マイクロプロセッサ演習

2004 年度

第 4 回

1 今回の演習に必要な予備知識

配列 (前回の補足) 前回のプログラム LoadStore3.asm は以下の流れを実行するプログラムであった。

1. メモリに、データ 4 つ分 (4 バイト $\times 4 = 16$ バイト) の領域を確保。
2. 先頭の 3 つのデータ領域 (array[0] 〜 array[2] と呼ぶことにする) に整数 1、2、3 を格納。
3. array[0] 〜 array[2] の値をレジスタ $s0, $s1, $s2 にロード。
4. $s0 $s1, $s2 の和を $s3 に格納。
5. $s3 の値をメモリ上の array[3] にストア。

この流れの概念図を示したのが図 1 である。この流れの中で 3. 〜 5.、すなわち「メモリから値をレジスタにロードし、計算を実行した後、結果をメモリ上にストアする」という流れは重要であり、今後の演習でも頻出するので、しっかり身につけて欲しい。

なお、ここではメモリ上の 4 つのデータ領域を array[0] 〜 array[3] と呼んだが、これは C 言語の**配列**の記法に従ったものである。

配列とはデータが表の様に複数個並んだものである。例えば、50 人のクラスのテストの点数は配列 array[] に、「array[0] = 80, array[1]= 70, $\cdots$, array[49]=58」などと格納できる。要素が n 個の配列は array[0] 〜 array[$n-1$] で表されることに注意。

また、配列の添字は 1 づつ増えるが、メモリアドレスは 4 づつ増えることにも注意しよう (メモリの 1 ワードが 4 バイトであることによる)。

2 [SPIM] 配列の和

いつものように演習用の Web ページから data04.zip をダウンロードしよう。

n 個の要素を持つ配列 array[n] の和、すなわち

$$S = \sum_{i=0}^{n-1} \text{array}[i] \tag{1}$$

を考えよう。$n = 3$ の時の S を求めるのがプログラム LoadStore3.asm であった。$n = 3$ であれば、MIPS のアセンブリ言語の命令 `add` を 2 回書くだけで良いが、n が大きくなると `add` 命令の記述回数もどんどん増えていってしまい、大変である。

このように、同じような命令を繰り返し実行するプログラムを記述する時のために、ほとんどのプログラム言語では「繰り返し構造」という仕組みを用意している。たとえば、C 言語では (1) 式は以下のように記述できる。

```
i=0;
S=0;
while(i < n){
      S = S + array[i];
      i = i+1;
}
```

ここで、`i` はカウンタのようなもので `i=i+1` に従って 1、2、3、と値が増えていき、`i<n` が成り立っている間は while ループの中身、すなわち `S = S + array[i]` が実行される。`i` が `n` 以上になるとプログラムは while ループを外れるが、そのとき `S` の値は (1) 式で表される配列 array[] の和になっている、という仕組みである。

この仕組みを MIPS のアセンブリ言語で行なうのがプログラム SumArray.asm である。このプログラムは完成済みのものなので、動かして動作を確認してみてほしい。$n = 10$ であり、array[i] $= i+1$

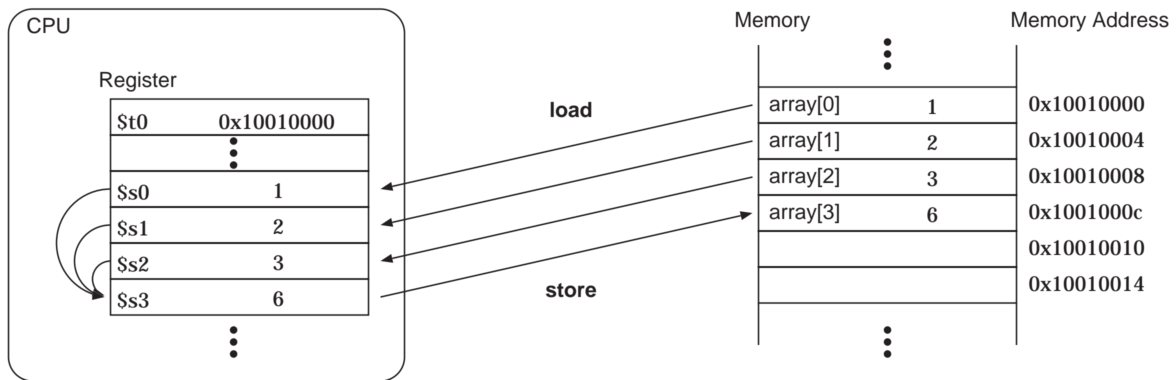

図 1: LoadStore3.asm のプログラムの流れの概念図

$(0 \leq i \leq 9)$ の値がメモリに格納されているときに (1) 式を計算するプログラムである。

1. 計算結果はどこに現れているか？
2. C 言語版でのカウンタ i の役割はどのレジスタが担っているか？
3. レジスタ $t0 はどんな役割をし、プログラムの進行とともに値はどう変化するか？

[**SumArray.asm を理解するためのヒント**]

- C 言語版での while{ } は、SumArray.asm では LOOP: 〜 ENDLOOP: に相当する。“LOOP” と “ENDLOOP” は単なるラベルであるので、好きな名前をつけて構わない。
- C 言語版での while{ } ループが繰り返される条件、すなわち (i < n) は、 SumArray.asm では

```
slt $t4, $t2, $t3
beq $t4, $zero, ENDLOOP
```

