ざっくりRISC-Vアセンブラ入門

環境構築から関数呼び出しまで

技術書典#6

omoikane

第 1 章

環境の整備

1.1　はじめに

本書では、非常にシンプルな命令セットと言われている、RISC-V のアセンブリ言語を学習していきます。

種本は、坂井弘亮著「熱血！ アセンブラ入門」です。この本では、コンパイラが吐くアセンブリ言語をとにかく読んで推定するという方法がとられていますが、本書では、同じやり方で、RISC-V のアセンブラを学んでいきます。私は、「熱血！ アセンブラ入門」を一通り読んだので、だいたい読み方が分かっていますので、全体を通じて説明が雑だと思われます。アセンブラに慣れていない方は、坂井本を是非読んでみてください。相当おもしろいです。

また、RISC-V といえば、デイビット・パターソン他著「RISC-V 原典」です。この他に、“The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2”（以下、ISA マニュアル）を適宜参照します。ISA マニュアルは、GitHub で公開されています。(https://github.com/riscv/riscv-isa-manual/blob/master/release/riscv-spec-v2.2.pdf)

1.2　コンパイル環境の構築

RISC-V は新しいアーキティクチャーなので、動作する実装はずいぶん増えてきたとはいえまだまだ入手せいが低く、お値段もけっこうします。よって同一ホスト上でコンパイルするのは非常に敷居が高い。そこで本書では、C 言語で書いた簡単なプログラムを、amd64 上の Linux でクロスコンパイルして、逆アセンブリ結果を見て RISC-V のアセンブラ言語を学んでいきます。

それでも、クロスコンパイラの準備は、gcc をソースからコンパイルする必要があり大変です。ですが、幸い Debian GNU/Linux のテスト版 (buster) には RISC-V の gcc が

パッケージになっていますので、このパッケージを使うことにしましょう。

Virtualbox に、Debian GNU/Linux buster (testing) をインストールするか、Docker の debian:buster イメージをベースに、環境を作成するのが良いでしょう。検証できていませんが、Ubuntu cosmic 以降でも良いはずです。apt コマンドで、gcc-riscv64-linux-gnu と、make パッケージがインストールされた Debian or Ubuntu を準備してください。

ちなみに私は Docker で環境を作成しました。私の Dockerfile はこんな感じです。

```
1  FROM debian:buster
2
3  RUN apt-get update && apt-get -y upgrade \
4          && apt-get -y install gcc-riscv64-linux-gnu make
5
6  RUN mkdir -p /var/local/work && chmod 777 /var/local/work
7  VOLUME /var/local/work
8  WORKDIR /var/local/work
9
10 # change to your uid:gid
11 USER 1000:1000
12
13 CMD ["make"]
```

gcc-riscv64-linux-gnu と make をインストールしただけのコンテナですね。あと、逆アセンブルした後のファイルを共有するために、ディレクトリを掘ったり、コマンドを実行するユーザーの uid を変更したりしています。

ここでポイントは、make コマンドを使用していることです。共有しているボリュームに Makefile を置いておくと、Makefile に書かれた処理をコンテナ上で実行してくれます。すなわち、コンテナを変更しなくても任意のコマンドを実行できるように、動作を変更する事ができます。本書では make を使いますが、バッチ処理を行うためだけに使うので、インタープリタならなんでも良いです。お好みのものを使用してください。（シェルや Python のほうがよかったかな...）

1.3　Hello RISC-V アセンブラ

環境が整ったら、動作試験をしてみましょう。Virtualbox に環境を構築した方は、ログイン後、適当なディレクトリで、Hello world をちゃちゃっと書きます。hello.c というファイル名で保存しましょう。

```
#include <stdio.h>
int main(void)
{
  printf("hello world");
  return 0;
}
```

そして以下のコマンドを実行します。

```
riscv64-linux-gnu-gcc -o hello hello.c
riscv64-linux-gnu-objdump -d hello >hello.s
```

