

計算機システム

中間課題

J2200071 齊藤 隆斗

1-a

\$s0 に与えられる 32 ビットの値を 8 ビットずつ 4 つに分割し、それぞれの和を計算し、答えを \$s1 に出力するプログラム。ただし、オーバーフロー（和が 8 ビットに収まらない）は気にしなくて良いものとする。

ここで、この問題を正常に処理して停止するようなプログラムを 2 つ作成した。1 つ目のプログラムは以下のようなものである。

```
.text
main:
    #init
    li $s0, 0x06030104
    li $s1, 0

    addi $t0, $s0, 0 # laod $s0
    li $t1, 0

    # first 8 bits
    andi $t2, $t0, 0xFF
    add $t1, $t1, $t2

    # next 8 bits
    srl $t0, $t0, 8 # shift 8 bits
    andi $t2, $t0, 0xFF
    add $t1, $t1, $t2

    # next 8 bits
    srl $t0, $t0, 8 # shift 8 bits
    andi $t2, $t0, 0xFF
    add $t1, $t1, $t2

    # last 8 bits
    srl $t0, $t0, 8 # shift 8 bits
    andi $t2, $t0, 0xFF
    add $t1, $t1, $t2

    # result into $s1
    addi $s1, $t1, 0

    jr $ra
```

2 つ目のプログラムは以下のようなものである。

```

.text
main:
    #init
    li $s0, 0x06030104
    li $s1, 0

    addi $t0, $s0, 0 # laod $s0
    li $t1, 0
    li $t3, 4 # counter

Loop:
    andi $t2, $t0, 0xFF
    add $t1, $t1, $t2
    srl $t0, $t0, 8 # shift 8 bits
    addi $t3, $t3, -1 # decrement counter
    bne $t3, 0, Loop

    # result into $s1
    addi $s1, $t1, 0

    jr $ra

```

1つ目のプログラムは、LSBの方から8bitを取り出し、それを、ある領域に格納する。その後、LSBの方の8bitは必要なくなるので8bit右シフトを行う。その後同じように8bitを取り出し、先程の領域へ取り出したデータを加算する。これを繰り返して行い、累積和を求めることで期待していた値を得ることができる。

2つ目のプログラムは、1つ目に用いたアルゴリズムをループを用いて実装している。

この2つのプログラムの比較を行う。

この2つのプログラムの違いとしてまずは、プログラムの柔軟性が挙げられる。1つ目のプログラムはそれぞれのバイトについて別々に処理を記述しているが、2つ目のプログラムは共通している処理について繰り返しを行う。2つ目のプログラムは繰り返しの回数を指定することで容易にプログラムの変更が行える。また、ループを用いることで全体としてのプログラムの記述量は2つ目の方が少なくなっている。

次に、これらのプログラムの違いとして、繰り返し時の条件判断の有無が挙げられる。2つ目のプログラムには条件判断が4回あるが、1つ目のプログラムには条件判断は存在しない。これを考慮すると単純な命令数のみを見て、1つ目のプログラムの方が効率が良いと言える。

1-b

\$s0 に正の数値 a を与えると、a 以下の素数を 0x1000A000 番地から順番に格納し、\$s1 にその個数を出力する プログラム。

作成したプログラムは以下のようになった。

```

.text
main:
    # init

```

```

    li $t0, 0x1000A000 # addr to table
    li $s0, 1000

    li $t1, 0 # num of prime-number
    li $t2, 2 # set $t2 to 2(counter)

Loop:
    li $t3, 2

    jal Is_Prime # function

Prime:
    sw $t2, 0($t0)
    addi $t1, $t1, 1
    addi $t0, $t0, 4

Not_Prime:
    beq $s0, $t2, Exit
    addi $t2, $t2, 1 # increment
    j Loop

Exit:
    addi $s1, $t1, 0 # $s1 <-- $t1
    li $v0, 10
    syscall

# is $t2 prime?
Is_Prime:
    beq $t2, $t3, Prime # 2 is prime number
Loop_1:
    div $t2, $t3
    mfhi $t4
    beq $t4, $zero, Not_Prime

    addi $t3, $t3, 1
    bne $t3, $t2, Loop_1
    j Prime

```

2

簡単なCプログラムを書き, コンパイル(gcc -Sfoo.c)して生成されるアセンブラプログラム(foo.s)を解説してみよ. また, 最適化オプション(-O1, -O2, -O3 など)有りと無しで比較してみよ.

作成したCプログラムは次のようなものである.

```
#include <stdio.h>

// 2^4 = 16
int main(void) {
    int i = 1;
    for (int j=0; j<4; j++) {
        i = 2 * i;
    }
    printf("%d\n", i);
    return 0;
}
```

このプログラムは $2^4 = 16$ を計算するプログラムである.

これを -S pow.c によってコンパイルしたものは次のようになった.

```
.file "pow.c"
.text
.section .rodata
.LC0:
.string "%d\n"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl $1, -4(%rbp)
movl $0, -8(%rbp)
jmp .L2
.L3:
sall -4(%rbp)
addl $1, -8(%rbp)
.L2:
cmpl $3, -8(%rbp)
jle .L3
movl -4(%rbp), %eax
movl %eax, %esi
movl $.LC0, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
```

```

.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 14.1.1 20240607 (Red Hat 14.1.1-5)"
.section .note.GNU-stack,"",@progbits

```

以下ではこのコードについて解説していく.

1. 変数の初期化:

```

movl    $1, -4(%rbp)
movl    $0, -8(%rbp)

```

ここでは2つの変数を初期化している. `-4(%rbp)` に1を、`-8(%rbp)` に0を設定している.これらは累積値とカウンタとして使用される.

2. ループ:

```

        jmp .L2
.L3:
    sall    -4(%rbp)
    addl    $1, -8(%rbp)
.L2:
    cmpl    $3, -8(%rbp)
    jle .L3

```

このループはカウンタが3以下である限り繰り返される. 各反復で `sall` による左シフトによって累積値 `-4(%rbp)` を2倍にし、`addl` によってカウンタ `-8(%rbp)` を1増やす.

3. 結果の出力:

```

movl    -4(%rbp), %eax
movl    %eax, %esi
movl    $.LC0, %edi
movl    $0, %eax
call    printf

```

計算結果を `%esi` レジスタに、フォーマット文字列のアドレスを `%edi` レジスタにロードし、`printf` を呼び出して結果を出力する.

このように、このプログラムは計算 2^4 を行い、その結果16を出力する.

同一のCプログラムを `-O3` オプションをつけてコンパイルしてもものは以下ようになった.

```

.file "pow.c"
.text
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "%d\n"
.section .text.startup,"ax",@progbits
.p2align 4
.globl main
.type main, @function
main:

```

```

.LFB11:
.cfi_startproc
subq $8, %rsp
.cfi_def_cfa_offset 16
movl $16, %esi
movl $.LC0, %edi
xorl %eax, %eax
call printf
xorl %eax, %eax
addq $8, %rsp
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE11:
.size main, .-main
.ident "GCC: (GNU) 14.1.1 20240607 (Red Hat 14.1.1-5)"
.section .note.GNU-stack,"",@progbits

```

このプログラムは以下を見ると単に 16 を出力するプログラムであることがわかる.

```

movl $16, %esi
movl $.LC0, %edi
xorl %eax, %eax
call printf

```

このように、-O3 オプションをつけると本来のプログラムから大幅に最適化され、繰り返しの構造までなくなっていることがみてとれた.