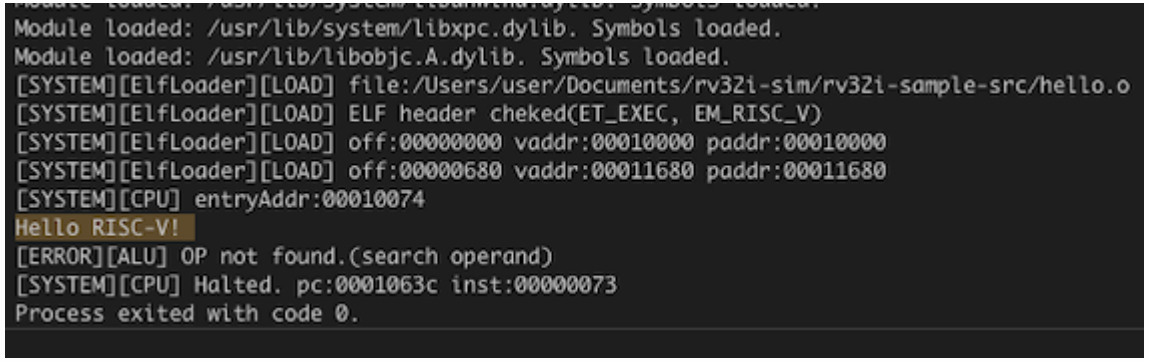
RISC-Vを実装してみる

最小限の命令セット(rv32i)でCPUエミュレータを作ってみた

作ったもの

- rv32iのエミュレータをc++で書きました
  - https://github.com/kamiyaowl/rv32i-sim
  - Close Issueに人間デバッガのメモが残されています...
- ~~出力が地味なので~~成果物として仕様と実装について書き留めておきます
  - 実装時系列で書いているのでまとまりがない点はご容赦ください



RISC-V の特徴

オープンな命令セット(ISA)

- x86-32みたいにベンダーIP(特許怖い)ではなく、公開されている → ***だれでも使える***
- だけど出資企業は多数あり安心(Google, Huawei, IBM, MS, Sumsung, ...)

RISC-V の特徴

オープンな命令セット(ISA)

- x86-32みたいにベンダーIP(特許怖い)ではなく、公開されている → ***だれでも使える***
- だけど出資企業は多数あり安心(Google, Huawei, IBM, MS, Sumsung, ...)

モジュラーISA

- 基本命令セット+機能ごとの命令セット で構成されている
- 世代を重ねるごとに命令がモリモリにならずに済みそう
  - インクリメンタルISAと呼ばれている

実装目標: *"Hello RISC-V!"*を出力できる

Cで書いたコードのコンパイル生成物を実行する

- 命令セット: ***rv32i***
  - ***i***: 基本整数命令のみを実装 (他色々は一切無視)

実装目標: *"Hello RISC-V!"*を出力できる

Cで書いたコードのコンパイル生成物を実行する

- 命令セット: ***rv32i***
  - ***i***: 基本整数命令のみを実装 (他色々は一切無視)
- メモリ、キャッシュ、周辺ペリフェラル: ***適当にエミュレート***
  - vaddrだけしっかりマップしてあげれば行けそう
  - UART TXバッファを`0x10000000`に配置(putcharしてあげる)

実装目標: *"Hello RISC-V!"*を出力できる

Cで書いたコードのコンパイル生成物を実行する

- 命令セット: ***rv32i***
  - ***i***: 基本整数命令のみを実装 (他色々は一切無視)
- メモリ、キャッシュ、周辺ペリフェラル: ***適当にエミュレート***
  - vaddrだけしっかりマップしてあげれば行けそう
  - UART TXバッファを`0x10000000`に配置(putcharしてあげる)
- プログラムのロード: ***最低限実装する***
  - ELFローダを作って、該当するvaddrにロードする