1 行目で、hello.c を riscv にクロスコンパイル、2 行目で逆アセンブルしてアセンブリ言語のソースファイルを取得しています。逆アセンブル結果の hello.s の最初のほうは以下のようになると思います。

```
hello: file format elf64-littleriscv
Disassembly of section .plt:

00000000000004b0 <.plt>:
 4b0: 00002397 auipc t2,0x2
 4b4: 41c30333 sub t1,t1,t3
 4b8: b583be03 ld t3,-1192(t2) # 2008 <__TMC_END__>
...
```

gcc のバージョンによって多少差があるかもしれませんが、気にしなくてかまいません。例えば、main 関数の逆アセンブルコードは以下のようになりました。

```
00000000000005c2 <main>:
 5c2: 1101 addi sp,sp,-32
 5c4: ec06 sd ra,24(sp)
 5c6: e822 sd s0,16(sp)
 5c8: 1000 addi s0,sp,32
 5ca: 4789 li a5,2
 5cc: fef42623 sw a5,-20(s0)
 5d0: fec42783 lw a5,-20(s0)
 5d4: 853e mv a0,a5
 5d6: fd1ff0ef jal ra,5a6 <add1>
 5da: fec42783 lw a5,-20(s0)
 5de: 278d addiw a5,a5,3
 5e0: 2781 sext.w a5,a5
```

```
 5e2: 853e mv a0,a5
 5e4: 60e2 ld ra,24(sp)
 5e6: 6442 ld s0,16(sp)
 5e8: 6105 addi sp,sp,32
 5ea: 8082 ret
```

20 ステップくらいのシンプルなコードが生成されているようですね。このように簡単なプログラムを逆アセンブルすることで、アセンブラを学んで行きます。

Docker で環境を作れるような方は、以上の説明でどうにでもできると思いますが、私が使用している Makefile を晒しておきます。参考にしてください。

```
CC=riscv64-linux-gnu-gcc #-O0
CCFLAGS= #-march=rv64imafdc -mabi=lp64d
OBJDUMP=riscv64-linux-gnu-objdump
OBJDUMPFLAGS=-d

BASENAME=$(basename $(SRC))
OBJ=$(BASENAME).o
TARGET=$(BASENAME).s

all: clean $(TARGET)

$(TARGET): $(BASENAME)
	$(OBJDUMP) $(OBJDUMPFLAGS) $^ >$@

$(BASENAME): $(SRC)
	$(CC) $(CCFLAGS) -o $@ $^

clean:
	rm -f $(TARGET) $(OBJ)* $(BASENAME) $(HEX)
```

また、Docker run のオプションもついでに晒しておきましょう。カレントディレクトリに Makefile とソースファイルがあるものとしています。

```
docker run -v $(pwd):/var/local/work [container id]
```

第 2 章

RISC-V Instruction Set Architecture

まず、RISCV の Instruction Set Architecture (以下 ISA) をみておきましょう。

RISC-V ISA は、モジュール型のアーキティクチャーで、基本命令セットに拡張命令セットを追加できるようになっています。そのため、CPU の設計者は、そのシステムに必要な機能だけを CPU に実装することができ、ロジック規模を最小限に抑えることができ、消費電力や開発コストを削減することができます。

基本命令セットを以下に示します。Frozen が Y のところは、もう ISA が変化しないことを示しています。

表 2.1　RISC-V 基本命令セット

Name	Version	Frozen	Remarks
RV32I	2.0	Y	32bit 整数
RV64I	2.0	Y	64bit 整数
RV32E	1.9	N	32bit 整数 組込用
RV128I	1.7	N	128bit 整数

128bit の命令セットとかあるのか...

ちなみに、RV64I は、RV32I に 64bit 演算、ロード・ストア関連などの 10 個ほどの命令が追加されたものです。また、RV32E は、RV32I のレジスタを 16 本にしたものです。

次に拡張命令セットを、表 2.2 に示します。

RISC-V の命令セットで動くプログラムを作成する場合は、どの基本命令セットにどの拡張が実装されているか確認する必要があります。そうしないと、実装されていない命令を実行しようとして CPU がエラーを起こす可能性があります。では、前回作成したクロスコンパイル環境では、どの命令セットがコンパイル可能なのでしょうか。コンパイル時に gcc に-v を渡すと表示されます。

大量に暗黙的に指定されているオプションが出力されますが、その中に、-march とい

表 2.2 RISC-V 拡張命令セット

Name	Version	Frozen	Remarks
M	2.0	Y	Integer Multiplication and Division
A	2.0	Y	Atomic Instructions
F	2.0	Y	Single-Precision Floating-Point
D	2.0	Y	Double-Precision Floating-Point
Q	2.0	Y	Quad-Precision Floating-Point
C	2.0	Y	Compressed Instructions
L	0.0	N	Decimal Floating-Point
B	0.0	N	Bit Manipulation
J	0.0	N	Dynamically Translated Languages
T	0.0	N	Transactional Memory
P	0.1	N	Packed-SIMD Instructions
V	0.2	N	Vector Operations
N	1.1	N	User-Level Interrupts

うオプションがあります。そこに指定されている値が、gcc でコンパイルしている命令セットです。私の環境では、以下のようになりました。

```
1 COLLECT_GCC_OPTIONS='-v' '-march=rv64imafdc' '-mabi=lp64d'
```

