情報学群実験第 2 レポート

# ARM アセンブリ言語の即値代入制限に関する実験

1250373 溝口洸熙[*1]

2022 年 12 月 7 日

[*1] 高知工科大学 情報学群 2 年 清水研究室

# 即値代入の制限における規則性

## 1　実験の目的

arm Assembler の mov 命令[*1]は，i386 Assembler で扱った mov 命令と対応する．しかし，arm Assembler の mov 命令において，レジスタに代入できる即値には制限がある．本実験の目的は，その制限について実験に基づき示すことである．

## 2　実験の方法

コンピュータ（src 1）を利用して，mov.s（src 3）ファイルをアセンブルする．

src 1　利用コンピュータ及び実行環境

```
$ uname -a
Linux KUT20VLIN-322 5.4.0-70-generic #78~18.04.1-Ubuntu SMP Sat Mar 20 14:10:07 UTC
2021 x86_64 x86_64 x86_64 GNU/Linux
$ arm-none-eabi-as --version
GNU assembler (2,27-9ubuntu1+9) 2.27
$ bash --version
GNU bash, バージョン 4.4.20(1)-release (x86_64-pc-linux-gnu)
```

src 2　アセンブル

```
1 $ arm-none-eabi-as mov.s -o mov.o
2 $ arm-none-eabi-ld mov.o -o mov
3 $ ./mov ; echo $?
4 3
```

src 2: 4 行目は，src 3: 5 行目のテスト値を出力する．

src 3　mov.s

```
1     .section    .text
2     .global     _start
3 _start:
4     mov r7, #1
5     mov r0, #3 @ test number
6     swi #0
```

アセンブル（src 2）の際に src 4 の出力が得られた場合は，src 3: 5 行目に問題があり即値の代入に失敗している．

src 4　Error 出力

```
mov.s: Assembler messages:
mov.s:5: Error: invalid constant (4d1)
   after fixup
```

様々なテスト値を簡単に試すためにスクリプトファイルによる自動実行プログラム（src 7）を作成し，実行した[*2]（src 6）．テストする即値は，$1 \leq N \leq 266240$ である．実行結果には，アセンブルできた即値とその即値の

[*1] レジスタに即値，またはレジスタの値を代入する命令．
[*2] 実行するに際して，空ファイル test.s を作成し，mov.s の一部を変更する必要がある．

前にアセンブルできた即値との差（前項との差）が出力される．

手入力での実験に関しては，実験したい数値 $n$ を src 3: 5 行目の `mov r0, #n` と入力し，アセンブルする．

src 5　mov.s 変更後

```
1      .equ    N,  %d
2      .section    .text
3      .global     _start
4  _start:
5      mov r7, #1
6      mov r0, #N @ test number
7      swi #0
```

src 6　実行

```
$ bash test.sh
```

src 7　test.sh

```
1  #!/bin/bash
2  W=0
3  for i in 'seq 0 266240'
4  do
5  sed -e "s/%d/$i/g" mov.s > test.s
6  arm-none-eabi-as test.s -o test.o > /dev/null 2>&1
7  if [ $? -eq 0]; then
8      echo "$i, OK, $(( $i - $W ))"
9      V=$i
10     W-$i
11 fi
12 done
```

## 3　実験結果

実験結果を Tbl: 1，Tbl: 2 に示す．$N(n)$ は 32 ビットの $n$ をビット反転させた数値である．

Tbl: 1　実験結果（シェルスクリプト）

| 入力数値 | | アセンブルの可否 | 前項との差 |
|---|---|---|---|
| 0 | | OK | - |
| 1 | | OK | 1 |
| 2 | | OK | 1 |
| ⋮ | | ⋮ | ⋮ |
| 256 | $(2^8)$ | OK | 1 |
| 260 | | OK | 4 |
| ⋮ | | ⋮ | ⋮ |
| 1024 | $(2^{10})$ | OK | 4 |
| 1040 | | OK | 16 |
| ⋮ | | ⋮ | ⋮ |
| 4096 | $(2^{12})$ | OK | 16 |
| 4160 | | OK | 64 |
| ⋮ | | ⋮ | ⋮ |
| 16384 | $(2^{14})$ | OK | 64 |
| 16640 | | OK | 256 |
| ⋮ | | ⋮ | ⋮ |
| 65536 | $(2^{16})$ | OK | 256 |
| 66560 | | OK | 1024 |
| ⋮ | | ⋮ | ⋮ |
| 262144 | $(2^{18})$ | OK | 1024 |
| 266240 | | OK | 4096 |

