出しについては引数にそのまま値を渡しています。 しかし `func_S128()` および `func_S256()` の呼び出しについては一度 `byval-temp` という構造体のインスタンスが作られています。 そこにデータをコピーしてから `byval-temp` のポインタが引数として渡されていることが分かります。 当然ですが、これもABIのルール通りです。

では次にRV64の方もチェックしてみます。

- `func_struct_simple.riscv64.static.bc.ll`

```
define dso_local i64 @func_S64([2 x i32] %elem.coerce) local_unnamed_addr #0 {
entry:
  %tmp.0.insert.ext = and i64 %elem.coerce, 4294967295
  ;; func_S32()の呼び出し部分。引数としてi64型の値をそのまま渡している。
  %call = call signext i32 @func_S32(i64 %tmp.0.insert.ext)

define dso_local signext i32 @func_S128([2 x i64] %elem.coerce) local_unnamed_addr #0
{
entry:
  %elem.coerce.fca.0.extract = extractvalue [2 x i64] %elem.coerce, 0
  %elem.coerce.fca.1.extract = extractvalue [2 x i64] %elem.coerce, 1
  ;; func_S64()の呼び出し部分。引数としてi64型の値を2つそのまま渡している。
  %call = call signext i32 @func_S64(i64 %elem.coerce.fca.0.extract)

define dso_local signext i32 @func_S256(%struct.S256* nocapture readonly %elem)
local_unnamed_addr #1 {
entry:
  %.unpack11 = load i64, i64* %1, align 8
  %2 = insertvalue [2 x i64] %0, i64 %.unpack11, 1
  ;; func_S128()の呼び出し部分。引数としてi64型の値を2つ渡している。
  %call = call signext i32 @func_S128([2 x i64] %2)

define dso_local signext i32 @func_S256_caller() local_unnamed_addr #2 {
entry:
  %byval-temp = alloca %struct.S256, align 8
...
  store i32 700, i32* %elem.sroa.9.0..sroa_idx24, align 4, !tbaa.struct !7
  %elem.sroa.10.0..sroa_idx26 = getelementptr inbounds %struct.S256, %struct.S256*
%byval-temp, i64 0, i32 1, i64 2
  store i32 800, i32* %elem.sroa.10.0..sroa_idx26, align 8, !tbaa.struct !7
  %elem.sroa.11.0..sroa_idx28 = getelementptr inbounds %struct.S256, %struct.S256*
%byval-temp, i64 0, i32 1, i64 3
  store i32 900, i32* %elem.sroa.11.0..sroa_idx28, align 4, !tbaa.struct !7
  ;; func_S256()の呼び出し部分。引数としてstruct.S256のポインタを渡している。
  %call = call signext i32 @func_S256(%struct.S256* nonnull %byval-temp)
```

呼び出され側と同様に、違いが表れるのは `func_S128()` の呼び出し側です。 `func_S32()` から `func_S128()` までは値を引数として直接呼び出しています。 `func_S256()` の呼び出しについては一度 `byval-temp` という構造体のインスタンスが作られています。 そこにデータをコピーしてから `byval-temp` のポインタが引数として渡されていることが分かります。 当然ですが、これもABIのルール通りです。

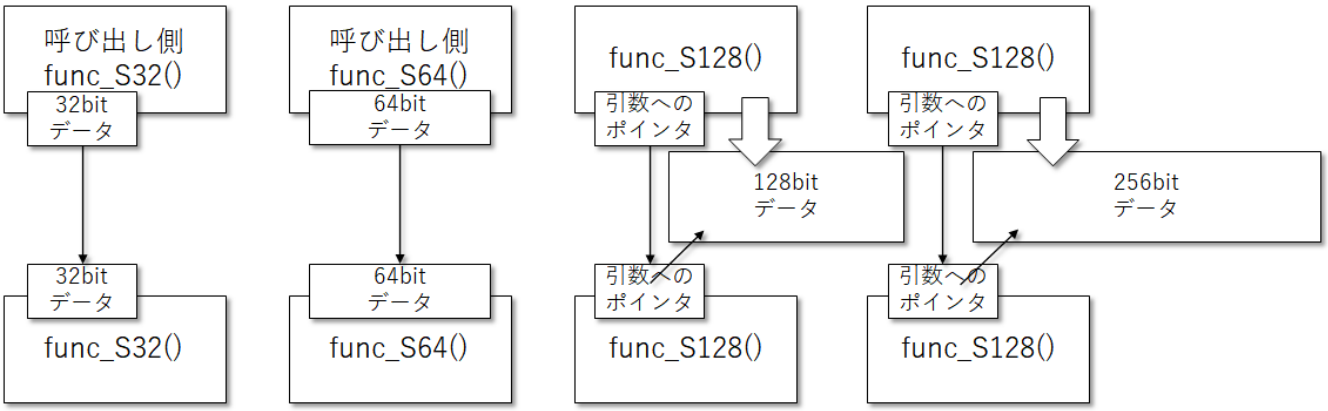



*Figure 5. RV32*における関数の値渡し。*64*ビットまでは値をそのまま引数として渡す。それ以上のデータサイズを渡す時は、引数データへのポインタを渡す。値渡しをして関数を呼び出す側は、引数として渡したい値をメモリ中に確保し、その場所へのポインタを関数に引数として渡す。

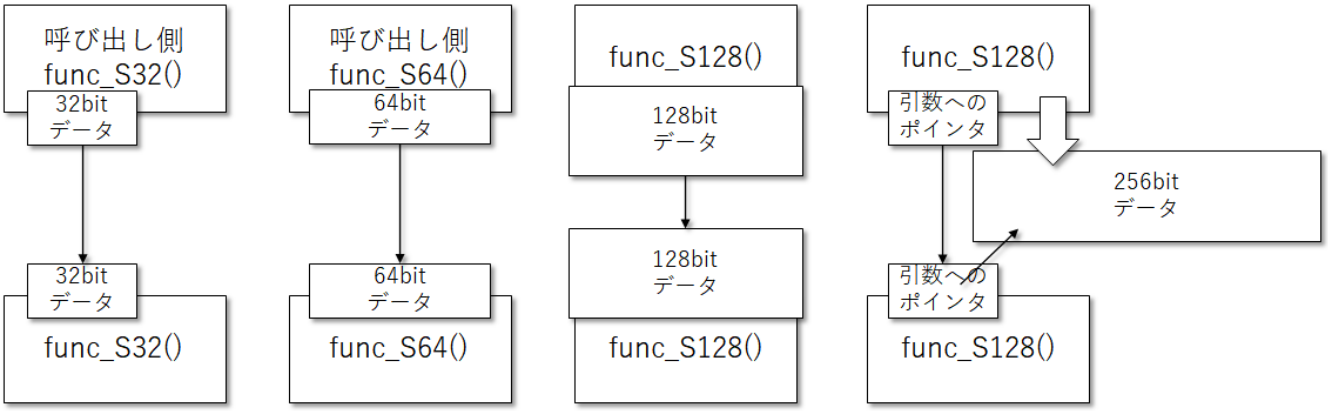



*Figure 6. RV64*における関数の値渡し。*128*ビットまでは値をそのまま引数として渡す。それ以上のデータサイズを渡す時は、引数データへのポインタを渡す。値渡しをして関数を呼び出す側は、引数として渡したい値をメモリ中に確保し、その場所へのポインタを関数に引数として渡す。

というわけで、実はClangを経由してLLVM IRを出力する時点でABIに則って引数を渡す仕組みが整っていることが分かりました。 構造体の値渡しについてはこれ以上何か追加で実装する必要はないので、上記のサンプルプログラムをコンパイルしてテストしてみましょう。

`func_S32()` から `func_S256()` までの構造体を扱うために以下のようなテストコードを作ります。 このテストコード自体はRISC-VのGCCでコンパイルされます。　リンクの時点でLLVMを使ってコンパイルした `func_S32()` から `func_S256()` のMYRISCVXアセンブリコードと接続してバイナリを作ります。

- `program/appendix_1/func_struct_simple/test_func_struct_simple.c`

```
#include <stdio.h>
#include "func_struct_simple.h"

extern int func_S32 (struct S32 elem);
extern int func_S64 (struct S64 elem);
extern int func_S128 (struct S128 elem);
extern int func_S256 (struct S256 elem);

int main()
{

  struct S256 s256;
  s256.s128.s64.s32.a = 100;
  s256.s128.s64.s32.b = 200;
  s256.s128.s64.s32.c = 300;
  s256.s128.s64.d     = 400;
  s256.s128.e         = 500;
  s256.f[0]           = 600;
  s256.f[1]           = 700;
  s256.f[2]           = 800;
  s256.f[3]           = 900;

  printf("total_S32  = %d\n", func_S32 (s256.s128.s64.s32));
  printf("total_S64  = %d\n", func_S64 (s256.s128.s64));
  printf("total_S128 = %d\n", func_S128(s256.s128));
  printf("total_S256 = %d\n", func_S256(s256));

  return 0;
}
```

`S256` 型の構造体のインスタンス `s256` を用意し、それぞれ値初期値を挿入しています。 `func_S32()` は `s256.s128.s64.s32` を渡しているので答えは `100+200+300= 600` になるはずです。 同様に `func_S64()` は `600+400 = 1000` 、 `func_S128()` は `1000 + 500 = 1500` 、 `func_S256()` は `1500+600+700+800+900 = 4500` となるはずです。

コンパイルしてみます。まずはRV32から行きます。

```
$ ${BUILD}/bin/clang  -O1 --target=riscv32-unknown-elf func_struct_simple.c -c -emit
-llvm \
    -o func_struct_simple.riscv32.static.bc
$ ${BUILD}/bin/llc -march=myriscvx32      --debug -disable-tail-calls -relocation
-model=static \
    -filetype=asm func_struct_simple.riscv32.static.bc \
    -o func_struct_simple.myriscvx32.static.S

# test_func_struct_simple.c と func_struct_simple.myriscvx32.static.medany.S を
# riscv-gccでコンパイルして1つのバイナリを作成。
riscv32-unknown-elf-gcc -march=rv32g -DPREALLOCATE=1 -static -mcmodel=medany
-std=gnu99 \
    -O2 -ffast-math -lm -lgcc \
    test_func_struct_simple.c \
    func_struct_simple.myriscvx32.static.medany.S \
    -o test_func_struct_simple_rv32

# 結果確認のために逆アセンブル
$ riscv32-unknown-elf-objdump -D test_func_struct_simple_rv32 >
test_func_struct_simple_rv32.dmp
```

無事にコンパイルが終了すると、結果確認のために `test_func_struct_simple_rv32.dmp` を確認します。

まずは引数を受け取る側からです。

## RV32における `func_S32(), func_S64()` の引数を受け取るアセンブリ命令

`func_S32` と `func_S64` はほぼ同じ仕組みなので `func_S32` のみ結果を示します。

- `func_struct_simple.myriscvx32.static.medany.S`

```
func_S32:                               # @func_S32
        andi    x11, x10, 255       # 8ビットx2 + 16ビットの値が
x10にパックされているので分解している。
        srli    x12, x10, 16
        add     x11, x12, x11
        srli    x10, x10, 8
        andi    x10, x10, 255
        add     x10, x11, x10
        ret

func_S64:                               # @func_S64
        addi    x2, x2, -16
        sw      x1, 12(x2)              # 4-byte Folded Spill
        sw      x2, 8(x2)               # 4-byte Folded Spill
        sw      x9, 4(x2)               # 4-byte Folded Spill
        addi    x9, x11, 0              # func_S32に向けて
x10で受け取った値渡しの引数をそのまま渡している
        call    func_S32
        add     x10, x10, x9
        lw      x9, 4(x2)               # 4-byte Folded Reload
        lw      x2, 8(x2)               # 4-byte Folded Reload
        lw      x1, 12(x2)              # 4-byte Folded Reload
        addi    x2, x2, 16
        ret
```