どうやら、64bit 基本整数 (rv64i) に、乗算・除算 (m)、アトミック (a)、単精度浮動小数点 (f)、倍精度浮動小数点 (d)、圧縮命令 (c) の拡張を使うコードを生成するようです。

また、-mabi は変数やポインタのサイズを指定します。‘lp64d‘は、long とポインタのサイズが 64bit で、float、double 型の処理にレジスタを使用するということです。

-mabi は、‘ilp32‘、‘ilp32d‘、‘ilp32f‘、‘lp64‘、‘lp64d‘、‘lp64f‘が指定可能です。おそらく、Debian の amd64 のパッケージをインストールしていれば同じになると思います。ホスト側が 32bit の場合は、rv32imafdc になるのではないかと思います。これ以降、ひとまずこのデフォルトの命令セットと ABI でコンパイルしていくことにします。

第 3 章

関数の呼び出しとリターン

3.1　はじめての関数

まずは、関数のコールとリターンを見てみましょう。以下の C ソースコードをコンパイルして逆アセンブルします。

```
1  void null()
2  {
3          return;
4  }
5  int zero()
6  {
7          return 0;
8  }
9  int one()
10 {
11         return 1;
12 }
13 int main()
14 {
15         zero();
16         return 0;
17 }
```

その結果、以下のようなコードが生成されました。関数 null と zero の部分を抜粋してみました。

```
1  00000000000005a6 <null>:
```

```
 5a6: 1141 addi sp,sp,-16
 5a8: e422 sd s0,8(sp)
 5aa: 0800 addi s0,sp,16
 5ac: 0001 nop
 5ae: 6422 ld s0,8(sp)
 5b0: 0141 addi sp,sp,16
 5b2: 8082 ret

00000000000005b4 <zero>:
 5b4: 1141 addi sp,sp,-16
 5b6: e422 sd s0,8(sp)
 5b8: 0800 addi s0,sp,16
 5ba: 4781 li a5,0
 5bc: 853e mv a0,a5
 5be: 6422 ld s0,8(sp)
 5c0: 0141 addi sp,sp,16
 5c2: 8082 ret
```

null は、なにもせず、なにも返却しない関数で、zero はただ 0 を返却するだけの関数です。逆アセンブル結果が格納されている拡張子が s のファイルには、左からプログラムの最初からのオフセット、次に機械語、ニーモニック、オペランドと続きます。

null と zero を比較すると、5 行目の nop が、14、15 行目の内容に変わっており、それ以外は同じであることがわかります。おそらく、ここに C で書いた関数の処理がアセンブリ言語で書かれていると思われます。では 1 行づつ読んでみましょう。

まず関数 null からです。ISA マニュアル 15 ページに nop 命令が定義されています。

> The NOP instruction does not change any user-visible state, except for advancing the pc. NOP is encoded as ADDI x0, x0, 0.（nop 命令は、pc を進めること以外、ユーザー変数を変更しない。NOP は、ADDI x0, x0, 0 に変換される。）

pc はプログラムカウンターで、現在実行している命令のアドレスを指しているレジスタです。ADDI 命令は、おそらく即値 (Immediate) と足し算 (ADD) をする命令です。ISA マニュアルによると、以下のように定義されています。

> ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.（ADDI は、符号拡張された 12 ビットの即値と rs1 の値を加算する。オーバーフローは無視し、単純に加算結果の下位 XLEN ビットを結果とする。addi rd,rs1,0 は、mv rd,rs 疑似命令の実装として使われる。）

XLEN というのは、64 ビットアーキティクチャなら 64、32 ビットアーキティクチャなら 32 が入ります。よって nop は、ADDI x0,x0, 0 となり、x0 レジスタに 0 を足して、結果を x0 に書くという命令になります。x0 レジスタは、常に定数値 0 が入っているゼロレジスタです。ゼロレジスタにゼロを足して結果をゼロレジスタに書いているので、メモリもレジスタの値も変化しません。ただ、pc だけは、次に実行する命令の番地を指すために、RISC-V だと 2 または 4 値が大きくなります。あと、レジスタのコピーに使う mv 命令の実態は、ADDI 命令だったのですね。

さて、コードに戻りましょう。関数 zero の 14 行目と 15 行目は何をやっているのでしょう。

```
14  5ba: 4781 li a5,0
15  5bc: 853e mv a0,a5
```