レジスタの実装

まずはどんなレジスタを持っているか確認

レジスタ>構成

- 32本の32bitのレジスタ: `x[0] ~ x[31]`
  - `x[0]`: Zeroレジスタ(読み出すと常に`0x0`/書き込みデータは破棄)
  - `x[1]~x[31]`: 特に区別なし(アセンブラレベルでは番号ごと推奨される用途あり)
- 32bitプログラムカウンタ
  - 加算命令などで直接参照することはできない
  - `auipc`, `jal`, `jalr`などで操作

レジスタ>実装

std::vectorで管理

```
class Reg {
  protected:
    vector<T> x;
    T pc;
  public:
    const size_t XLEN = 32;
    void reset() {
        x = vector<T>(XLEN, 0x0);
        pc = 0;
    }
```

レジスタ>実装

レジスタとPCへの読み書きと、PCのインクリメントを実装

```
T    read(uint8_t addr);
void write(uint8_t addr, T data);
T    read_pc();
void write_pc(T data);
T    get_pc_offset();
void incr_pc();
```

※*T - テンプレートにしているが、uint32_tに固定などで良さそう*

命令デコーダの実装

どんな命令があるのか、どうやって見分けるのか

命令デコーダ>形式

基本整数命令には6種類に分けられる

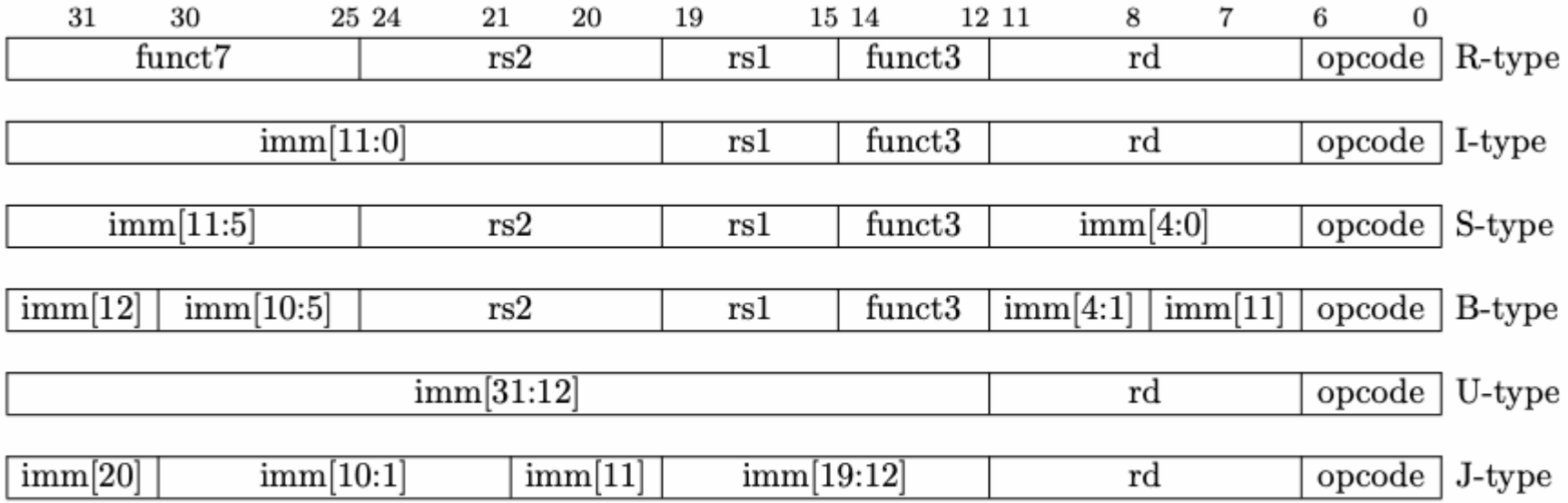


Figure 2.3: RISC-V base instruction formats showing immediate variants.