Tbl: 2　実験結果（手入力）

| 入力数値 | アセンブルの可否 |
|---|---|
| 0x7fffff | MISS |
| 0xffffffe | MISS |
| 0xffffff | OK |
| $N(2^8)$ | OK |
| $N(2^{10})$ | OK |
| $N(2^{14})$ | OK |
| $N(0)$ | OK |

Tbl: 1 を入力数値 $n \in \{0\} \cup \mathbb{Z}_+$ に対して，アセンブルの可能条件を一般化する．

**規則 1**

$$2^p < n \leq 2^{p+2} \qquad (p = 2k + 6)$$

となる $k(1 \leq k \leq 5)$ に対して，

$$n \bmod 4^k = 0$$

であることが，$n$ をアセンブルできる条件である．

# 4 考察

**■結果からの考察** Tbl: 1 より，mov 命令による 18 ビットまでの即値の代入に関しては**規則 1** に従っていることが判る．さらに，入力数値のビット列が 8 ビットに収まり，その 8 ビットに収まったビット列の先頭と，レジスタの 1 ビット目との距離が 2 の倍数である必要がある（Fig: 1）．

**■条件の推測** Fig: 1，Fig: 2 より，入力数値に対して右に $2n(n < 16)$ 回ローテートし，上位 24 ビットが全て同じ数値であれば mov 命令での即値代入が可能であると推測した．Tbl: 2 より，入力数値 $i$ が 0xFFFFFF より大きい場合，$N(i)$ が**規則 1** に準拠するならばアセンブル可能であることが予想される（Fig: 2）．

従って，入力数値 $n \in \{0\} \cup \mathbb{Z}_+$ に対して，アセンブルの可能条件は以下のようになることが予想される．

**規則**

$$n' = \begin{cases} n & (0 \leq n < 2^{24}) \\ N(n) & (2^{24} \leq n < 2^{31} - 1) \end{cases} \tag{1}$$

---

$$2^p < n' \leq 2^{p+2} \qquad (p = 2k + 6)$$

となる $k(1 \leq k \leq 5)$ に対して，

$$n' \bmod 4^k = 0$$

であることが，$n$ をアセンブルできる条件である．

Fig: 1 アセンブル可能な即値の例 1

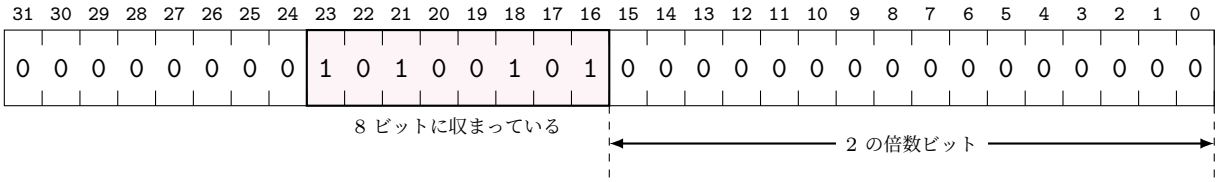



Fig: 2 アセンブル可能な即値の例 2

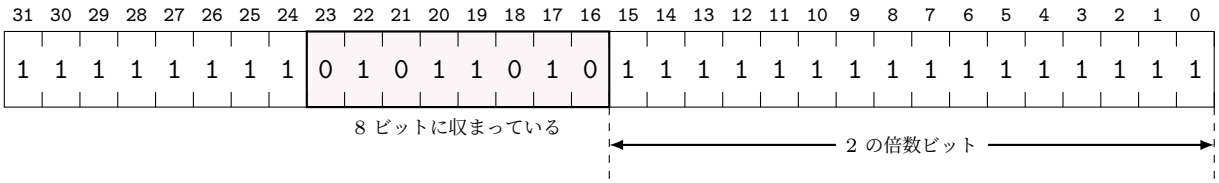
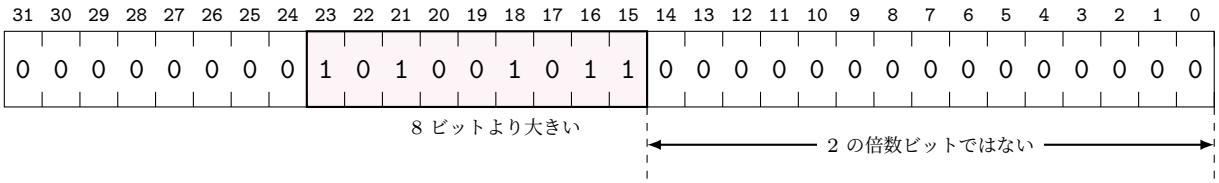



