記号処理・第 8 回

2017 年 5 月 30 日

今回の内容

8.1 仮想計算機 MVM . . . . . 8−1
8.2 演習問題 . . . . . 8−8

8.1 仮想計算機 MVM

この節では、仮想的な計算機 MVM を導入し、次回以降、この MVM の機械語を Minimum C コンパイラのオブジェクト言語としてそのコード生成部の構築をしていくことにします。

MVM の 概要

仮想計算機 MVM の概要は次のようなものです。

1. 20bit (1MB) のアドレス空間を持つ。
2. 機械語命令はすべて 32bit 長である。
3. プログラムの実行は 0 番地から開始される。
4. 16 個の 32bit 長レジスタ (`R0`, `R1`, . . ., `R15`) を持つ。この内、`R0` は、値が常に 0 であるような特殊なレジスタであり、`R14` はスタックポインタとして、`R15` はプログラムカウンタとして用いられる。
5. レジスタ間で 32bit 長の整数演算を行うことができ、その結果によって 4 つのフラグ `Z` (演算結果が 0)、`N` (演算結果が負)、`C` (最上位ビットからの繰り上がりや繰り下がり発生)、`V` (オーバーフロー発生) が設定される。
6. 32bit 長のデータは、その上位バイトがメモリの下位アドレスに、下位バイトが上位アドレスに格納される (Big Endian)。

メモ

MVM の 動作

MVM は次のような C プログラムに相当する動作を行います。ただし、ここに用いられている `unsigned char` 型は 8bit 長であるものとし、`int` 型や `unsigned int` 型は 32bit 長であるものとします。

```
#define MEMSIZE (1<<20)          /* 1MB */

typedef unsigned int Boolean;    /* 真理値  */
typedef unsigned char Byte;      /* 8bit長の符号なし整数 */
typedef unsigned int Word;       /* 32bit長の符号なし整数 */
```

```
Word mem[MEMSIZE/4];
Word R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15;
Boolean Z, N, C, V;

MVM ()
{
    Word inst;

    Z = N = C = V = 0;      /* 各フラグの初期値は 0 */
    R14 = MEMSIZE;          /* スタックポインタの初期値はメモリ空間の末尾 */
    R15 = 0;                /* プログラムカウンタの初期値は 0 番地 */
    while (1) {
        inst = mem[R15/4];  /* 機械語命令を読み込む */
        R0 = 0;             /* R0 の値は常に 0 */
        R15 += 4;           /* プログラムカウンタを進める */
        inst の実行
    }
}
```

メモ

MVM の命令群

MVM の機械語命令には以下のような 3 種類の形式があります。ただし、* 印の付された機械語命令は、これから開発する Minimum C コンパイラでは実際には利用しません。

形式 1

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
op	*i*	*j*	*d*

op	命令	動作
0000	LD Ri, Rj, d	Ri = mem[(Rj + Ext(d))/4];
0001	ST Ri, Rj, d	mem[(Rj + Ext(d))/4] = Ri;
0010*	LDB Ri, Rj, d	Ri = ext(((Byte *)mem)[Rj + Ext(d)]);
0011*	LDUB Ri, Rj, d	Ri = ((Byte *)mem)[Rj + Ext(d)];
0100*	STB Ri, Rj, d	((Byte *)mem)[Rj + Ext(d)] = Ri;

ただし、関数 ext(operand) および Ext(operand) は、それぞれ 8bit と 20bit の符号付き整数 operand を 32bit に符号拡張するもので、以下のように定義される関数です。

```
#define BIT_ON(x, b)    (((x) & (1 << (b))) != 0)

/* 8bitから32bitへの符号拡張を行う */
```

```
Word ext(Word operand) {
    if (BIT_ON(operand, 7))
        return operand | ~((1 << 8) - 1);
    return operand;
}

/* 20bitから32bitへの符号拡張を行う */
Word Ext(Word operand) {
    if (BIT_ON(operand, 19))
        return operand | ~((1 << 20) - 1);
    return operand;
}
```

LD 命令は「ロード (load) 命令」と呼ばれ、メモリ中の (指定したアドレスの) 32bit 長のデータを (指定した) レジスタにコピーする働きをします。また、ST 命令は「ストア (store) 命令」と呼ばれ、(指定した) レジスタに記憶されている 32bit 長のデータをメモリ中 (指定したアドレスに) コピーします。LDB、LDUB、STB も同様ですが、32bit 長ではなく、8bit 長のデータを扱います。LDB と LDUB の違いは、メモリ中の 8bit 長のデータを 32bit 長のレジスタに格納する際に符号拡張を行うかどうかの違いです。