32ビットのデータ `sturct S32` は1つのレジスタ `x10` を経由して渡されました。 8ビットx2と16ビットの値が1つのレジスタにパックされているので、論理演算とシフト演算を使用して分解してから計算しています。

## RV32における `func_S128(), func_S256()` の引数を受け取るアセンブリ命令

`func_S128` と `func_S256` はほぼ同じ仕組みなので `func_S128` のみ結果を示します。

- `func_struct_simple.myriscvx32.static.medany.S`

```
func_S128:                              # @func_S128
        addi    x2, x2, -16
        sw      x1, 12(x2)              # レジスタ退避
        sw      x2, 8(x2)               # レジスタ退避
        sw      x9, 4(x2)               # レジスタ退避
        addi    x9, x10, 0              # x10にはS128構造体引数のポインタが入っている
        lw      x11, 4(x9)              # func_S64()のための引数はx10と
x11に設定される。
        lw      x10, 0(x9)
        call    func_S64                # func_S64()を呼び出して計算した後、elem.e
        lw      x11, 8(x9)
        add     x10, x10, x11           # elem.eの下位32ビットを加算
        sltu    x16, x10, x13           # elem.eは64ビットなので上位
32ビットもロードして加算する準備
```

```
        addi    x13, zero, 1
        addi    x14, zero, 0
...
        lw      x9, 4(x2)               # レジスタ復帰
        lw      x2, 8(x2)               # レジスタ復帰
        lw      x1, 12(x2)              # レジスタ復帰
        addi    x2, x2, 16
        ret
func_S256:                              # @func_S256
        addi    x2, x2, -32
        sw      x1, 28(x2)              # 4-byte Folded Spill
        sw      x2, 24(x2)              # 4-byte Folded Spill
        sw      x9, 20(x2)              # 4-byte Folded Spill
        addi    x9, x10, 0
        addi    x10, x2, 0
        ori     x11, x10, 4
        lw      x12, 4(x9)
        sw      x12, 0(x11)             #
func_S128の引数準備。引数をメモリにストアする。
        lw      x11, 12(x9)
        sw      x11, 12(x2)             #
func_S128の引数準備。引数をメモリにストアする。
        lw      x11, 8(x9)
        sw      x11, 8(x2)              #
func_S128の引数準備。引数をメモリにストアする。
        lw      x11, 0(x9)
        sw      x11, 0(x2)              #
func_S128の引数準備。引数をメモリにストアする。
        call    func_S128               # S128の合計値の計算
        addi    x12, x10, 0
        lw      x15, 28(x9)             # S256の残りのuint32_tデータをロード
        lw      x6, 24(x9)              # S256の残りのuint32_tデータをロード
        lw      x5, 20(x9)              # S256の残りのuint32_tデータをロード
        lw      x16, 16(x9)             # S256の残りのuint32_tデータをロード
        addi    x13, zero, 1
        addi    x14, zero, 0
        add     x17, x12, x16           # 加算を実行。RV32での
64ビット値の加算なので少し複雑
        add     x7, x17, x5
        add     x28, x7, x6
        add     x10, x28, x15
        sltu    x30, x10, x28
        addi    x29, x13, 0
```

`func_S256()` が `func_S128()` を呼び出すときはメモリに値をストアし、引数 `x10` にストアした値の先頭アドレスを格納して渡していることが分かります。　つまりRV32では、64ビットよりも大きな値の引数渡しにはメモリを介していることが分かります。 一方で `func_S128()` が `func_S64()` を呼び出すときは、引数の構造体は64ビットなのでレジスタ2つ（ `x10` と `x11` ）を使用して引数渡しをしていることが分かります。

### RV64における func_S128(), func_S256() の引数を受け取るアセンブリ命令

では続いてRV64における挙動を見てみます。 func_S32(), func_S128() における動作はRV32と同じなので省略して、func_S128() と func_S256() でのRV32との違いを見てみます。

- func_struct_simple.myriscvx64.static.medany.S

```
func_S128:                              # @func_S128
        # func_S128()の引数受け取りはx10、x11レジスタを使用して渡される。
        addi    x2, x2, -32
        sd      x1, 24(x2)              # 8-byte Folded Spill
        sd      x2, 16(x2)              # 8-byte Folded Spill
        sd      x9, 8(x2)               # 8-byte Folded Spill
        addi    x9, x11, 0
        # func_S64()の呼び出し方法はRV32と同じ。引数はx10を経由して渡される。
        call    func_S64
        add     x10, x10, x9
        ld      x9, 8(x2)               # 8-byte Folded Reload
        ld      x2, 16(x2)              # 8-byte Folded Reload
        ld      x1, 24(x2)              # 8-byte Folded Reload
        addi    x2, x2, 32
        ret

func_S256:                              # @func_S256
        # func_S256()の引数受け取りはメモリを介して行われる。
        addi    x2, x2, -32
        sd      x1, 24(x2)              # 8-byte Folded Spill
        sd      x2, 16(x2)              # 8-byte Folded Spill
        sd      x9, 8(x2)               # 8-byte Folded Spill
        addi    x9, x10, 0
        ld      x11, 8(x9)              # 引数の受け取り
        ld      x10, 0(x9)              # 引数の受け取り
        call    func_S128               # func_S128 への引数渡しはx10と
x11を経由して行われる。
        lwu     x11, 16(x9)             # 引数の受け取りと加算
        add     x10, x10, x11
        lwu     x11, 20(x9)             # 引数の受け取りと加算
        add     x10, x10, x11
        lwu     x11, 24(x9)             # 引数の受け取りと加算
        add     x10, x10, x11
        lwu     x11, 28(x9)             # 引数の受け取りと加算
        add     x10, x10, x11
        ld      x9, 8(x2)               # 8-byte Folded Reload
        ld      x2, 16(x2)              # 8-byte Folded Reload
        ld      x1, 24(x2)              # 8-byte Folded Reload
        addi    x2, x2, 32
        ret
```

違いは func_S128() の引数の渡し方のみです。128ビットサイズの構造体の引数は、レジスタを2本使って渡されます。 RV32ではメモリを経由して引数を渡していましたが、RV64ではレジスタを経由して渡して

いるところが異なります。

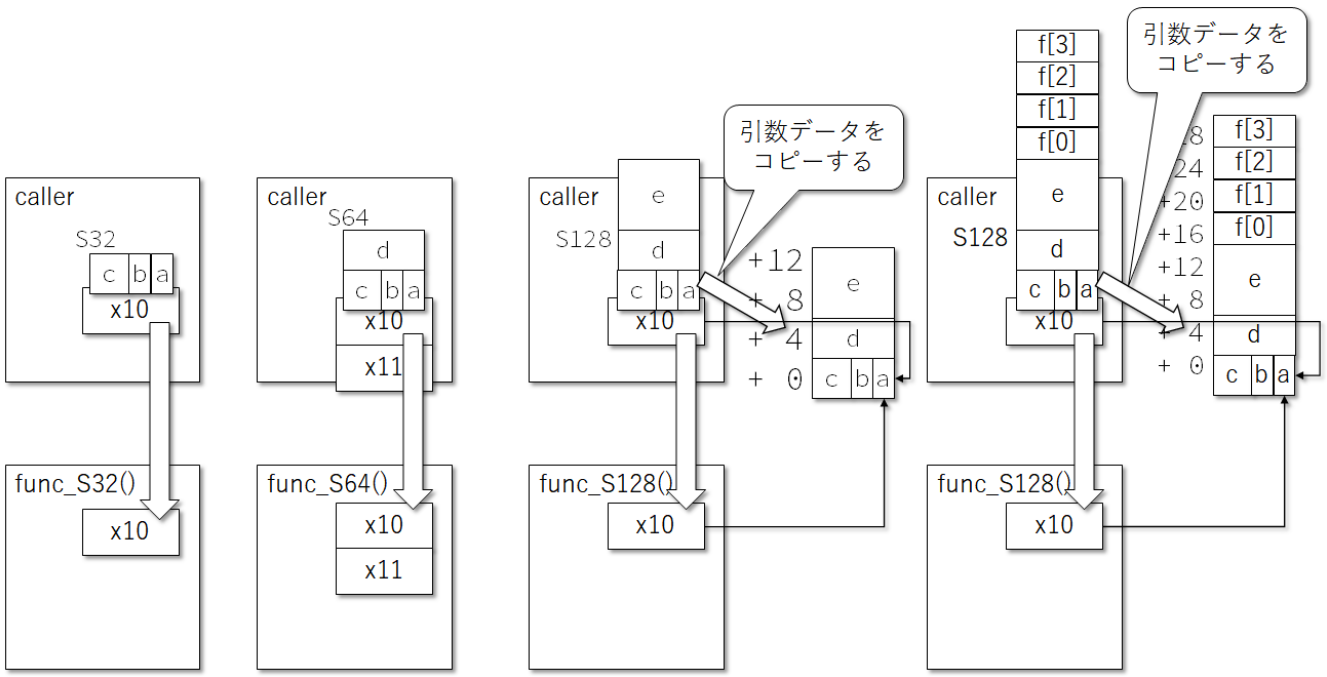



*Figure 7. fRV32*における関数の値渡し。*64*ビットまでは値をそのまま引数として渡す。それ以上のデータサイズを渡す時は、引数データへのポインタを渡す。値渡しをして関数を呼び出す側は、引数として渡したい値をメモリ中に確保し、その場所へのポインタを関数に引数として渡す。