14 行目は li 命令です。ISA マニュアル p.76 によると

> C.LI loads the sign-extended 6-bit immediate, imm, into register rd. C.LI is only valid when rd ≠ x0. C.LI expands into addi rd, x0, imm[5:0].（C.LI は、符号拡張した即値 imm をレジスタ rd にロードする。C.LI は、rd≠x0 の時のみ有効である。C.LI は、addi rd, x0, imm[5:0] に展開される。）

とのことです。命令の前に C がついているのは圧縮命令を意味しています。実は、RISC-V の命令長は 32bit 固定なのですが、ここまでで出てきた逆アセンブル結果の機械語長が 4 桁、すなわち 16 ビットしかありません。これは、オペランドが 2 つしかなかったり、短いサイズの即値用に命令を別に作ることでプログラムサイズを小さくする圧縮命令が使用されているためです。圧縮命令より実行に必要なメモリサイズを削減することができます。実行時は CPU のデコーダーが 32bit の命令に変換します。

ということで、li 命令は即値をレジスタにロードする命令ということがわかりました。ということは、14 行目の li a5,0 は、レジスタ a5 に 0 を代入していることがわかります。そして 15 行目で、レジスタ a5 の値をレジスタ a0 にコピーしています。

残り 16 行目以降の命令は、null と同じですので、C で書いたコードとは関係の無い、後始末などの処理であると思われます。これらの命令では、a0 レジスタに変更を加えていません。おそらく関数の戻り値の受け渡しは、a0 レジスタが使用されているようです。関数 one の同じ部分を見ると、以下のようになっており、この予想は正しいことがわかります。

```
1  5ca: 4785 li a5,1
2  5cc: 853e mv a0,a5
```

3.2　関数の呼び出し

それでは呼び出し側を見てみましょう。逆アセンブルした main を見てみます。

```
1  00000000000005d4 <main>:
2   5d4: 1141 addi sp,sp,-16
3   5d6: e406 sd ra,8(sp)
4   5d8: e022 sd s0,0(sp)
5   5da: 0800 addi s0,sp,16
6   5dc: fcbff0ef jal ra,5a6 <null>
7   5e0: 4781 li a5,0
8   5e2: 853e mv a0,a5
9   5e4: 60a2 ld ra,8(sp)
10  5e6: 6402 ld s0,0(sp)
11  5e8: 0141 addi sp,sp,16
12  5ea: 8082 ret
```

6 行目で zero にジャンプしているようです。6 行目の機械語長が 32 ビットであることから、jal 命令は圧縮命令ではないようです。ISA マニュアルの p.11 に jal 命令が定義されています。

> The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

まぁいろいろ書いてありますが、rd に pc+4 を代入して、指定したオフセットアドレスにジャンプする命令です。rd に代入される pc+4 というのは、関数の実行が終わったあとに戻ってくるアドレスである、リターンアドレスになります。6 行目で、ra にリターンアドレスを格納しています。また、jal の第 2 引数である 5a4 は、関数 null の最初の命令のオフセット 0x05a4 と一致していることがわかります。

3.3　引数がある関数呼び出し

それでは、引数がある場合はどうでしょう。次の C のソースコードをクロスコンパイルして逆アセンブルします。

```
int add(int a, int b){
        int result;
        result = a + b;
        return result;
}

int main(){
        add(0x123, 0x456);
        return 0;
}
```

逆アセンブル結果を以下に示します。

```
00000000000005a6 <add>:
 5a6: 7179 addi sp,sp,-48
 5a8: f422 sd s0,40(sp)
 5aa: 1800 addi s0,sp,48
 5ac: 87aa mv a5,a0
 5ae: 872e mv a4,a1
 5b0: fcf42e23 sw a5,-36(s0)
 5b4: 87ba mv a5,a4
 5b6: fcf42c23 sw a5,-40(s0)
 5ba: fdc42703 lw a4,-36(s0)
 5be: fd842783 lw a5,-40(s0)
 5c2: 9fb9 addw a5,a5,a4
 5c4: fef42623 sw a5,-20(s0)
 5c8: fec42783 lw a5,-20(s0)
 5cc: 853e mv a0,a5
 5ce: 7422 ld s0,40(sp)
 5d0: 6145 addi sp,sp,48
 5d2: 8082 ret

00000000000005d4 <main>:
 5d4: 1141 addi sp,sp,-16
 5d6: e406 sd ra,8(sp)
 5d8: e022 sd s0,0(sp)
 5da: 0800 addi s0,sp,16
 5dc: 45600593 li a1,1110
 5e0: 12300513 li a0,291
 5e4: fc3ff0ef jal ra,5a6 <add>
 5e8: 4781 li a5,0
```

```
29  5ea: 853e mv a0,a5
30  5ec: 60a2 ld ra,8(sp)
31  5ee: 6402 ld s0,0(sp)
32  5f0: 0141 addi sp,sp,16
33  5f2: 8082  ret
```