メモ

形式 2

31 30 29 28 27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
op	*i*	*d*

op	命令	動作
10000000	LDI Ri, d	Ri = Ext(d);
10000001	LDHI Ri, d	Ri = (d << 16);
10000010	POP Ri	Ri = mem[R14 / 4]; R14 += 4;
10000011	PUSH Ri	R14 -= 4; mem[R14 / 4] = Ri;
10000100	CALL Ri, d	R14 -= 4; mem[R14 / 4] = R15; R15 = Ri + Ext(d);
10000101*	JPL Ri, d	R12 = R15; R15 = Ri + Ext(d);
10001000	IN Ri	printf ("? "); scanf ("%d", &Ri);
10001001	OUT Ri	printf("%d\n", Ri);
10001010*	INQ Ri	scanf ("%d", &Ri);
10001011*	OUTQ Ri	printf("%d", Ri);
10001100*	OUTS Ri	printf("%s", Ri + Ext(d));
10001111	EXIT Ri	exit(Ri);
10010000	JP Ri, d	R15 = Ri + Ext(d);
10010010	JPE Ri, d	if (Z == 1) R15 = Ri + Ext(d);

op	命令	動作
10010011	JPNE Ri, d	`if (Z == 0) R15 = R`i `+ Ext(`d`);`
10010100	JPGT Ri, d	`if (N == V && Z == 0) R15 = R`i `+ Ext(`d`);`
10010101	JPLE Ri, d	`if (N != V || Z == 1) R15 = R`i `+ Ext(`d`);`
10010110	JPLT Ri, d	`if (N != V) R15 = R`i `+ Ext(`d`);`
10010111	JPGE Ri, d	`if (N == V) R15 = R`i `+ Ext(`d`);`
10011000*	JPGTU Ri, d	`if (C == 0 && Z == 0) R15 = R`i `+ Ext(`d`);`
10011001*	JPLEU Ri, d	`if (C == 1 || Z == 1) R15 = R`i `+ Ext(`d`);`
10011010*	JPC Ri, d	`if (C == 1) R15 = R`i `+ Ext(`d`);`
10011011*	JPNC Ri, d	`if (C == 0) R15 = R`i `+ Ext(`d`);`
10011100*	JPNEG Ri, d	`if (N == 1) R15 = R`i `+ Ext(`d`);`
10011101*	JPNNEG Ri, d	`if (N == 0) R15 = R`i `+ Ext(`d`);`
10011110*	JPV Ri, d	`if (V == 1) R15 = R`i `+ Ext(`d`);`
10011111*	JPNV Ri, d	`if (V == 0) R15 = R`i `+ Ext(`d`);`

LDI は 20bit 長までの定数データを (指定した) レジスタに格納する命令です。20bit 長では表現できないくらい絶対値の大きいデータは、LDHI 命令と後述の ADDI 命令や ORI 命令を組み合わせることで、レジスタに格納することができます。

メモ

POP 命令と PUSH 命令は、(指定された) レジスタの値を、メモリ中のスタックに push あるいは、スタックから pop します。この時、R14 がスタックポインタとして用いられます。CALL 命令は、次に実行する予定だった機械語命令の置かれているアドレス (プログラムカウンタ R15 の値) を、このスタックに push して、指定されたアドレスに jump します。MVM は、このアドレスから機械語プログラムの実行を続けます。CALL 命令によってスタックに退避された R15 の元の値は、「POP R15」を実行することで復帰させることができますので、MVM に、元の場所に戻って機械語プログラムの実行を続けさせることができます。つまり、CALL 命令と POP R15 命令を使って、サブルーチンの呼び出しと復帰を実現することができるわけです。

メモ

IN 命令や OUT 命令は、MVM に、C 言語の scanf や printf に相当する (整数値の) 入出力処理を行わせます。このような処理は、現実には、入出力装置を制御するための複雑な機械語プログラムを実行することによってでしか実現できませんが、MVM で