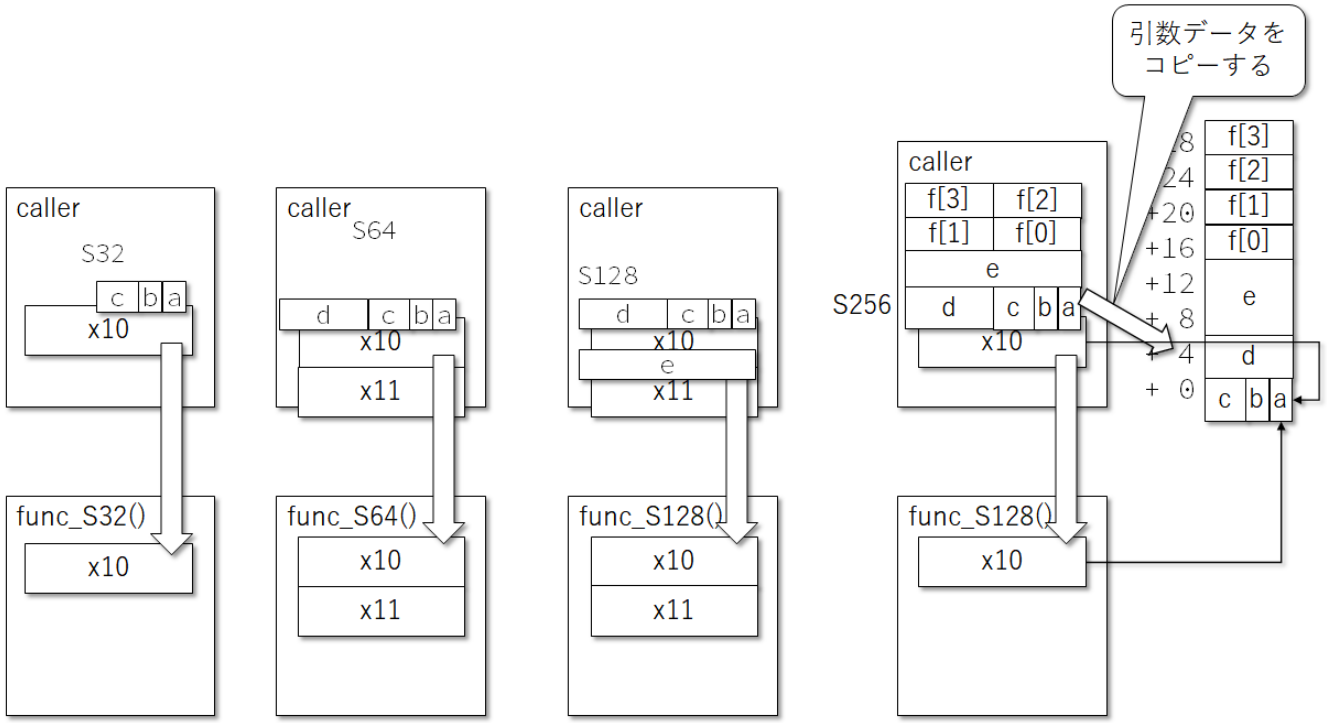



*Figure 8. RV64*における関数の値渡し。*128*ビットまでは値をそのまま引数として渡す。それ以上のデータサイズを渡す時は、引数データへのポインタを渡す。値渡しをして関数を呼び出す側は、引数として渡したい値をメモリ中に確保し、その場所へのポインタを関数に引数として渡す。

ではテストプログラムを作って動作を確認してみましょう。以下のようなコードを用意しました。

- program/appendix_1/func_struct_simple/test_func_struct_simple.c

```
#include <stdio.h>
#include "func_struct_simple.h"

extern int func_S32 (struct S32 elem);
extern int func_S64 (struct S64 elem);
extern int func_S128 (struct S128 elem);
extern int func_S256 (struct S256 elem);

int main()
{

  struct S256 s256;
  s256.s128.s64.s32.a = 100;
  s256.s128.s64.s32.b = 200;
  s256.s128.s64.s32.c = 300;
  s256.s128.s64.d     = 400;
  s256.s128.e         = 500;
  s256.f[0]           = 600;
  s256.f[1]           = 700;
  s256.f[2]           = 800;
  s256.f[3]           = 900;

  printf("total_S32  = %d\n", func_S32 (s256.s128.s64.s32));
  printf("total_S64  = %d\n", func_S64 (s256.s128.s64));
  printf("total_S128 = %d\n", func_S128(s256.s128));
  printf("total_S256 = %d\n", func_S256(s256));

  return 0;
}
```

まずはRV32からです。以下のようにしてバイナリを作成します。

```
$ riscv32-unknown-elf-gcc -march=rv32g -DPREALLOCATE=1 -static -mcmodel=medany
-std=gnu99 \
    -O2 -ffast-math -lm -lgcc test_func_struct_simple.c \
    func_struct_simple.myriscvx32.static.medany.S \
    -o test_func_struct_simple_rv32
```

このプログラムをシミュレータで実行します。以下のようにしてシミュレーションします。

```
$ spike --isa=rv32imac pk test_func_struct_simple_rv32
```

```
bbl loader
total_S32  = 600
total_S64  = 1000
total_S128 = 1500
total_S256 = 4500
```

想定通りの結果が得られました。構造体の値渡しは上手く行っているようです。

次にRV64でテストを行ってみましょう。以下のようにしてバイナリを作成します。

```
$ riscv64-unknown-elf-gcc -march=rv64g -DPREALLOCATE=1 -static -mcmodel=medany
-std=gnu99 -O2 \
    -ffast-math -fno-common -fno-builtin-printf -nostdlib -nostartfiles -lm -lgcc \
    test_func_struct_simple.c func_struct_simple.myriscvx64.static.medany.S \
    crt.S syscalls.c -o test_func_struct_simple_rv64 -T link.ld
```

```
total_S32  = 600
total_S64  = 1000
total_S128 = 1500
total_S256 = 4500
```

こちらも想定通りの結果です。構造体の値渡しの命令がうまく生成できていることが確認できました。

# [LowLayer] 可変長引数のサポート

## [LowLayer] 可変長引数はどのようにして実現されるのか

可変長引数というのは、関数の引数の数を固定せずに、任意の数の引数を渡すことができる仕組みのことを言います。 関数を定義するとき、通常はその関数の引数は数が決まっており、各引数の型なども決まっています。 しかし可変長引数では、引数の数を固定せずにいくらでも引数を渡すことができるようになります。 引数の数が固定されないことで、関数をより柔軟に活用することができます。

C言語の可変長引数では、たとえば以下のような記述が可能となります。 C言語の文法的な詳細についてはここでは説明しませんが、`sum_i()` の引数の一つに `...` という構文が挿入されています。 ここは引数が可変長であることを意味し、先頭の `amount` が可変長引数で渡される実際の引数の数を指定し、そのあとに可変長の引数が渡されます。

- `program/appendix_1/vararg.c`

```
#include <stdarg.h>

int sum_i(int amount, ...)
{
  int i = 0;
  int val = 0;
  int sum = 0;

  va_list vl;
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
    val = va_arg(vl, int);
    sum += val;
  }
  va_end(vl);

  return sum;
}


int test_vararg()
{
  int a = sum_i(10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
  return a;
}
```

可変長引数を渡す側は、最初の引数として以降に渡す引数の数を指定し、そこから所望の数だけ引数を並べます。

```
  int a = sum_i(10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
```

可変長引数を受け取る側は、`va_list`, `va_start`, `va_end` といった特殊な型を使用して処理します。 上記のサンプルプログラムでは、`amount` の数だけループを回して `va_arg()` 関数を使用して引数を取得して、それぞれの引数に対して処理を行います。 可変長引数の処理が完了すると `va_end` を呼び出して処理を完了します。

この可変長引数は特殊な文法のように見えて、普段C言語を使うときには非常にお世話になっています。 たとえば `printf()` などの文字を出力する場合には知らず知らずのうちに可変長引数を使用しています。 表示するデータの数において引数の数が変わるので、可変長引数となっているわけです。

```
// printfは可変長引数を受け取ることができる
printf ("Hello %s", llvm_string);                    // 2つの引数を渡している。
printf ("Hello %s %d", llvm_string, llvm_version);  // 3つの引数を渡している。
```

この可変長引数はC言語の文法としては少し特殊な形をしているかもしれませんが、フロントエンドによって一度IRの形式になってしまえば、あとはレジスタを割り付けて命令を生成するだけなのでそこまで難

しくはなさそうです。 DAGによってさらに別の種類のノードをサポートする必要があります。

しかし、可変長引数でも基本的な考え方は変わりません。関数を呼び出す側は、可能な数だけレジスタに引数を渡し、レジスタに入りきらなければスタックに積み上げていきます。 呼び出された関数の処理が、これまでと異なります。 つまり、効率的に可変長引数を扱うために、レジスタ経由で受け付けた引数をいったんスタックにすべて退避していく処理が追加されます。 これにより、実際の引数を処理するプログラムを効率的に扱うことができるようになります。

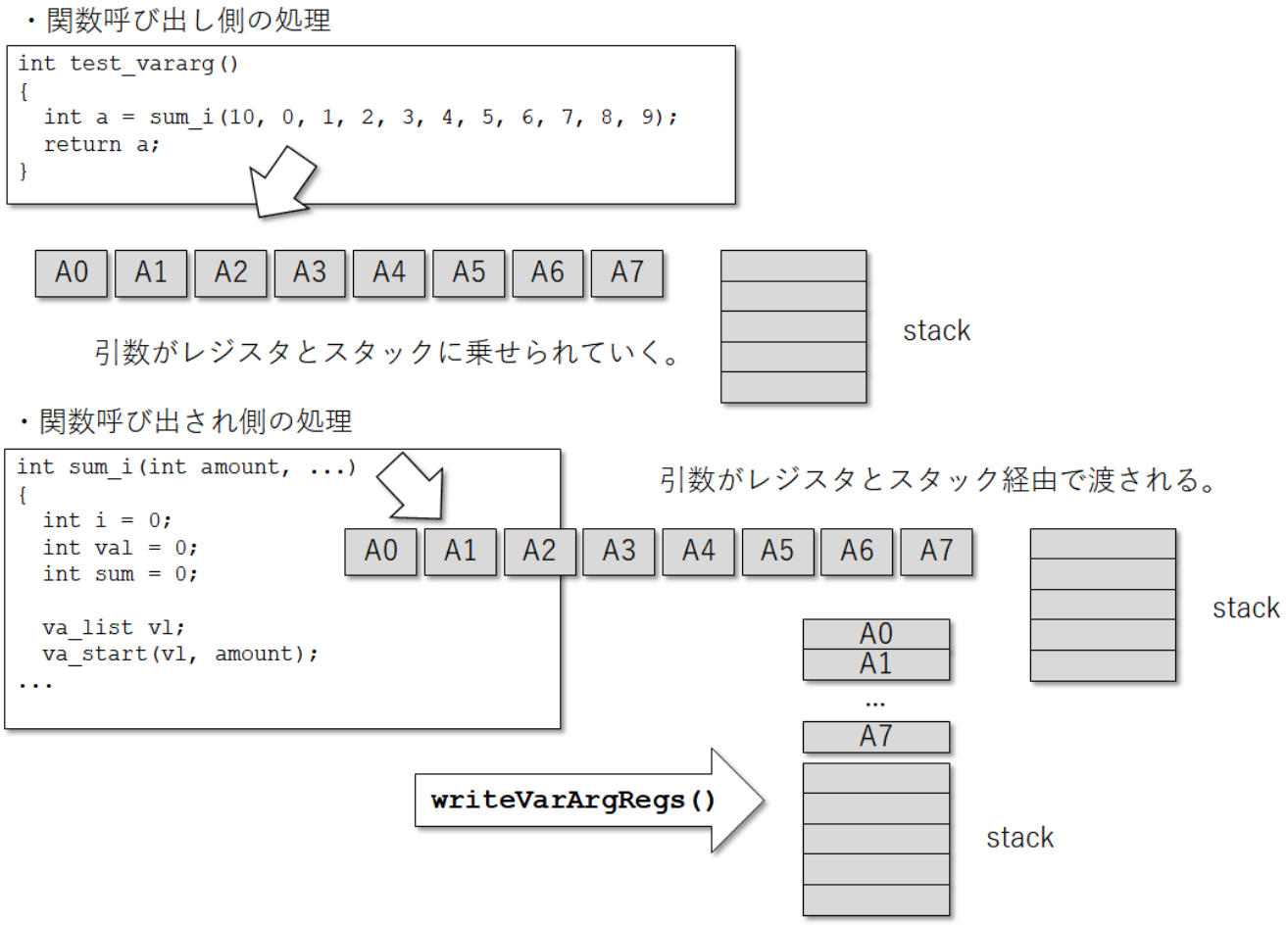



*Figure 9.* 可変長引数での引数の受け渡し方。関数呼び出し側は変更ないが、呼び出され側はスタックへの退避処理が追加される。

可変長引数にMYRISCVXでサポートするために必要な工程は主に以下の通りです。

1. 可変長引数に関係するノードの生成条件を変更する。`VASTART` のみを取り扱うように変更する。
2. `VASTART` に基づいてDAGを生成するための `MYRISCVXTarget::selectVASTART()` を実装する。これにより可変長引数の情報がメモリに渡されるDAGが生成される。
3. `MYRISCVXTargetLowering::LowerFormalArguments()` 内に `writeVarArgRegs()` の呼び出しを追加し、可変長引数を受け取るとその値をスタックに積み上げる操作を追加する。