命令デコーダ>形式

- `opcode`, `funct3`, `funct7`
  - 命令の種類判別に使う
- `rs1`, `rs2`: Register Source
  - 計算の元データを読み込むレジスタを指定(`x[0]` ~ `x[31]`)
- `imm`: Immediate
  - 計算に使用する即値
  - 命令によってビット幅が異なったり、符号拡張が必要だったりする
- `rd`: Register Destination
  - 計算結果(など)を格納するレジスタを指定
  - `pc`は参照できない、`x[0]`を指定したら結果は捨てる

命令デコーダ>分岐順序

命令の決定

- opcode(レジスタ演算、即値演算、Load、Store、分岐、他などで分かれる)
  - funct3(だいたいここで決まる)
  - funct7(add/sub, srl/sraなど似た命令はここで決まる)

命令デコーダ>分岐順序

命令の決定

- opcode(レジスタ演算、即値演算、Load、Store、分岐、他などで分かれる)
  - funct3(だいたいここで決まる)
  - funct7(add/sub, srl/sraなど似た命令はここで決まる)

例(funct3, funct7フィールドがある場合)

- opcode(0b0110011): [`add,sub,sll,slt,sltu,xor,srl,sra,or,and`]
- funct3(0b000): [`add,sub`] → funct7(0b0100000): `sub`

命令デコーダ>実装

opでの検索例。opcodeで単純に検索している

`inst`: pcの指している命令の生の値, `instructions`: 定義した命令リスト

```
uint8_t opcode = (inst >> 0) & 0x7f;
vector<Inst<DATA, ADDR>> filter_op;
std::copy_if(
  instructions.begin(), instructions.end(),
  std::back_inserter(filter_op),
  [&opcode] (const Inst<DATA, ADDR>& i) {
    return opcode == i.opcode;
  }
);
```

命令デコーダ>実装

フィールドの分解は命令が決まれば、読み方が確定する

```
case ImmType::S:
    args.imm_raw =
        (((inst >> 25) & 0x7f) << 5) |
        ((inst >> 7) & 0x1f);
    args.funct3  = (inst >> 12) & 0x7;
    args.rs1     = (inst >> 15) & 0x1f;
    args.rs2     = (inst >> 20) & 0x1f;

    args.rd      = 0x0;
    args.funct7  = 0x0;
    args.imm_signed = convert_signed(args.imm_raw, 12);
```

命令デコーダ>実装(符号拡張)

即値(imm)はsignedとして演算に使う場合があるので、計算しておく

`imm`: 即値の値, `bit_width`: `imm`のデータ幅

```
int32_t convert_signed(uint32_t imm, size_t bit_width)
  size_t shift = (32 - bit_width);
  uint32_t shifted = imm << shift;
  int32_t  shifted_sign = static_cast<int32_t>(shifted);
  int32_t dst = shifted_sign >> shift;
  return dst;
}
```

命令の実装

実際の演算はどうなってるか

命令>種類と概要

- (R-Type) レジスタ-レジスタ間演算: `0b0110011`
  - [`rs1`, `rs2`]で計算をして、`rd`に結果を書き込み

命令>種類と概要

- (R-Type) レジスタ-レジスタ間演算: `0b0110011`
  - [`rs1`, `rs2`]で計算をして、`rd`に結果を書き込み
- (I-Type) レジスタ-即値間演算: `0b0010011`
  - [`rs1`, `imm`]で計算をして、`rd`に結果を書き込み

命令>種類と概要

- (R-Type) レジスタ-レジスタ間演算: `0b0110011`
  - [`rs1`, `rs2`]で計算をして、`rd`に結果を書き込み
- (I-Type) レジスタ-即値間演算: `0b0010011`
  - [`rs1`, `imm`]で計算をして、`rd`に結果を書き込み
- (S-Type) Store: `0b0100011`
  - Memのアドレス(`rs1`+`signed(imm)`)に`rs2`を加工(byte mask等)して書き込み