は単純化のために CPU 自身にこのような機能が備わっているものとしています。

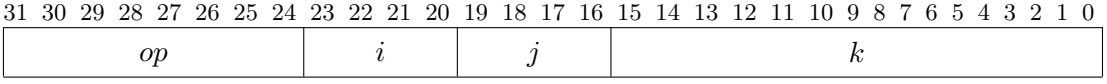
メモ

JP 命令以降の命令群は、分岐命令と呼ばれ、MVM による機械語プログラムの実行の流れを制御するために用いられます。JP 命令は、R15 (プログラムカウンタ) の値を (指定された) 値に (無条件に) 変更し、MVM にそこから機械語プログラムを続行させます。他の分岐命令は、MVM 中のフラグの値を調べ、特定の条件が成り立っている場合にのみ分岐させます。たとえば、2 つの整数 a と b が、レジスタ Ri と Rj にそれぞれ格納されているとき、後述する減算命令 SUB を使って、「SUB R0, Ri, Rj」という機械語命令を MVM に実行させると、$a-b$ の計算結果に応じて、Z、N、C、V などのフラグが設定されます。この状態で、条件付き分岐命令 JPE、JPNE、JPGT、JPLE、JPLT、JPGE を実行すれば、次の表のように条件付きで分岐を行うことができます。

命令	動作	略語 (ニーモニック) の意味
JPE	$a=b$ の時だけ分岐	jump if equal to
JPNE	$a\neq b$ の時だけ分岐	jump if not equal to
JPGT	$a>b$ の時だけ分岐	jump if greater than
JPLE	$a\leqq b$ の時だけ分岐	jump if less than or equal to
JPLT	$a<b$ の時だけ分岐	jump if less than
JPGE	$a\geqq b$ の時だけ分岐	jump if greater than or equal to

メモ

形式 3



op	命令	動作
10100000	ADD Ri, Rj, Rk	Ri = Add(Rj, Rk, 0);
10100001*	ADDC Ri, Rj, Rk	Ri = Add(Rj, Rk, C);
10100010	SUB Ri, Rj, Rk	Ri = Sub(Rj, Rk, 0);
10100011*	SUBC Ri, Rj, Rk	Ri = Sub(Rj, Rk, C);
10100100*	AND Ri, Rj, Rk	Ri = Logical(Rj & Rk);
10100101*	OR Ri, Rj, Rk	Ri = Logical(Rj | Rk);
10100110*	XOR Ri, Rj, Rk	Ri = Logical(Rj ^ Rk);

op	命令	動作
10100111*	NXOR Ri, Rj, Rk	Ri = Logical(~(Rj ^ Rk));
10101000*	SLL Ri, Rj, Rk	Ri = (Rj << Rk);
10101001*	SRL Ri, Rj, Rk	Ri = (Rj >> Rk);
10101010*	SRA Ri, Rj, Rk	Ri = SRA(Rj, Rk);
10110000	ADDI Ri, Rj, k	Ri = Add(Rj, k, 0);
10110001*	ADDIC Ri, Rj, k	Ri = Add(Rj, k, C);
10110010	SUBI Ri, Rj, k	Ri = Sub(Rj, k, 0);
10110011*	SUBIC Ri, Rj, k	Ri = Sub(Rj, k, C);
10110100*	ANDI Ri, Rj, k	Ri = Logical(Rj & k);
10110101*	ORI Ri, Rj, k	Ri = Logical(Rj | k);
10110110*	XORI Ri, Rj, k	Ri = Logical(Rj ^ k);
10110111*	NXORI Ri, Rj, k	Ri = Logical(~(Rj ^ k));
10111000*	SLLI Ri, Rj, k	Ri = (Rj << k);
10111001*	SRLI Ri, Rj, k	Ri = (Rj >> k);
10111010*	SRAI Ri, Rj, k	Ri = SRA(Rj, k);
11000000	MUL Ri, Rj, Rk	Ri = (int)Rj * (int)Rk;
11000001	DIV Ri, Rj, Rk	Ri = (int)Rj / (int)Rk;
11000010	MOD Ri, Rj, Rk	Ri = (int)Rj % (int)Rk;
11000100*	MULU Ri, Rj, Rk	Ri = Rj * Rk;
11000101*	DIVU Ri, Rj, Rk	Ri = Rj / Rk;
11000110*	MODU Ri, Rj, Rk	Ri = Rj % Rk;