まず呼び出し側から見てみましょう。27 行目で関数 add にジャンプしています。25、26 行目の li 命令で引数をレジスタに代入しています。li 命令の第 2 オペランドの 1110 は 0x456、291 は 0x123 です。どうやら引数は、レジスタ a0、a1 を経由して渡されているようです。

次に関数 add を見てみます。2 行目では、呼び出し時のスタックポインタ sp から 48 を引いて、関数 add で使うスタックを 48byte 確保しています。次の 3 行目では、s0 の値を sp+40 のアドレスにコピーして、4 行目で sp+48 を s0 に代入しています。すなわち、sp から s0 の間が関数 add でつかえるスタックになります。

5、6 行目で、レジスタ経由で渡された引数を a4, a5 レジスタにコピーしたあと、7、8、9 行目でスタックに退避しています。で、せっかくスタックに入れた引数を 10、11 行目で a4、a5 レジスタに戻して、12 行目で足し算、結果を a5 レジスタに入れています。13 行目で計算結果の a5 レジスタを、スタックに退避しています。14 行目でまた a5 レジスタに戻して、15 行目で戻り値を渡すために a0 にコピー。16、17 行目では、s0 と sp を呼び出し時の値にもどして、18 行目でリターンしていることがわかります。

関数 add で出てきた命令は、sd、sw、ld、lw です。これらは、sd、sw はレジスタの値を指定されたアドレスに格納する命令です。sd は 8byte、sw は 4byte 書き込みます。同様に、ld、lw は指定されたアドレスの値をレジスタに 8byte、4byte それぞれロードする命令です。

なんか引数を無駄にレジスタに退避させたりしていて、あまり効率的ではないコードが出力されているように思えますが、引数をスタックに退避させるのは、レジスタを節約するためでしょう。また、15 行目で計算結果をスタックに退避しているのは、C のソースにある、‘result=a+b;’ に相当すると思われます。そして、‘return result;’ なので、変数 result に相当するスタックから値を取り出して、リターンするというコードになっていると考えると、C で書いたソースコードどおりのコードが生成されていることがわかります。

あとがき

あー間に合わなかったー

環境づくりと関数呼び出しだけで終わってしまいました.... 本当は、制御構造やヒープへのアクセスとかちゃんと書きたかったのですが、ごめんなさい。時間切れでした。次回続編を書きます。お楽しみに。

RISC-V のアセンブラの話を書こうと思ったのは、バイナリアンと一緒に仕事をしたとき eMMC に入っているファームウェアの実行ファイルを、普通に読んでアセンブリレベルでガンガン改ざんしているところを目撃したからでした。

アラフォーのおじさんだと小さい頃に見たテレビで、コンピュータを機械語をレベルでハッキングするみたいなシーンを見た記憶があるのですが、私よりも一回り若い仲間が、似たようなことをやっているのは衝撃的でした。私もやってみたいと思ってアセンブリ言語を勉強しています。本書はその成果？ の一部になります。

RISC-V のアセンブラを学んで思うのが、アーキテクチャがシンプルな設計で、命令も少ないのですごく読みやすいということです。MIPS の遅延スロットとか、ARM のように pc が 2 つ先のアドレスを指しているとか、へんなクセがないということでしょう。まだ仕事につかえるほど普及していませんが、アセンブリ言語に慣れるといった、初期の学習には最適なアーキテクチャなのかもしれません。また、LLVM で RISC-V サポートが始まったり、Raspberry 財団が RISC-V 基金に加盟したりと、今後の動きも楽しみです。

十分にチェックしたつもりですが、内容に間違いがある可能性があります。もし間違いがありましたらそれは 100% 私の責任です。何かありましたら、下記の twitter アカウントまでお知らせください。

最後に、本書は Debian/GNU Linux、TeXLive2018、git、GNU Make、vim、atom といった、オープンソースソフトウェアを使って作成されました。前回からこれまで使っていた、pLaTeX ではなく、LuaLaTeX で組版してみました。また日本語のフォントは IPAex フォントを PDF に埋め込んでいます。このような有益なソフトウェアを開発、維持、管理していただいているすべての皆様に感謝します。また、このページまでたどり着いてくれた読者の方（おそらくあなただけです）に感謝します。ありがとうございました。

2019 年 4 月 13 日　omoikane (@jm6xxu) 拝

著者：omoikane (@jm6xxu)
発行：2019 年 4 月 13 日