命令>種類と概要

- (R-Type) レジスタ-レジスタ間演算: `0b0110011`
  - [`rs1`, `rs2`]で計算をして、`rd`に結果を書き込み
- (I-Type) レジスタ-即値間演算: `0b0010011`
  - [`rs1`, `imm`]で計算をして、`rd`に結果を書き込み
- (S-Type) Store: `0b0100011`
  - Memのアドレス(`rs1`+`signed(imm)`)に`rs2`を加工(byte mask等)して書き込み
- (I-Type) Load: `0b0000011`
  - Memのアドレス(`rs1`+`signed(imm)`)から読み出し、加工して`rd`に書き込み

命令>種類と概要

- (B-Type) Branch: `0b1100011`
  - [`rs1`, `rs2`]が特定の条件を満たしたら、`pc`に`pc`+`signed(imm)`を設定

命令>種類と概要

- (B-Type) Branch: `0b1100011`
  - [`rs1`, `rs2`]が特定の条件を満たしたら、`pc`に`pc`+`signed(imm)`を設定
- (I-Type) jalr: `0b1100111`
  - `rd`に`pc + 4`を書き込み, `pc`に`rs1`+`signed(imm)`を設定

命令>種類と概要

- (B-Type) Branch: `0b1100011`
  - [`rs1`, `rs2`]が特定の条件を満たしたら、`pc`に`pc`+`signed(imm)`を設定
- (I-Type) jalr: `0b1100111`
  - `rd`に`pc + 4`を書き込み, `pc`に`rs1`+`signed(imm)`を設定
- (J-Type) jal: `0b1101111`
  - `rd`に`pc + 4`を書き込み, `pc`に`pc`+`signed(imm)`を設定

命令>種類と概要

- (B-Type) Branch: `0b1100011`
  - [`rs1`, `rs2`]が特定の条件を満たしたら、`pc`に`pc`+`signed(imm)`を設定
- (I-Type) jalr: `0b1100111`
  - `rd`に`pc + 4`を書き込み, `pc`に`rs1`+`signed(imm)`を設定
- (J-Type) jal: `0b1101111`
  - `rd`に`pc + 4`を書き込み, `pc`に`pc`+`signed(imm)`を設定
- (U-Type) auipc: `0b0010111`
  - `rd`に`pc`+`signed(imm)`を設定

命令>種類と概要

- (U-Type) lui: `0b0110111`
  - `rd`に`imm`を設定

命令>実装

Instクラスを定義, 実際の処理はprocessに委譲

```
template<typename DATA, typename ADDR>
class Inst {
    public:
        string name;
        uint8_t opcode;
        uint8_t funct3;
        uint8_t funct7;
        ImmType immType;
        function<Process<DATA, ADDR>> process;
```

命令>実装

Instクラスの実行は、命令のパース→`this->process`に丸投げ

- `parse_args`: 先程実装した命令デコードする関数
- `args`に`rs1,rs2,rd,imm, ...`情報が入っている

```
    void run(Reg<DATA>& reg, Mem<DATA, ADDR>& mem, DATA inst) {
        Args args;
        parse_args(inst, this->immType, args);
        this->process(reg, mem, args);
    }
}
```

命令>実装

命令種類ごとに共通処理をラップ。`p: func<DATA(DATA)>`などを外から指定する

```
inline Inst<DATA, ADDR> alu_32i_s_inst(string name, /* 中略 */ ...) {
return Inst<DATA, ADDR>(
    name,0b0100011,funct3,0x0,ImmType::S,
    [&] (Reg<DATA>& reg, Mem<DATA, ADDR>& mem, const Args args) {
        ADDR addr = reg.read(args.rs1) + args.imm_signed;
        DATA data = p(reg.read(args.rs2));

        mem.write(addr, data);
        reg.incr_pc();
    }
);
```

命令>実装

先の共通関数で同一`opcode`の命令を実装。全て集めて`instructions`として定義

```
using S    = int32_t;
using ADDR = uint32_t;
alu_32i_r_inst<S, ADDR>(
  "add",
  0b000,
  0b0000000,
  [] (S a, S b) { return a + b; }
),
```