まずは、可変長引数のC言語プログラムから生成されたLLVM IRを見てみましょう。

```
$ ${BUILD}/bin/clang  -O1 --target=riscv32-unknown-elf vararg.c -c \
    -emit-llvm -o vararg.riscv32.static.bc
$ ${BUILD}/bin/llvm-dis vararg.riscv32.static.bc -o vararg.riscv32.static.bc.ll
```

まずは呼び出し側、 `test_vararg()` のIRです。可変長引数であることを示す `...` が付加されていますが、関数が呼び出される際に付加される引数が列挙されています。

```
define dso_local i32 @test_vararg() local_unnamed_addr #0 {
entry:
  %call = call i32 (i32, ...) @sum_i(i32 10, i32 0, i32 1, i32 2, i32 3, i32 4, i32 5,
i32 6, i32 7, i32 8, i32 9)
  ret i32 %call
}
```

次に呼び出され側、 `vaarg()` のIRです。可変長引数であることを示す `...` が付加されています。

```
define dso_local i32 @sum_i(i32 %amount, ...) local_unnamed_addr #0 {
entry:
  // 変数vlを用意する。
  %vl = alloca i8*, align 4
  %0 = bitcast i8** %vl to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* nonnull %0) #2
  // vastartを呼び出す。
  call void @llvm.va_start(i8* %0)
  %cmp8 = icmp sgt i32 %amount, 0
  br i1 %cmp8, label %for.body, label %for.end

// 可変長引数を呼び出すループ本体
for.body:                                         ; preds = %entry, %for.body
  %sum.010 = phi i32 [ %add, %for.body ], [ 0, %entry ]
  %i.09 = phi i32 [ %inc, %for.body ], [ 0, %entry ]
  // 可変長引数の値を取り出すためのポインタを生成する。
  %argp.cur = load i8*, i8** %vl, align 4
  %argp.next = getelementptr inbounds i8, i8* %argp.cur, i32 4
  store i8* %argp.next, i8** %vl, align 4
  %1 = bitcast i8* %argp.cur to i32*
  // 引数のロード
  %2 = load i32, i32* %1, align 4
  // 加算の実行
  %add = add nsw i32 %2, %sum.010
  %inc = add nuw nsw i32 %i.09, 1
  %exitcond = icmp eq i32 %inc, %amount
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body, %entry
  %sum.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ]
  call void @llvm.va_end(i8* nonnull %0)
  call void @llvm.lifetime.end.p0i8(i64 4, i8* nonnull %0) #2
  ret i32 %sum.0.lcssa
}
```

可変長引数と言えども、C言語の挙動とほとんど変わらないLLVM IRが生成されていることが分かりま

す。 ループの内部でポインタを生成し引数の格納場所を生成し、引数をロードしては加算しています。

# [LLVM] 可変長引数をサポートするためのLLVM実装

## SelectionDAGの生成

まず、VAARG, VACOPY, VAEND の3つは最適化の最中に生成されないように抑制します。つまり、MYRISCVXTargetLowering にて、 setOperationInAction() で対象となるノードの生成を抑制します。

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

```
MYRISCVXTargetLowering::MYRISCVXTargetLowering(const MYRISCVXTargetMachine &TM,
                                               const MYRISCVXSubtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
...
  // 可変長引数のためのノード設定
  setOperationAction(ISD::VASTART,   MVT::Other, Custom);

  setOperationAction(ISD::VAARG,     MVT::Other, Expand);
  setOperationAction(ISD::VACOPY,    MVT::Other, Expand);
  setOperationAction(ISD::VAEND,     MVT::Other, Expand);
```

- VASTART : 可変長引数の処理を開始します。
- VAARG : 可変長引数から実際の値を取り出します。
- VACOPY : 可変長引数の値をコピーします。
- VAEND : 可変長引数の処理を終了します。

VASTART だけはカスタム実装に設定したので、 lowerVASTART() のみ実装します。

lowerVASTART() では、可変長引数のアドレスをメモリにストアします。 LowerFormalArguments() つまり可変長引数を持つ関数が呼び出されたときの引数の処理について、可変長引数であれば専用のルーチン writeVarArgRegs() を呼び出します。 この writeVarArgRegs() は、レジスタ渡しをしたデータをスタックに戻します。 せっかくレジスタ渡しをしたのにどうしてスタックに戻すんだ？と思われるかもしれませんが、それ以降の処理は一律してfor文の処理で統一するため、レジスタ渡し、 メモリロードの2種類でアセンブリを生成したくない、という意図があると思われます。

```
  /* for文の前に、レジスタ渡しした可変長引数の要素をスタックに置いておく。 */
...
  for (i = 0; i < amount; i++)
  {
    /* valの取得はスタックから一律に受け取る */
    val = va_arg(vl, int);
    sum += val;
  }
```

## 可変長引数を受け取る側の処理

まず、 VASTART 命令の処理から入りましょう。 LowerOperation() のswitch文に1つ条件を追加して、VASTART ノードにぶつかった場合は lowerVASTART() にジャンプします。

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

```
SDValue MYRISCVXTargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  // SelectionDAGのノード種類をチェック
  switch (Op.getOpcode())
  {
    // SELECTノードはカスタム関数で処理する
    case ISD::SELECT       : return lowerSELECT(Op, DAG);
    // GlobalAddressノードであれば，lowerGlobalAddress()を呼び出す
    // GlobalAddressノードはあらかじめ
setOperationAction()でカスタム処理を呼び出すように設定してある
    case ISD::GlobalAddress: return lowerGlobalAddress(Op, DAG);
    // VASTARTノードはカスタム関数で処理する
    case ISD::VASTART      : return lowerVASTART       (Op, DAG);
```

VASTART では、引数の先頭を指すポインタを用意し、これを関数フレームの先頭に配置（メモリストア）します。 このポインタをインデックスにして、各引数に対するアクセスが行われます。

さらに追加する処理として、 LowerFormalArguments() の中で、引数に IsVarArg フラグが立っていた場合の処理を追加します。 この場合には writeVarArgRegs() を呼び出して可変長引数をすべてスタックに格納していきます。

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

```
/// LowerFormalArguments()
// 引数渡しにおいて、引数を渡す方法を実装する
SDValue
MYRISCVXTargetLowering::LowerFormalArguments(SDValue Chain,
                                             CallingConv::ID CallConv,
                                             bool IsVarArg,
                                             const SmallVectorImpl<ISD::InputArg>
&Ins,
                                             const SDLoc &DL, SelectionDAG &DAG,
                                             SmallVectorImpl<SDValue> &InVals)
const {
...
  // 可変長引数ならば，writeVarArgRegsに飛んで引数をスタックに格納していく
  if (IsVarArg)
    writeVarArgRegs(OutChains, Chain, DL, DAG, CCInfo);
```

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

```
void MYRISCVXTargetLowering::writeVarArgRegs(std::vector<SDValue> &OutChains,
                                             SDValue Chain, const SDLoc &DL,
                                             SelectionDAG &DAG,
                                             CCState &State) const {
  ArrayRef<MCPhysReg> ArgRegs = ABI.GetVarArgRegs();
  unsigned Idx = State.getFirstUnallocated(ArgRegs);
  unsigned RegSizeInBytes = Subtarget.getGPRSizeInBytes();
  MVT RegTy = MVT::getIntegerVT(RegSizeInBytes * 8);
  const TargetRegisterClass *RC = getRegClassFor(RegTy);
  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  MYRISCVXFunctionInfo *MYRISCVXFI = MF.getInfo<MYRISCVXFunctionInfo>();

...
  for (unsigned I = Idx; I < ArgRegs.size();
       ++I, VaArgOffset += RegSizeInBytes) {

    // レジスタを経由して渡された引数をすべてスタックに積み上げていく
    LLVM_DEBUG(dbgs() << "writeVarArgRegs I = " << I << '\n');

    unsigned Reg = addLiveIn(MF, ArgRegs[I], RC);
    SDValue ArgValue = DAG.getCopyFromReg(Chain, DL, Reg, RegTy);
    FI = MFI.CreateFixedObject(RegSizeInBytes, VaArgOffset, true);
    SDValue PtrOff = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
    SDValue Store =
        DAG.getStore(Chain, DL, ArgValue, PtrOff, MachinePointerInfo());
    cast<StoreSDNode>(Store.getNode())->getMemOperand()->setValue(
        (Value *)nullptr);
    OutChains.push_back(Store);
  }
}
```

`writeVarArgRegs()` の実装を見てみると、レジスを経由して渡された変数がすべてメモリに格納されていることが分かります。 実際に `va_arg` による引数処理が入る前に、すべての引数をメモリ中で順番に配置しておくことにより、可変長引数の処理において「ここまではレジスタから取ってくる」「ここから先はメモリから取ってくる」、 という条件分岐を持たなくても良くなります。

### 可変長引数の関数を呼び出す側の処理

次に、可変長引数の関数を呼び出す側の処理ですが、これは追加の実装は必要ありません。 これまで通り引数をレジスタに渡し、レジスタに収まりきらなければスタックに積んでいけば良いのです。

## [LLVM] 可変長引数の実コードで確認

上記のコードをコンパイルして、どのようなコードが生成されているのか見てみます。MYRISCVX64で試行します。

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

### 可変長引数を渡す側（test_vararg()）

- vararg.myriscvx64.static.S

```
# %bb.0:                                # %entry
        addi    x2, x2, -48
        sd      x1, 40(x2)              # 8-byte Folded Spill
        sd      x2, 32(x2)              # 8-byte Folded Spill
        addi    x10, x0, 9          # 第10引数 = 9の格納
        sd      x10, 16(x2)
        addi    x10, x0, 8          # 第9引数 = 8の格納
        sd      x10, 8(x2)
        addi    x10, x0, 7          # 第8引数 = 7の格納
        sd      x10, 0(x2)
        addi    x10, x0, 10         # 引数の数: amountの格納
        mv      x11, x0             # 第1引数 = 0の格納
        addi    x12, x0, 1          # 第2引数 = 1の格納
        addi    x13, x0, 2          # 第3引数 = 2の格納
        addi    x14, x0, 3          # 第4引数 = 3の格納
        addi    x15, x0, 4          # 第5引数 = 4の格納
        addi    x16, x0, 5          # 第6引数 = 5の格納
        addi    x17, x0, 6          # 第7引数 = 6の格納
        call    sum_i
```