  の 2 行と、

```
j LOOP
```

  に相当する。それぞれの命令の意味は付録を参照。なお、$zero は値 0 が格納されている特別なレジスタである。

3 [SPIM] 繰り返し構造を使ったプログラムの記述

問題 2 でループ構造を持ったプログラムの働きをみてもらったが、この問題ではそのようなプログラムを実際に記述してもらう。

先程のプログラムと同様、array$[i] = i + 1$ $(0 \leq i \leq 9)$ なる配列がメモリに格納されている時、その値を画面に表示するプログラムを記述してみよう。

ここで、「画面に表示する」とは、演習第 1 回で取り扱ったサンプルプログラム hello.asm のような振舞いをするプログラムを書く、ということである。(忘れた人は、もう一度動かしてみよう)

配列の要素一つ一つを表示するごとに改行を入れるのは面倒なので、要素が連続して表示させるプログラムで構わない (つまり表示される結果が “12345678910” となるプログラムでよい)。

プログラムの骨組みは PrintArray.asm なので、このファイルに追加するかたちで記述すること。

なお、レジスタ (例えば $t0) の値を画面に表示するのは以下の 3 行の記述で行なうこと。この 3

```
li      $v0, 1
add     $a0, $t0, $zero
syscall
```

行の意味は現在では気にしなくて構わない。

A [補足] これまでのまとめ

A.1 DOS 版 SPIM の命令

DOS 版 SPIM の命令のうち、ここまでの範囲で知っておくと良いものを列挙しておく。

- `load "filename"`
  プログラム "filename" を読み込む。
- `run`
  ロードしたプログラムを実行する。
- `exit` または `quit`
  SPIM を終了する。
- `print OBJ`
  OBJ の内容を表示する。OBJ はレジスタ、メモリアドレス、ラベル等。(例、"print $t0", "print 0x10010000", "print S" 等)

A.2 MIPS のアセンブリ言語の命令

ここまでに取り扱った MIPS のアセンブリ言語の命令をまとめておく。演習で取り扱っていないくても、関連するものは載せてある (引き算 `sub` など)。なお、$\$R_i$ 等はレジスタを、N は整数値を、C は定数値を、L はラベルを表す。

[演算系]

- `add` $\$\mathrm{R}_i$, $\$\mathrm{R}_j$, $\$\mathrm{R}_k$
  $\$\mathrm{R}_i = \$\mathrm{R}_j + \$\mathrm{R}_k$
- `sub` $\$\mathrm{R}_i$, $\$\mathrm{R}_j$, $\$\mathrm{R}_k$
  $\$\mathrm{R}_i = \$\mathrm{R}_j - \$\mathrm{R}_k$
- `addi` $\$\mathrm{R}_i$, $\$\mathrm{R}_j$, C
  $\$\mathrm{R}_i = \$\mathrm{R}_j + \mathrm{C}$

[ロード・ストア系]

- `la` $\$\mathrm{R}_i$, L
  ラベル L のアドレスの値をレジスタ $\$\mathrm{R}_i$ にロードする。
- `li` $\$\mathrm{R}_i$, C
  定数値 C をレジスタ $\$\mathrm{R}_i$ にロードする。
- `lw` $\$\mathrm{R}_i$, N($\$\mathrm{R}_j$)
  $\$\mathrm{R}_j + \mathrm{N}$ のアドレスの内容をレジスタ $\$\mathrm{R}_i$ にロードする。(N は 4 の倍数)
- `sw` $\$\mathrm{R}_i$, N($\$\mathrm{R}_j$)
  レジスタ $\$\mathrm{R}_i$ の内容を $\$\mathrm{R}_j + \mathrm{N}$ のアドレスにストアする。(N は 4 の倍数)

[条件分岐系]

- `beq` $\$\mathrm{R}_i$, $\$\mathrm{R}_j$, L
  $\$\mathrm{R}_i = \$\mathrm{R}_j$ ならばラベル L へ分岐する。そうでなければ、何もせず次の命令へ進む。
- `bne` $\$\mathrm{R}_i$, $\$\mathrm{R}_j$, L
  $\$\mathrm{R}_i \neq \$\mathrm{R}_j$ ならば、ラベル L へ分岐する。そうでなければ、何もせず次の命令へ進む。
- `slt` $\$\mathrm{R}_i$, $\$\mathrm{R}_j$, $\$\mathrm{R}_k$
  $\$\mathrm{R}_j < \$\mathrm{R}_k$ ならばレジスタ $\$\mathrm{R}_i$ に 1 を代入、そうでなければ 0 を代入。
- `j` L
  無条件にラベル L へジャンプ。
- `jr` $\$\mathrm{R}_i$
  レジスタ $\$\mathrm{R}_i$ に格納されているアドレスへジャンプ。

[疑似命令系]

- `.text`: テキスト領域の開始
- `.data`: データ領域の開始
- `.globl` L: ラベル L を大局的に参照可能な記号と宣言する。
- `.word` n: 1 語のデータ n を配置。
- `.space` n: n バイトの領域を確保
- `.asciiz` "Strings": 文字列を配置。(末尾に NULL 文字を追加)