ラムダ式は最高だ

命令>実装

同じ手順で他の命令も作成(uintの明示が必要なところは`static_cast<U>`で)

```
"add"   , 0b000, 0b0000000, [] (S a, S b) { return a + b; }),
"sub"   , 0b000, 0b0100000, [] (S a, S b) { return a - b; }),
"sll"   , 0b001, 0b0000000, [] (S a, S b) { assert(b > -1); return stati
"slt"   , 0b010, 0b0000000, [] (S a, S b) { return a < b ? 0x1 : 0x0; })
"sltu"  , 0b011, 0b0000000, [] (S a, S b) { return static_cast<U>(a) < s
"xor"   , 0b100, 0b0000000, [] (S a, S b) { return static_cast<U>(a) ^ s
"srl"   , 0b101, 0b0000000, [] (S a, S b) { assert(b > -1); return stati
"sra"   , 0b101, 0b0100000, [] (S a, S b) { assert(b > -1); return a >>
"or"    , 0b110, 0b0000000, [] (S a, S b) { return static_cast<U>(a) | s
"and"   , 0b111, 0b0000000, [] (S a, S b) { return static_cast<U>(a) & s
```

命令>実装

`jalr`などは、`imm`を符号拡張する必要があるので次のようにしてある。

`imm_signed`は符号拡張済なので、`int32_t`同士の演算になっている。

```
"jalr", 0b1100111, 0x0, 0x0, ImmType::I,
[] (Reg<S>& reg, Mem<S, ADDR>& mem, const Args args) {
    reg.write(args.rd, reg.read_pc() + reg.get_pc_offset());

    S rs1 = reg.read(args.rs1);
    S dst = rs1 + args.imm_signed;
    reg.write_pc(dst);
}
```

Mem

rv32iとして実装すべきところはもう終わった...

Mem>実装

`std::map<uint32_t, int32_t>`で書き込まれたデータを返すようにした

```
class Mem {
    private:
        std::map<ADDR, uint8_t> mem;
    public:
        DATA read_byte(ADDR addr) {
            if (mem.count(addr) == 0) {
                sim::log::warn("[MEM] uninitialized mem access at %08x\n
                mem[addr] = 0xa5; // 本来ランダム初期化されるので
            }
            return mem[addr];
        }
```

Mem>実装

Mapに追記するだけ。UartTxバッファに書き込みがあった場合は即時stdoutする。

```
const ADDR UART_PERIPHERAL_BASE_ADDR = 0x10000000;
const ADDR UART_PERIPHERAL_SIZE      = 0x00000001;
void write_byte(ADDR addr, DATA data) {
    mem[addr] = data & 0xff;
    // UART
    if (this->UART_PERIPHERAL_BASE_ADDR <= addr &&
        (addr < this->UART_PERIPHERAL_BASE_ADDR
                + this->UART_PERIPHERAL_SIZE)) {
        sim::log::uart(static_cast<char>(mem[addr]));
    }
}
```

ELF Loader

バイナリを変換して読み込むのは不格好なので作ってみた

ELF Loader>やるべきこと

- ELF形式のファイル (`readelf`か`objdump`が便利)
  - 頭はELF Header, Program Header, Section Headerで構成されている
  - 先頭の`7f 45 4c 46`はマジックナンバー、書式チェックに使える

ELF Loader>やるべきこと

- ELF形式のファイル (`readelf`か`objdump`が便利)
  - 頭はELF Header, Program Header, Section Headerで構成されている
  - 先頭の`7f 45 4c 46`はマジックナンバー、書式チェックに使える

- ~~Mem周りの実装が適当なので、~~Sectionは気にせずProgram Headerに着目
  - ***LOAD指定された領域を、指定通りのvaddrに読み込んであげる***
  - ***エントリポイントのvaddrを控え、起動時のpcに設定***

ELF Loader>やるべきこと

- ELF形式のファイル (`readelf`か`objdump`が便利)
  - 頭はELF Header, Program Header, Section Headerで構成されている
  - 先頭の`7f 45 4c 46`はマジックナンバー、書式チェックに使える