レジスタ渡しできる限りはレジスタ経由で渡します。a0-a7までのレジスタを使って引数を渡し、それでも足りないので残りはスタック経由で渡しています。

### 可変長引数を受け取る側（sum_i()）

```
sum_i:                                  # @sum_i

# %bb.0:                                # %entry
        addi    x2, x2, -64
        sd      x17, 56(x2)             # 第7引数をスタックに格納
        sd      x16, 48(x2)             # 第6引数をスタックに格納
        sd      x15, 40(x2)             # 第5引数をスタックに格納
        sd      x14, 32(x2)             # 第4引数をスタックに格納
        sd      x13, 24(x2)             # 第3引数をスタックに格納
        sd      x12, 16(x2)             # 第2引数をスタックに格納
        sd      x11, 8(x2)              # 第1引数をスタックに格納
        addi    x11, x2, 8
        sd      x11, 0(x2)
        mv      x11, x0
        addi    x12, x0, 1
        slt     x12, x10, x12
        bne     x12, x0, $BB0_3
        j       $BB0_1
$BB0_1:                                 # %for.body.preheader
        addi    x11, x0, 0
        addi    x12, x11, 0
$BB0_2:                                 # %for.body
                                        # =>This Inner Loop Header: Depth=1
        ld      x13, 0(x2)
        addi    x14, x13, 8
        sd      x14, 0(x2)
        lw      x13, 0(x13)             # スタックから引数を取り出す
        add     x11, x13, x11           # 加算処理
        addiw   x12, x12, 1
        slt     x13, x12, x10
        bne     x13, x0, $BB0_2
        j       $BB0_3
$BB0_3:                                 # %for.end
        addiw   x10, x11, 0
        addi    x2, x2, 64
        ret
```

まず、レジスタ経由で渡した引数をすべてスタックに格納します。もったいないですが、こうすることで以降の処理コードをより簡潔にします。続いて、実際の引数の処理に入ります。スタックから引数をロードして、加算が実行されます。

# 末尾再帰関数呼び出しの実装

## [LowLayer] 末尾再帰とは

関数呼び出しにはさまざまな最適化の形がありますが、その中の一つである末尾関数呼び出しでの最適化を実行してみましょう。末尾関数呼び出しとは、ある関数 `f1()` が関数 `f2()` の最後の処理として呼び出されるケースを言います。

```
int f1(a, b) { a + b; }
int f2(c, d, e, f) {
    /* さまざまな処理*/
    return f1(10, 20);
}
```

この場合、関数 `f1()` を呼び出すためには引数渡しの処理、スタックの処理、そして呼び出しから戻ってきた場合には戻り値の処理やスタックの後片付けを行う必要があります。

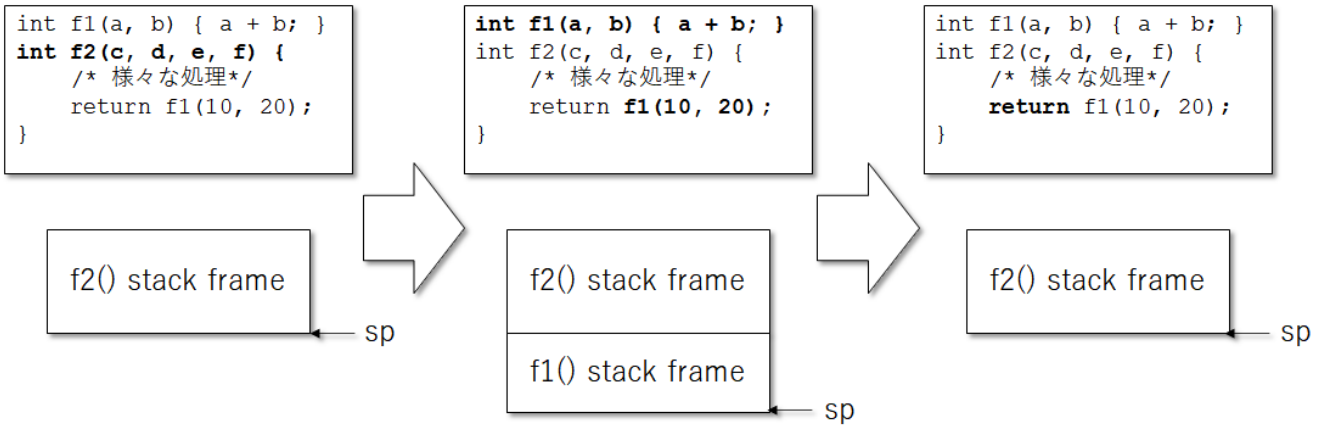



*Figure 10.* 通常のスタックフレームの作成手順。関数が呼び出されると新たなスタックフレームが作成される。

しかし、場合によってはこれは不要なことがあります。 つまり、どっちにしろ `f1` から戻った後は `f2` からも戻ってしまうのだから、せっかくなので `f2` のスタック領域なども `f1` で使わせてもらって、 関数呼び出し時のスタックの掘り下げを省略してしまいましょう。 末尾呼び出しではスタックフレームを追加することなく `f1` を呼び出すことができます。

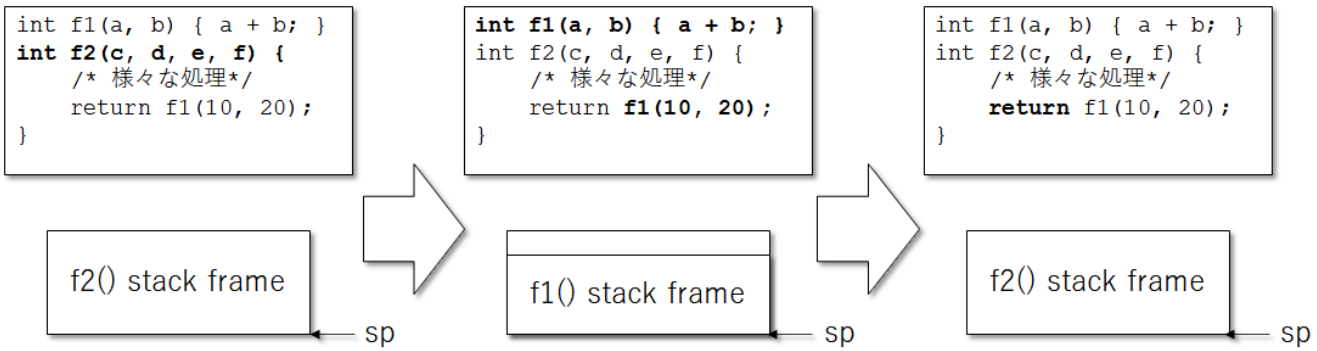
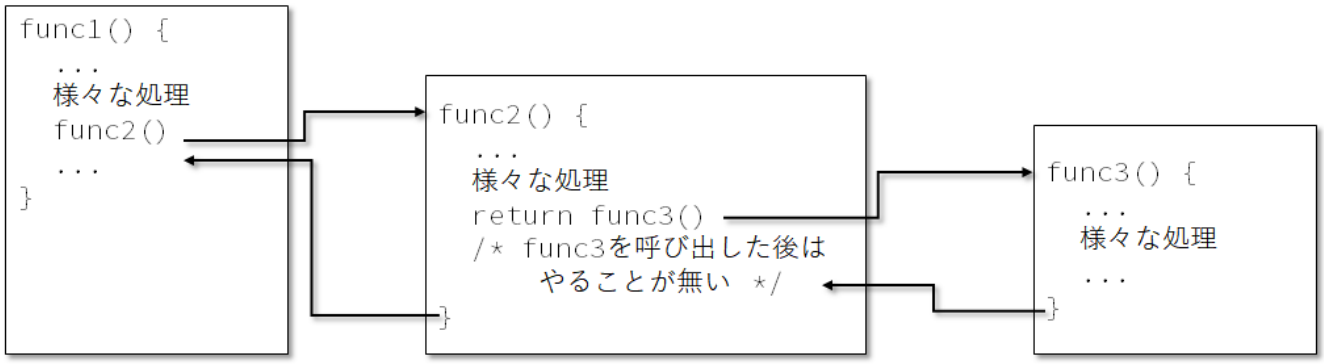


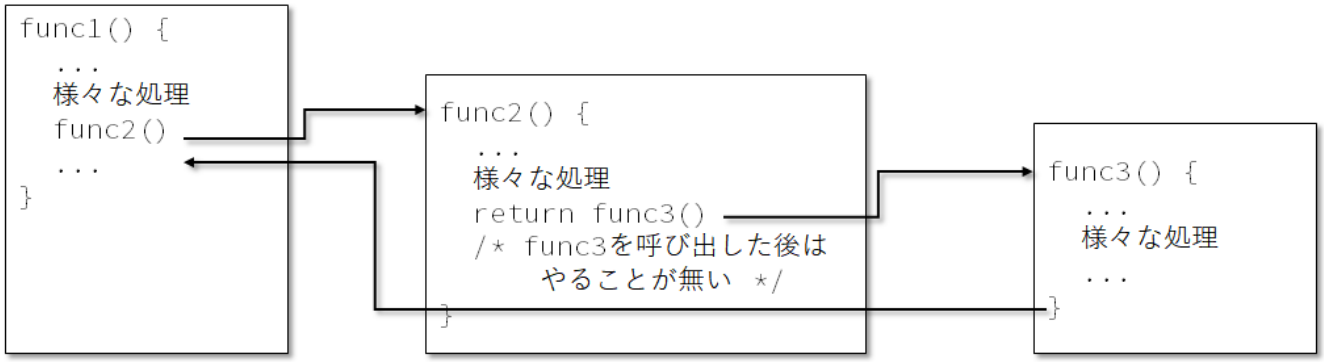

*Figure 11. Call*のスタックフレームの作成手順。スタックフレームは新たに作成されず、引数の調整だけを行って関数に飛ぶ。

この末尾再帰には以下のメリットがあります。

- 関数フレームを節約することでメモリの使用量を節約できます。
- 末尾再帰を呼び出したあとにその関数ではもう実行することが無いため、直接おおもとの関数に戻ることができます（Figure 12）。




(a)末尾再帰を使用しない場合の関数呼び出し。




(b)末尾再帰を使用する場合の関数呼び出し。

*Figure 12.* 末尾再帰による関数呼び出しから戻るときの処理を省略するケース。末尾再帰の場合は*func3()*の実行が終わったあとに*func2()*の内部ではもうやることが無いので、*func2()*に戻ることなく*func3()*から*func1()*に直接戻った方が効率が良い。

しかし、この末尾呼び出しはすべての条件下で実現可能なわけではありません。　上記で説明した通り、`f2` のスタックフレームを使用して `f1` が実行されます。 つまり、 `f1` が使う予定であるスタックフレームのサイズが `f2` のスタックフレームのサイズを超えていてはいけません。 また、 `f1` のスタックフレームのサイズが最初から算出できない場合にも末尾関数呼び出し最適化は適用できません。 `f1` のスタックフレーム以外の場所まで破壊してしまう可能性があるからです。