形式3の命令群は、MVMに、四則演算やビット毎の論理演算を行わせるものです。ただし、関数 Add()、Sub()、Logical()、SRA() は以下のように定義されます。

```
#define MSB(x)      (((x) & (1 << 31)) != 0)

/* 加算 */
Word Add(Word operand1, Word operand2, Word carry) {
    Word result = operand1 + operand2 + carry;

    Z = (result == 0);
    N = MSB(result);
    C = ((MSB(operand1) && MSB(operand2))
            || ((MSB(operand1) || MSB(operand2)) && !MSB(result)));
    V = ((MSB(operand1) && MSB(operand2) && !MSB(result))
            || (!MSB(operand1) && !MSB(operand2) && MSB(result)));
    return result;
}

/* 減算 */
Word Sub(Word operand1, Word operand2, Word carry) {
    Word result = operand1 - operand2 - carry;
```

```
    Z = (result == 0);
    N = MSB(result);
    C = ((!MSB(operand1) && MSB(operand2))
            || ((!MSB(operand1) || MSB(operand2)) && MSB(result)));
    V = ((MSB(operand1) && !MSB(operand2) && !MSB(result))
            || (!MSB(operand1) && MSB(operand2) && MSB(result)));
    return result;
}

/* 論理演算結果のフラグへの反映 */
Word Logical(Word result) {
    Z = (result == 0);
    N = MSB(result);
    C = V = 0;
    return result;
}

/* 算術的右シフト */
Word SRA(Word operand1, Word operand2) {
    Word result = (operand1 >> operand2);

    if (MSB(operand1)) {
        if (operand2 < 32)
            result = (result | (~0 << (32 - operand2)));
        else
            result = ~0;
    }
    return result;
}
```

ここで、4つのフラグ Z、N、C、V は、次のように設定されることに注意してください。

- Z フラグ　計算結果が0のときに1、そうでなければ0となる。
- N フラグ　計算結果が負(最上位ビットが1)のときに1、そうでなければ0となる。
- C フラグ　最上位ビットからの繰り上がりや繰り下がりがあれば1、そうでなければ0となる。
- V フラグ　オーバーフローが起これば1、そうでなければ0となる。

メモ

MVM の機械語プログラムの例

つぎは、Minimum C で書かれたプログラムと、それと同じ処理を行う MVM の機械語プログラムの 1 例です。機械語プログラムはメモリ中の 0 番地から 59 番地までに並べられており、Minimum C の大域変数であった x と y の値は、それぞれ 60 番地と 64 番地から 4 バイトの領域に記憶されています。

Minimum C

```
int x;
int y;

main ()
{
    input x;

    input y;

    if (x < y) {



        x = y + 1;



    }
    print x;

}
```

MVM

```
番地
60
64


番地              機械語(2進数)                命令
 0  10001000 00010000 00000000 00000000  IN   R1
 4  00010001 00000000 00000000 00111100  ST   R1, R0, 60
 8  10001000 00010000 00000000 00000000  IN   R1
12  00010001 00000000 00000000 01000000  ST   R1, R0, 64
16  00000001 00000000 00000000 00111100  LD   R1, R0, 60
20  00000010 00000000 00000000 01000000  LD   R2, R0, 64
24  10100010 00000001 00000000 00000010  SUB  R0, R1, R2
28  10010111 11110000 00000000 00010000  JPGE R15, 16
32  00000001 00000000 00000000 01000000  LD   R1, R0, 64
36  10000000 00100000 00000000 00000001  LDI  R2, 1
40  10100000 00010001 00000000 00000010  ADD  R1, R1, R2
44  00010001 00000000 00000000 00111100  ST   R1, R0, 60
48  00000001 00000000 00000000 00111100  LD   R1, R0, 60
52  10001001 00010000 00000000 00000000  OUT  R1
56  10001111 00000000 00000000 00000000  EXIT R0
```

8.2 演習問題

次の Minimum C プログラムに対応する MVM の機械語プログラムを作りなさい。

1.

```
int x;

main ()
{
    input x;
    if (x < 0)
        x = -x;
    print x;
}
```

2.

```
int i, n, sum;

main ()
{
    input n;
    sum = 0;
    i = 1;
    while (i <= n) {
        sum = sum + i*i;
        i = i + 1;
    }
    print sum;
}
```

記号処理・第 8 回・終わり