- ~~Mem周りの実装が適当なので、~~Sectionは気にせずProgram Headerに着目
  - ***LOAD指定された領域を、指定通りのvaddrに読み込んであげる***
  - ***エントリポイントのvaddrを控え、起動時のpcに設定***

どんな情報が得られるか`$ riscv32-unknown-elf-readelf -a`してみる

ELF Loader> Header

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10074
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4632 (bytes into file)
```

ELF Loader> Header

```
Flags:                             0x0
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         2
Size of section headers:           40 (bytes)
Number of section headers:         14
Section header string table index: 13
```

- ***Entry point address***(`e_entry`): 0x10074 ← 探してたやつ
- Start of program headers(`e_phoff`): 52 ← 探してた
- Size of program headers(`e_ehsize`): 52
- Number of program headers(`e_phnum`): 2 ← 領域いくつあるかは大事

ELF Loader> Header読み込み

gnu-toolchainにelf.hで定義されているとおりに読めばいい(RISC-Vに限らず)

```
typedef struct elf32_hdr{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    ...
```

ELF Loader> Header読み込み

`std::ifstream`で順番に読み出すだけなので、特筆することはなさそう

```
std::ifstream ifs(elf_path, ifstream::in | ifstream::binary);

Elf32_Ehdr hdr = {};
ifs.read((char*)(&hdr.e_ident[0]),  EI_NIDENT);
ifs.read((char*)(&hdr.e_type),      sizeof(hdr.e_type));
ifs.read((char*)(&hdr.e_machine),   sizeof(hdr.e_machine));
ifs.read((char*)(&hdr.e_version),   sizeof(hdr.e_version));
...
assert(hdr.e_ident[0] == 0x7f);
assert(hdr.e_ident[1] == 'E');
assert(hdr.e_ident[2] == 'L');
```

ELF Loader> Program Header

e_phnumで指定した数だけ、以下のエントリが連続して記述されている。

```
typedef struct elf32_phdr{
    Elf32_Word    p_type;   // 領域の種類(ロード可能，動的リンク，補足等...)
    Elf32_Off     p_offset; // セグメント先頭へのファイル先頭からのオフセット
    Elf32_Addr    p_vaddr;  // メモリ上の仮想アドレス
    Elf32_Addr    p_paddr;  // 物理アドレスとして予約されている→使わない
    Elf32_Word    p_filesz; // セグメントのファイルイメージのバイト数
    Elf32_Word    p_memsz;  // 仮想メモリイメージでのバイト数→filesz使うので不
    Elf32_Word    p_flags;  // 領域の読み書き実行(X/W/R)のフラグ
    Elf32_Word    p_align;  // セグメントのアライン
} Elf32_Phdr;
```

ELF Loader> Program Header

- ~~動的リンクはさておき~~以下のデータが対応するように読み込んであげる
- ELFファイルの読み取り領域
  - Offset(`p_offset`)
  - Offset(`p_offset`) + FileSiz(`p_filesz`)
- Memへの展開先
  - VirtAddr(`p_vaddr`)
  - VirtAddr(`p_vaddr`) + MemSiz(`p_memsz`)

```
Program Headers:
  Type Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD 0x000000 0x00010000 0x00010000 0x00680 0x00680 R E 0x1000
  LOAD 0x000680 0x00011680 0x00011680 0x00444 0x00460 RW  0x1000
```

ELF Loader> Program Header実装

`std::seekg`を使って指定領域を読み出して、Memに書き込むCallbackを叩く

```
if (phdr.p_type == 1) { // PT_LOAD
    auto current = ifs.tellg();
    ifs.seekg(phdr.p_offset, std::ifstream::beg);
    // Callbackの実装がしょぼいのでがんばって1byteずつ読むよ...
    for(int i = 0 ; i < phdr.p_filesz ; ++i) {
        char buf;
        ifs.read(&buf, 1);
        write(phdr.p_vaddr + i, static_cast<uint8_t>(buf));
    }
    ifs.seekg(current, std::ifstream::beg);
}
```

ELF Loader> Section Header[参考]

人力gdbするときにめっちゃ見た。直接バイナリ追うほうがメインだったけど

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000005fc  00010074  00010074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000010  00010670  00010670  00000670  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .eh_frame     00000004  00011680  00011680  00000680  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .init_array   00000004  00011684  00011684  00000684  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

ELF Loader> Section Header[参考]