では、実際にLLVMで末尾関数呼び出しの最適化を実行してみましょう。以下のような関数を考えます。これを末尾再帰で表現するとどのようになるかを見てみましょう。

- `program/appendix_1/tailcall.c`

```
int tail_call_func(int a, int b) {
  int total = external_func(a);
  for (int i = a; i < b; i++) {
    total += i;
  }
  return total;
}

int inc(int a)
{
  return a+1;
}

int tail_call_main (int a, int b, int c, int d) {
  int e = inc(a) + inc(b);
  int f = inc(c) + inc(d);

  return tail_call_func(e, f);
}
```

（途中で `inc()` を読んでいるのは、関数の途中で別の関係ない関数を呼び出すことによりレジスタ退避を発生させ、スタックフレームを必ず作り出すようにしたものです。）

# [LLVM] 末尾再帰のLLVMへの実装

まずは上記のサンプルプログラムをclangでコンパイルし、LLVM IRを生成してみます。

```
$ ${BUILD}/bin/clang -O1 --target=riscv64-unknown-elf tailcall.c -c -emit-llvm \
    -o tailcall.riscv64.static.bc
$ ${BUILD}/bin/llvm-dis tailcall.riscv64.static.bc -o tailcall.riscv64.static.bc.ll
```

生成されたLLVM IR `tailcall.riscv64.static.bc.ll` を見てみると以下のようになっています。

- `tailcall.riscv64.static.bc.ll`

```
define dso_local i32 @tail_call_func(i32 %a, i32 %b) local_unnamed_addr #0 {
entry:
  ; ここでは直接関係ないので省略
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @inc(i32 %a) local_unnamed_addr #0 {
entry:
  %add = add nsw i32 %a, 1
  ret i32 %add
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @tail_call_main(i32 %a, i32 %b, i32 %c, i32 %d)
local_unnamed_addr #0 {
entry:
  %call = call i32 @inc(i32 %a)
  %call1 = call i32 @inc(i32 %b)
  %add = add nsw i32 %call1, %call
  %call2 = call i32 @inc(i32 %c)
  %call3 = call i32 @inc(i32 %d)
  %add4 = add nsw i32 %call3, %call2
  %call5 = call i32 @tail_call_func(i32 %add, i32 %add4)
  ret i32 %call5
}
```

この中で、最後に関数呼び出ししているのは `tail_call_main` 内の `tail_call_func()` です （実際には `ret` 文がありますが、関数の戻り値制御以外で実質的な最期の命令は `tail_call_func` の呼び出しです）。

この時、LLVM IR的には `call [戻り値] @関数名(引数)` となっているのを `tail call [戻り値] @関数名(引数)` に変えます。 実はこれで `tail_call_func()` の呼び出しが末尾再帰呼び出しに置き換わります。 末尾再帰の効果を見るために、callを書き換えたLLVM IRのテキストファイルを `llc` に渡して効果を確認していきましょう。

- `tailcall.riscv64.static.bc.tail.ll`

```
define dso_local i32 @tail_call_func(i32 %a, i32 %b) local_unnamed_addr #0 {
entry:
  ; ここでは直接関係ないので省略
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @inc(i32 %a) local_unnamed_addr #0 {
entry:
  %add = add nsw i32 %a, 1
  ret i32 %add
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @tail_call_main(i32 %a, i32 %b, i32 %c, i32 %d)
local_unnamed_addr #0 {
entry:
  %call = call i32 @inc(i32 %a)
  %call1 = call i32 @inc(i32 %b)
  %add = add nsw i32 %call1, %call
  %call2 = call i32 @inc(i32 %c)
  %call3 = call i32 @inc(i32 %d)
  %add4 = add nsw i32 %call3, %call2
  ;; callからtail callに変更
  %call5 = tail call i32 @tail_call_func(i32 %add, i32 %add4)
  ret i32 %call5
}
```

**NOTE**

コラム：末尾再帰の呼び出しを*clang*から直接生成できないのか？

今回は末尾再帰呼び出しのLLVM　IRを生成するためにわざわざ生成したIRファイルを手で書き換えました。 実は `clang` オプションを用いて末尾再帰を自動的に生成する方法は無いかといろいろ調査したのですが、見つけることができませんでした。

実はclangでは簡単な末尾再帰なら最適化で単純なループに変換してしまいます。　たとえば、末尾再帰と言えば有名な以下のようなプログラムを考えます（ある値`n`に対する`n!`を計算する関数です）。

```
int factorial_tail_impl(int n, int accum)
{
  if (n == 1) {
    return accum;
  }
  return factorial_tail_impl(n-1, accum * n);
}

int facotorial_tail(int n)
{
  return factorial_tail_impl(n, 1);
}
```

factorial_tail_imp() は末尾再帰の形式をしており、筆者がclang-8で実験していた時は問題なくtail callが生成できていました。

```
$ ${BUILD}/bin/clang-8 -Oz factorial.c -c -emit-llvm -o factorial.riscv64.static.bc
$ ${BUILD}/bin/llvm-dis factorial.riscv64.static.bc -o factorial.riscv64.static.bc.ll
```

- factorial.riscv64.static.bc.ll

```
define dso_local i32 @factorial_tail_impl(i32 %n, i32 %accum) local_unnamed_addr #0 {
entry:
  br label %tailrecurse

tailrecurse:                                      ; preds = %if.end, %entry
  %n.tr = phi i32 [ %n, %entry ], [ %sub, %if.end ]
  %accum.tr = phi i32 [ %accum, %entry ], [ %mul, %if.end ]
  %cmp = icmp eq i32 %n.tr, 1
  br i1 %cmp, label %return, label %if.end

if.end:                                           ; preds = %tailrecurse
  %sub = add nsw i32 %n.tr, -1
  %mul = mul nsw i32 %accum.tr, %n.tr
  br label %tailrecurse

return:                                           ; preds = %tailrecurse
  ret i32 %accum.tr
}

; Function Attrs: minsize nounwind optsize readnone uwtable
define dso_local i32 @facotorial_tail(i32 %n) local_unnamed_addr #0 {
entry:
  %call = tail call i32 @factorial_tail_impl(i32 %n, i32 1) #1
  ret i32 %call
}
```

factorial_tail_imp が何やら思ったのと違う形で変換されてしまいました。 clangのオプションで使った -Oz はIRのサイズを減少させるためのオプションです。 factorial_tail_impl() の呼び出しの部分で慰み程度に tail call が使われていますが、肝心の factorial_tail_imp には末尾再帰が行われていません。その代わりに、

```
int fatorial_tail_imp(int n, int accum)
{
    if (n == 1) {
        return accum;
    } else {
        for (; n >= 1; n--) {
            accum = accum * n;
        }
    }
    return accum;
}
```

というさらに単純なコードに変換してしまいます。これだと末尾再帰サポートのための実装にならないので無理やりIRを改造したというわけです。

===

では、このIRを処理して末尾関数呼び出し最適化を適用してみましょう。末尾再帰を挿入するためには、末尾再帰のカスタムノードを挿入する必要があります(Figure 13)

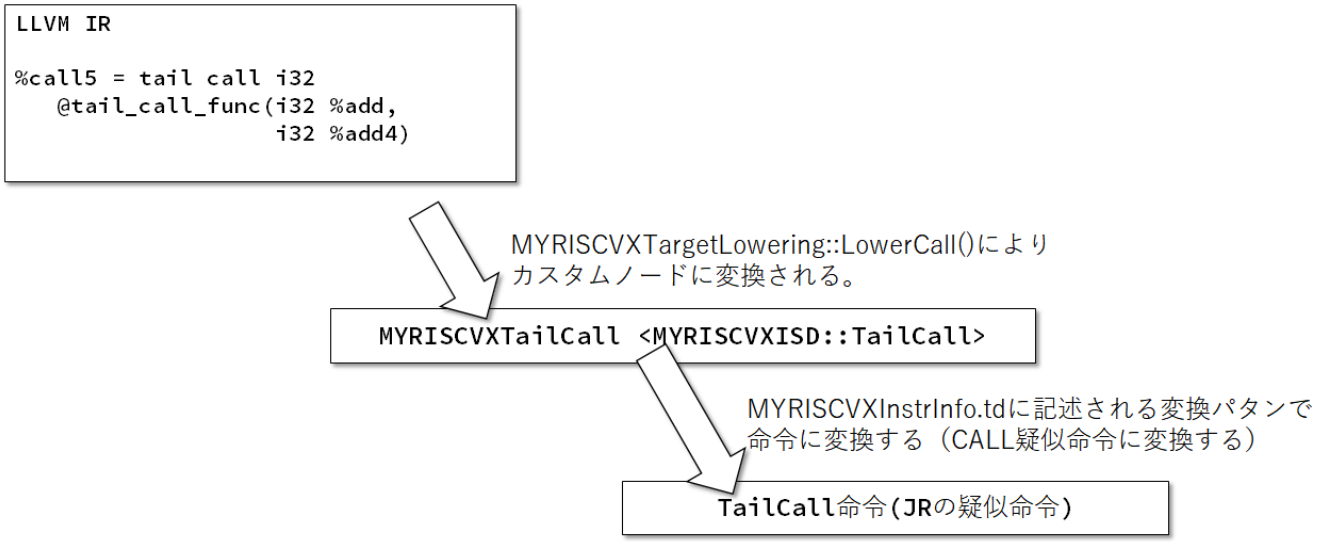



*Figure 13. LLVM IRからカスタムノードを経由し関数呼び出しに変換されるフロー。LLVM IRのtail call命令はカスタムノードMYRISCVXTailCallに一度変換され、さらに変換パタンによりMYRISCVXの命令に変換される。*

実装しなければならないのは、

- 末尾関数呼び出し最適化が適用可能かチェックする
- スタックフレームを生成しないように制御する
- 引数渡しの処理を行う
- 関数ジャンプ
- 戻り値処理

と、とりあえずは関数呼び出しのシーケンスの部分に手を加えれば良さそうです。 これらはすべてIRからDAGへの変換時に適用できるので、 `MYRISCVXISelLowering.cpp` に記述されている `MYRISCVXTargetlowering` クラスに手を加えましょう。 関数呼び出しの部分なので、

MYRISCVXTargetLowering:::LowerCall() を改造すれば良さそうです。

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

```
SDValue
MYRISCVXTargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                                  SmallVectorImpl<SDValue> &InVals) const {

  SelectionDAG &DAG                     = CLI.DAG;
  SDLoc DL                              = CLI.DL;
  SmallVectorImpl<ISD::OutputArg> &Outs = CLI.Outs;
  SmallVectorImpl<SDValue> &OutVals     = CLI.OutVals;
  SmallVectorImpl<ISD::InputArg> &Ins   = CLI.Ins;
  SDValue Chain                         = CLI.Chain;
  SDValue Callee                        = CLI.Callee;
...
  CallingConv::ID CallConv              = CLI.CallConv;
  bool IsVarArg                         = CLI.IsVarArg;


  EVT PtrVT = getPointerTy(DAG.getDataLayout());

  MachineFunction &MF = DAG.getMachineFunction();
  const TargetFrameLowering *TFL = MF.getSubtarget().getFrameLowering();
  bool IsPIC = isPositionIndependent();

  // 関数呼び出しの引数解析を行い，各オペランドの配置方法を決める
  SmallVector<CCValAssign, 16> ArgLocs;
  CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                 ArgLocs, *DAG.getContext());
  CCInfo.AnalyzeCallOperands (Outs, CC_MYRISCVX);

  // Get a count of how many bytes are to be pushed on the stack.
  unsigned NextStackOffset = CCInfo.getNextStackOffset();

  // 末尾最適化が本当に適用可能かチェックする
  if (IsTailCall)
    IsTailCall = isEligibleForTailCallOptimization(CCInfo, NextStackOffset,

*MF.getInfo<MYRISCVXFunctionInfo>());

  if (IsTailCall) {
    ++NumTailCalls;
  }
```