Fig: 3 アセンブルが不可能な即値の例



**■arm Assembler の mov 命令の詳細**　資料 [1] によると, ARM データの処理命令ビットレイアウトで，即値に当てられているのは下位 12 ビットのみである．

さらに，12 ビットの即値を 12 ビットの番号として処理するのではなく，4 ビットの回転値と 8 ビットの値として処理する（Fig: 4）．この回転値 $r$ は即値 $I$ を右へ $2r$ ローテートするために用いる．

Fig: 4　即値の利用

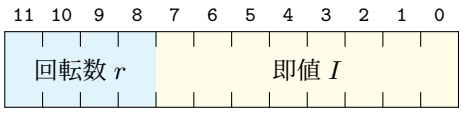
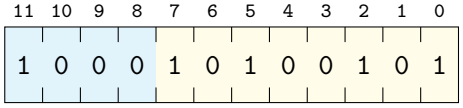
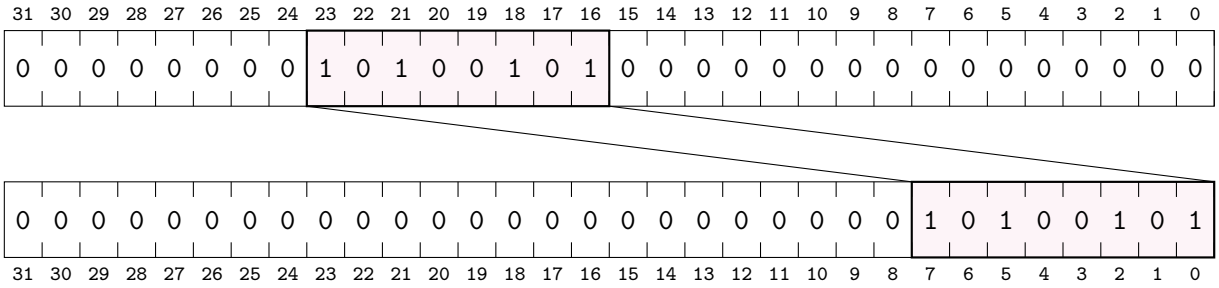



Fig: 5　結果




仮に $i =$ 0x00D30000 が入力されたとする．$i$ のビット列と右ローテートを Fig: 6 に示す．$i$ はそのままだと即値として利用できないため，即値として利用できる下位 8 ビット $I$ にローテートさせる．（シフトではない.）このとき 16 ビットシフトさせたのでこの値を 2 で割った 8 を回転数 $r$ に格納し，下位 12 ビットのみで 23 ビット必要である数値 $i =$ 0x00D30000 を Fig: 5 のように表すことができる．

Fig: 6　$i$ のビット列，右ローテート




**■このような設計になっている理由**　ARM データ処理命令のビットレイアウト [1] によると，1 つのレジスタ内に Cond や，加減算，移動，比較などを正確に定義するための値も格納されている．32 ビット全てを即値に当てることができず，より少ないビット数でより多くの数値を表現するために，このような設計になったのだろう．この設計においての不都合は，幾つかの即値を mov 命令において代入できないことだが，$2^n (n \in \mathbb{N}, 0 \leq n \leq 31)$ や，32 ビット・ワード内の任意の 4 バイトの位置にあるバイト値の代入を保証しており，最も一般的なケースをカバーしているので使用上の問題は少ないと言える．[2, p.52]

# 謝辞

本実験課題は本学情報学群 1250372 三上 柊氏と共同で実施した．また本学情報学群 1250383 山本 昇永氏にはシェルスクリプトファイル（src 7）の作成にご協力いただいた．これらの方々に深く感謝いたします．

溝口 洸熙

# 参考文献

[1] alisdair mcdiarmid. *arm immediate value encoding*. https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/.

[2] S. Furber, アーム. 改訂 ARM プロセッサ: 32 ビット RISC のシステム・アーキテクチャ. Design wave books. CQ 出版, 2001.