```
Idx Name          Size      VMA       LMA       File off  Algn
  4 .fini_array   00000004  00011688  00011688  00000688  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .data         00000428  00011690  00011690  00000690  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  6 .sdata        0000000c  00011ab8  00011ab8  00000ab8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .bss          0000001c  00011ac4  00011ac4  00000ac4  2**
```

完成

細かいところは実装を参考にしてください...

動かしてみる

動かす>実行するコードの記述

`"Hello RISC-V"`をUART TXバッファに書くコードを作成

```
#include <stdint.h>

#define UART_PERIPHERAL_BASE_ADDR (0x10000000)

void uart_tx(const char c) {
    volatile uint8_t* uartTxPtr = (volatile uint8_t*)UART_PERIPHERAL_BAS
    // TODO: もし実機を完全に模倣するなら送信バッファFullフラグで待ったりする
    // TxBufに値を書き込み
    *uartTxPtr = (uint8_t)c;
}
```

動かす>実行するコードの記述

```
void print(const char* str) {
    for(int i = 0 ; str[i] != '\0' ; ++i) {
        uart_tx(str[i]);
    }
}
int main(void) {
    const char* hello = "Hello RISC-V! \n";
    print(hello);

    return 0;
}
```

動かす>コンパイル

toolchainを何度もconfigureし直していたので、うんざりしてDockerfile作成

```
FROM ubuntu:18.04

ENV RISCV=/opt/riscv
ENV PATH=$RISCV/bin:$PATH
WORKDIR $RISCV

RUN apt update
RUN apt install -y autoconf automake autotools-dev curl \
                   libmpc-dev libmpfr-dev libgmp-dev gawk \