IsTailCall 変数は関数呼び出しが末尾関数呼び出しになっている場合にまずはTrueとなります。 次に、本当に末尾関数呼び出し最適化が適用可能かチェックします。これを行っているのが isEligibleForTailCallOptimization です。

末尾関数呼び出し最適化が適用できる条件としては、

- 呼び出し先もしくは呼び出し元が byval の引数を持っている場合はダメです。
- 呼び出し先が使用する予定のスタックサイズ(NextStackOffset)よりも呼び出し元が持っているスタックの サイズ FI.getIncomingArgSize() が大きい場合はダメです。

上記をすべてクリアできれば末尾関数呼び出し最適化が適用できます。

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

```
bool MYRISCVXTargetLowering::
isEligibleForTailCallOptimization(
    CCState &CCInfo,
    unsigned NextStackOffset, const MYRISCVXFunctionInfo& FI) const {

  // ByVal引数（構造体など）を持っている場合は適用不可能
  if (CCInfo.getInRegsParamsCount() > 0 || FI.hasByvalArg())
    return false;

  // 呼び出し先の関数のスタックサイズが呼び出し元のスタックサイズよりも大きい場合は
  // 適用不可能
  return NextStackOffset <= FI.getIncomingArgSize();
}
```

では、 LowerCall に戻りましょう。 IsTailCall がTrueとなり、末尾関数呼び出し最適化が有効として読み進めます。

末尾関数呼び出し最適化が有効の場合は、新たにスタックフレームを作らないので CALLSEQ_START は生成しません。

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

```
  // 関数呼び出しに必要な処理がここから開始されることを意味する
  if (!IsTailCall)
    Chain = DAG.getCALLSEQ_START(Chain, NextStackOffset, 0, DL);
```

いよいよ TailCall ノードを挿入します。上記で使用した MYRISCVXISD::TailCall ノードも定義しておく必要があります。 これは MYRISCVXInstrInfo.td で新たなノードとして定義します。MYRISCVXCallSeqStart や MYRISCVXCallSeqEnd のように、 TailCall ノードもノードとして定義しておきます(あとで除去するため)。

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

```
  // カスタムTailCallノードを挿入
  if (IsTailCall) {
    return DAG.getNode(MYRISCVXISD::TailCall, DL, MVT::Other, Ops);
  }
```

この MYRISCVXISD::TailCall は MYRISCVXInstrInfo.td の中でパタンとして変換されます。つまり、

1. LowerCall 内で MYRISCVXISD::TailCall ノードを生成する。
2. MYRISCVXISD::TailCall ノードは MYRISCVXInstrInfo.td で定義した生成パタンに基づいて変換される。
3. 変換されたパタンに基づいて命令が生成される。

という手順になります。では具体的に MYRISCVXInstrInfo.td でどのような変換ノードを定義すれば良いのか見てみます。

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

```
// Tail call
def MYRISCVXTailCall : SDNode<"MYRISCVXISD::TailCall", SDT_MYRISCVXCall,
                              [SDNPHasChain, SDNPOptInGlue, SDNPOutGlue,
SDNPVariadic]>;
```

まずはカスタムノードとして MYRISCVXTailCall ノードを定義しました。MYRISCVXTargetLowering::LowerCall 内で末尾再帰が発見されるとこのノードに変換されるコードについては先ほど見ました。 ではこの MYRISCVXTailCall はどのように別のノードに変換されるのかを見てみます。 やり方は通常の関数コールと似ています。

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

```
let isCall = 1, isTerminator = 1, isReturn = 1, isBarrier = 1, hasDelaySlot = 0,
    hasExtraSrcRegAllocReq = 1 in {
  def PseudoTAILCALL : MYRISCVXPseudo<(outs), (ins call_symbol:$target),
"tail\t$target", []>;
}
```

疑似命令として PseudoTAILCALL と PseudoTAILCALLReg ノードを定義しました。 これらのノードはRISC-VのABI定義に則りアセンブリとしては疑似命令 tail $target と表示されます。次にこのノードの変換方法について定義します。

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

```
// カスタムTail CallノードはTail疑似命令に変換される
def : Pat<(MYRISCVXTailCall tglobaladdr:$dst),  (PseudoTAILCALL tglobaladdr:$dst)>;
def : Pat<(MYRISCVXTailCall texternalsym:$dst), (PseudoTAILCALL texternalsym:$dst)>;
```

PseudoTAILCALLReg について変換方法を示しました。 MYRISCVXTailCall カスタムノードは tglobaladdr および texeternalsym の型の引数について、 PseudoTAILCALLReg による変換が可能であることを示しています。 こうして最終的に MYRISCVXTailCall カスタムノードは tail 疑似命令に変換されます。

# [LowLayer] 末尾再帰を使用した場合とそうでない場合の生成命令比較

ここまでで、末尾関数呼び出し最適化を適用した場合とそうでない場合についてアセンブリ命令を生成してみます。

まずは末尾再帰を適用しない場合です。

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

- 末尾関数呼び出し最適化を適用しない場合の生成されたアセンブリ命令 `tailcall.myriscvx64.static.S`

```
tail_call_main:                         # @tail_call_main

# %bb.0:                                # %entry
        # スタックフレームを作る。
        addi    x2, x2, -48
        # レジスタのスタック退避
        sd      x1, 40(x2)              # 8-byte Folded Spill
        sd      x2, 32(x2)              # 8-byte Folded Spill
...
        addi    x10, x19, 0
        # tail_call_funcの呼び出しは通常の関数呼び出し。
        call    tail_call_func
        # 関数から戻るときはスタックからレジスタを戻して後片付け。
        ld      x20, 0(x2)              # 8-byte Folded Reload
        ld      x19, 8(x2)              # 8-byte Folded Reload
        ld      x18, 16(x2)             # 8-byte Folded Reload
        ld      x9, 24(x2)              # 8-byte Folded Reload
        ld      x2, 32(x2)              # 8-byte Folded Reload
        ld      x1, 40(x2)              # 8-byte Folded Reload
        # スタックフレームを元に戻す。
        addi    x2, x2, 48
        ret
```

`tail_call_func` の呼び出し部分は通常の `call` 疑似命令が使用されていることが分かります。

続いて末尾再帰を適用します。先ほどコピーして作成した `tailcall.riscv64.static.bc.tail.ll` を使用します。

```
$ ${BUILD}/bin/llc -march=myriscvx64    \
    --debug -tailcallopt -relocation-model=static -filetype=asm \
    -o tailcall.myriscvx64.static.tail.S \
    tailcall.riscv64.static.bc.tail.ll
# 入力ファイルとしてtailcall.riscv64.static.bc
の代わりにtailcall.riscv64.static.bc.tail.llを指定した。
# llcはバイナリコードファイル(bcファイル)だけでなく、テキスト形式のLLVM
IRファイルも受け取ることができるので
# 便利です。
```

- 末尾関数呼び出し最適化を適用しない場合の生成されたアセンブリ命令 `tailcall.myriscvx64.static.tail.S`

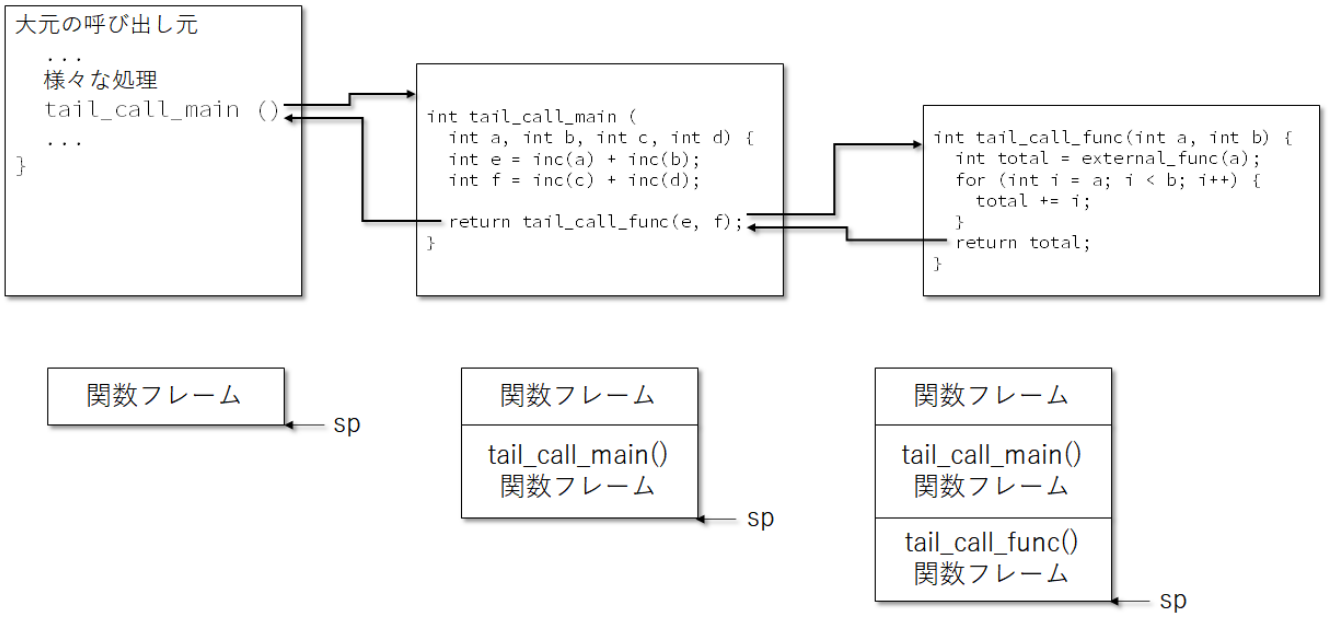
`tail_call_main` のみ抜粋します。

```
tail_call_main:                         # @tail_call_main

# %bb.0:                                # %entry
        # スタックフレームを作る。
        addi    x2, x2, -48
        # レジスタのスタック退避
        sd      x1, 40(x2)              # 8-byte Folded Spill
        sd      x2, 32(x2)              # 8-byte Folded Spill
...
        add     x11, x10, x18
        addi    x10, x19, 0
        # 関数から戻るときはスタックからレジスタを戻して後片付け。
        ld      x20, 0(x2)              # 8-byte Folded Reload
        ld      x19, 8(x2)              # 8-byte Folded Reload
        ld      x18, 16(x2)             # 8-byte Folded Reload
        ld      x9, 24(x2)              # 8-byte Folded Reload
        ld      x2, 32(x2)              # 8-byte Folded Reload
        ld      x1, 40(x2)              # 8-byte Folded Reload
        # このタイミングでスタックフレームを元に戻す。
        addi    x2, x2, 48
        # tail疑似命令により
tail_call_funcにジャンプする。ここから先には戻ってくる事は無い。
        # tail_call_funcから直接tail_call_mainの呼び出し先に戻ることを期待している。
        tail    tail_call_func
```

大きく2つの点が見て取れますでしょうか。

- `tail_call_func` の呼び出しは、末尾再帰無しの場合は `call` 疑似命令、末尾再帰ありの場合は `tail` 疑似命令を使用しています。
- どちらの場合もスタックフレームは作りますが、末尾再帰の場合はスタックフレームの破壊はtail疑似命令による `tail_call_func` 関数の呼び出し前です。 つまり、`tail_call_func` 関数は `tail_call_main` の使っていたスタックフレームの領域を再利用します（今回の例では `tail_call_func` は小さいのでスタックフレームを作りませんが）。




(a)末尾再帰を使用しない場合の関数呼び出しの流れと関数フレームの動き

*Figure 14.* 末尾再帰を使用しない場合の関数呼び出しの流れと関数フレームの流れ。関数フレームは関数の入れ子の呼び出し回数だけ作られる。

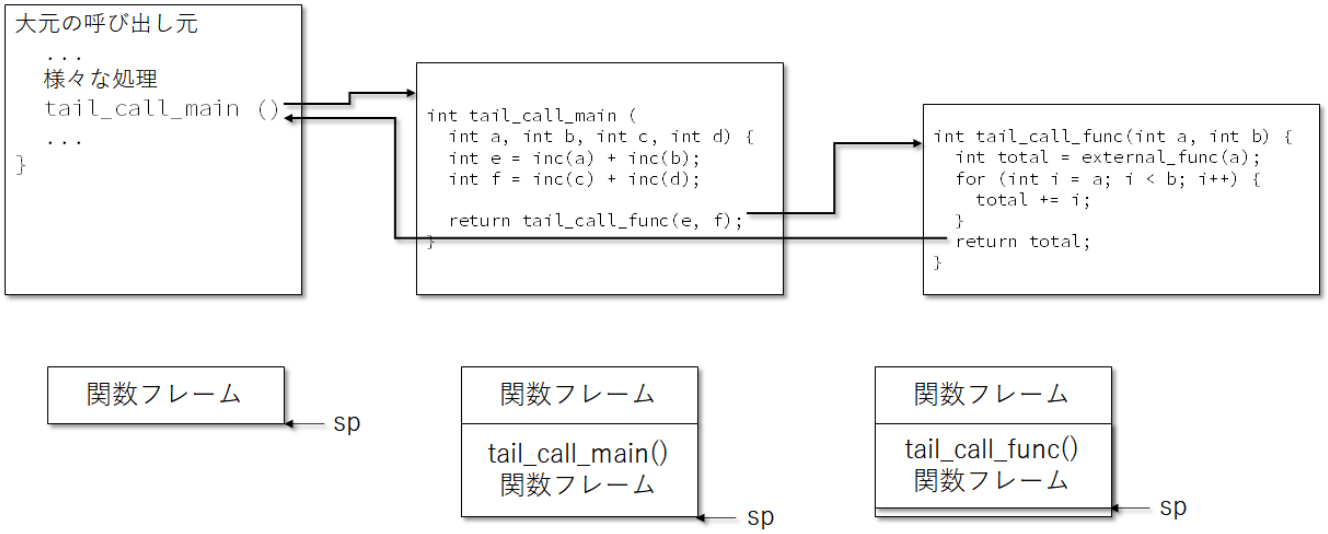



*Figure 15.* 末尾再帰を使用する場合の関数呼び出しの流れと関数フレームの流れ。 `tail_call_main()` の関数フレームの場所は `tail_call_func()` によって再利用される。また、`tail_call_func()` から直接大元の呼び出し元に戻される。

## CALLとTAIL疑似命令はどのようにRISC-V命令に変換されるのか

さて、ここまで命令を生成しておいて今更ですが、疑似命令 `call` と `tail` はどのようなアセンブリ命令へ変換されるのでしょうか。

`tailcall.myriscvx64.static.S` と `tailcall.myriscvx64.static.tail.S` をGCCでコンパイルして生成されるオブジェクトを比較してみます。

```
# callはどのように変換されるのかを観察する。
$ riscv64-unknown-elf-gcc -c tailcall.myriscvx64.static.S
$ riscv64-unknown-elf-objdump -d tailcall.myriscvx64.static.o

# tail はどのように変換されるのかを観察する。
$ riscv64-unknown-elf-gcc -c tailcall.myriscvx64.static.tail.S
$ riscv64-unknown-elf-objdump -d tailcall.myriscvx64.static.tail.o
```

```
# tailcall.myriscvx64.static.oの逆アセンブリ結果
0000000000000008 <tail_call_main>:
...
  5e:   00098513                mv      a0,s3
  62:   00000097                auipc   ra,0x0
                        62: R_RISCV_CALL        tail_call_func
                        62: R_RISCV_RELAX       *ABS*
  66:   000080e7                jalr    ra
  6a:   6a02                    ld      s4,0(sp)
```

```
# tailcall.myriscvx64.static.tail.o
0000000000000008 <tail_call_main>:
...
  6c:   70a2                    ld      ra,40(sp)
  6e:   6145                    addi    sp,sp,48
  70:   00000317                auipc   t1,0x0
                        70: R_RISCV_CALL        tail_call_func
                        70: R_RISCV_RELAX       *ABS*
  74:   00030067                jr      t1
```

`call` 疑似命令は `jalr` 命令に、 `tail` 疑似命令は `jr` 命令に置き換わっていることが分かります。 `jalr` と `jr` の違いは `RA` レジスタに戻り先アドレスを格納するかどうかの違いですから、`tail` はraレジスタを更新しません。 これはつまり「自分の所に戻ってくるな」ということを意味します。 `tail_call_func` の実行が終わったら、 `tail_call_main` を呼び出した大元の関数にすぐさま戻ってくることを期待しているのです（Figure 16, Figure 17）

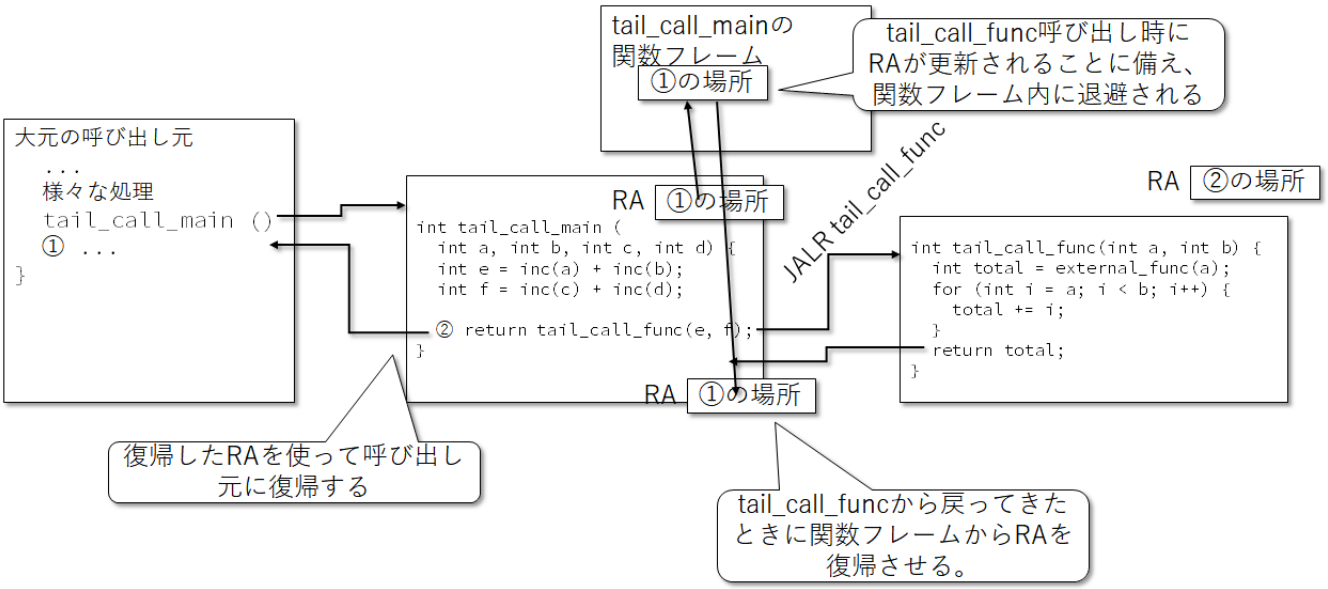



*Figure 16.* 末尾再帰を使用しない場合の関数呼び出しの流れ*ra*レジスタの動き。*tail_call_func*の呼び出し時に*JALR*により*RA*が更新されてしまうことに備え、□の場所を格納している*RA*の値をあらかじめ関数フレーム内に退避している。*tail_call_func*の実行が終了すると関数フレームから*RA*を復帰させ、大元の呼び出し元に戻る。

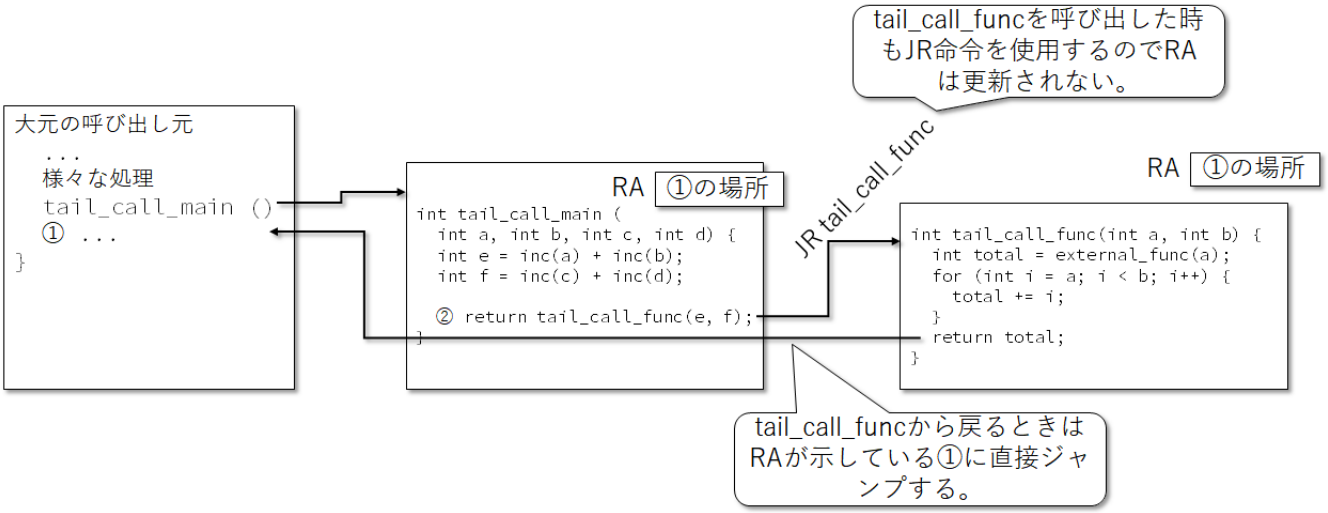



*Figure 17.* 末尾再帰を使用する場合の関数呼び出しの流れ*ra*レジスタの動き。関数*tail_call_func*を呼び出すときに*JALR*命令を使用せず*JR*命令を使用するため*ra*レジスタは更新されない。*tail_call_func*から戻るときは*RA*に格納されている□（おおもとの呼び出し元）に直接ジャンプする。