                   build-essential bison flex texinfo gperf \
                   libtool patchutils bc zlib1g-dev libexpat-dev
```

動かす>コンパイル

内容としては、riscv-gnu-toolchainの手順通りにビルドしているだけ

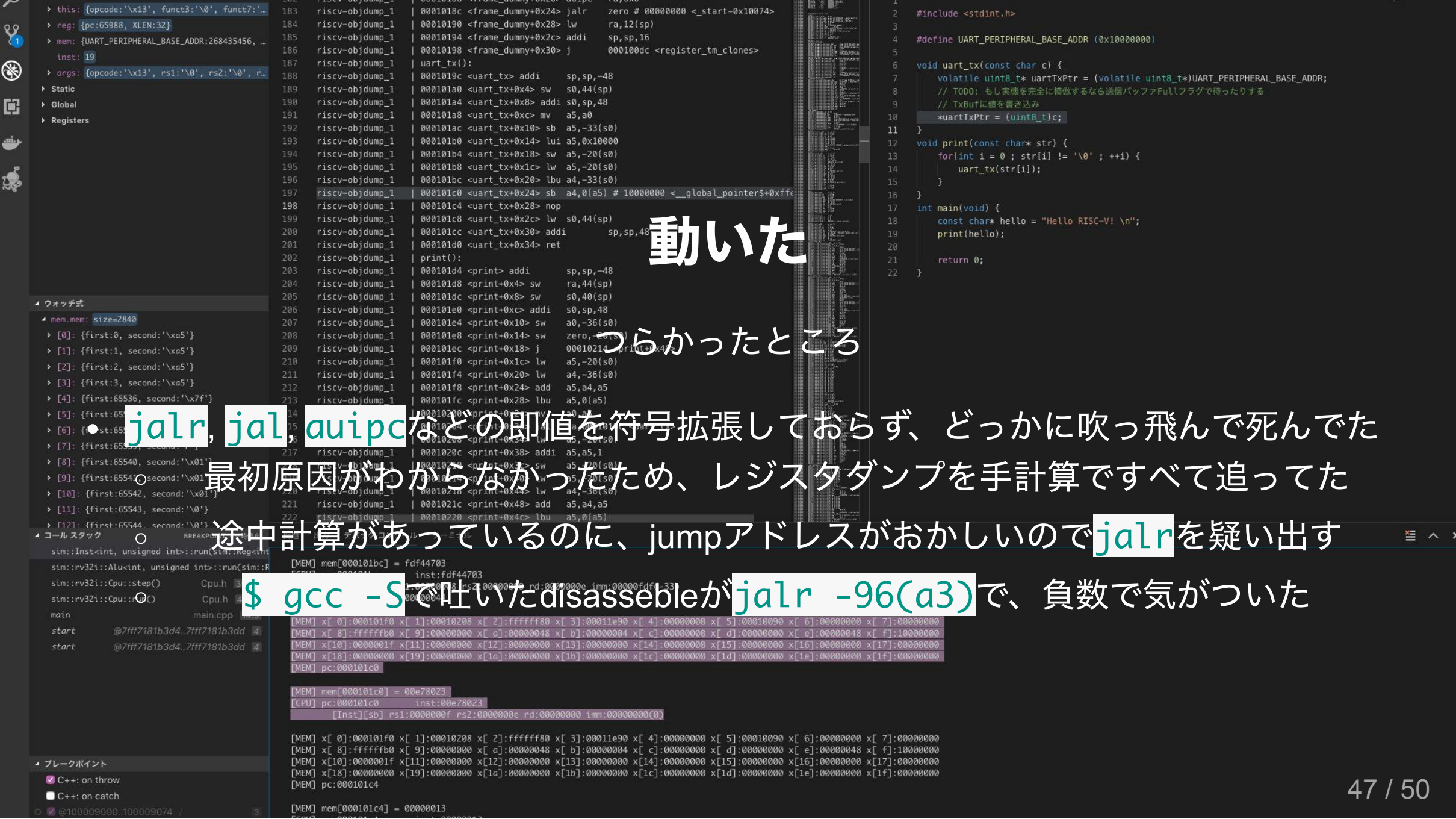
```
RUN apt install -y git

RUN git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

RUN cd riscv-gnu-toolchain && ./configure --prefix=/opt/riscv --with-arc

WORKDIR /work
```

動かす>コンパイル

dockerコマンドを直で叩きたくない教徒なので、docker-compose.ymlを作成

`$ docker-compose up`でhello.oを作成したら、***いよいよ読み込ませて実行する。***

```
services:
  riscv-compile:
    build: .
    volumes:
      - ./:/work
    command:
      riscv32-unknown-elf-gcc \
        -march=rv32i -mabi=ilp32 \
        -o /work/hello.o /work/hello.c
```


動いた
つらかったところ
jalr, jal, auipcなどの即値を符号拡張しておらず、どっかに吹っ飛んで死んでた
最初原因がわからなかったため、レジスタダンプを手計算ですべて追ってた
途中計算があっているのに、jumpアドレスがおかしいのでjalrを疑い出す
$ gcc -Sで吐いたdisassebleがjalr -96(a3)で、負数で気がついた

まとめ

- RISC-Vは簡単ですごい
  - エミュレータが1週間ぐらいでできた(半分ぐらいtoolchainのビルドしてた)
  - いろいろな実装があるので覗いてみると楽しい
- どう伸びるかわからないけど、**ISA自体は**綺麗にまとまっている
  - モジュラーISAなので欲しい機能だけ作ればOK(rv32imfd + 独自命令とか)
  - 関係ないけど学生実験MIPSとかやってたなぁとか思い出した
- C++わからん
  - 高まりたい

引用/参考

- riscv.org
- riscv/riscv-isa-manual - Github
- RISC-V原典 オープンアーキテクチャのススメ: 日経BP社
  - 著: デイビッド・パターソン
  - 著: アンドリュー・ウォーターマン
  - 訳: 成田 光影

Fin.