ソフトウェアのハードウェア化を考える

―― ソート回路の設計を例に

水木元由

Cベース言語によるハードウェアの設計が注目を集めている．ハードウェアをC言語で設計できるようになれば，従来マイクロプロセッサを用いて処理していた機能（ソフトウェア）を，専用回路（ハードウェア）化しやすくなるであろう．しかし，ある一つの機能をソフトウェアで処理する場合と，ハードウェアで処理する場合とでは，最適なアルゴリズムが異なることが多い．本稿では，ソフトウェアで処理されることの多いソート（整列）のハードウェア化を考えてみる．（編集部）

はじめに

ハードウェアの設計は，Verilog HDLやVHDLのような言語を使って行われるようになりました．最近では，Cベース言語の設計も注目を集めています．

これらの言語で記述されたハードウェアは，専用のソフトウェアでコンパイルして回路を生成します．このスタイルは，一見しただけでは，ソフトウェア開発と同じです．ハードウェアの設計環境がソフトウェアの設計環境に近づいているかのようにも見えます．

システムLSI時代になり，ハードウェアとソフトウェアのトレードオフが重要視されています．いずれは，仕様さえ与えてやれば，設計者が意識しなくても，ハードウェア化する機能とソフトウェアで処理する機能を適切に振り分けてくれるようなコンパイラが誕生するのかもしれません．その場合，プロセッサ・コアも必要な機能のみで構成されたスケーラブルなモジュールが生成されることでしょう．ハードウェアとソフトウェアの垣根があいまいになるとみることもできます．しかし現状では，ハードウェアとソフトウェアのトレードオフは，まだまだ設計者の頭を悩ます大きな問題の一つといえます．

また，Cベース言語による設計の時代を迎えても，ソフトウェアとして記述したコードを，ハードウェア用にそのまま使えるとはかぎりません．言語仕様の微妙な違いのみならず，ソフトウェアのアルゴリズムがハードウェア化に適するとはかぎらないからです．

そこで本稿では，ある一つの機能をハードウェアで専用回路化することを考えていくことにします．通常ソフトウェアで処理されることの多いアルゴリズムを，ハードウェア記述言語で記述して，実際に動作させてみることにします．

いろいろなソート

ソフトウェアのアルゴリズムを学ぶにあたり，必ず出てくる話題の一つにソート（整列）があります．ソートは数値データを扱う際には不可欠な処理の一つです．通常，ソフトウェア処理されることが多いソートですが，ここでは，ハードウェア処理によるソート回路を設計してみたいと思います．

一口にソートといっても，さまざまなアルゴリズムが存在します．

よくシャッフルされたカードを順に並べ直すのに，みなさんはどのような方法をとるでしょうか．ある人は床にばらまいて，小さい順に選んでいくでしょう（選択整列法）．またある人は，両手でカードを広げ，端から順にテーブルに整列させながら置いていくことでしょう（挿入整列法）．きちんと整列するまでシャッフルを繰り返す方法を採用する人はいないと思います．

ソートの手法として，代表的なものだけでも以下のようなものがあり，それぞれ特徴があります．たとえば，手法によって，カードを整列させるのに必要な場所（メモ

〔リスト1〕C言語によるクイック・ソートのプログラム

```
#include <stdio.h>
#include <stdlib.h>
#define SORT_MEMBERS 16

int quick_sort(int *a, int lower, int upper)
{
    int sample;       /* 大小振り分けサンプル値に使用 */
    int i, u ,l;      /* ループ変数 */
    int tmp;          /* スワップ時の一時データ */
    int step = 0;     /* 入れ替え回数 */

    sample = (a[lower] + a[upper]) / 2;
    l = lower;  u = upper;

    do {/* 大小振り分け．サンプル値より大きい物は左，
                                        小さい物は右に */
        /* 入れ替える数字を探す．少なくとも自分自身で止まる*/
        while (a[l] < sample)
            l++;
        while (sample < a[u])
            u--;

        if (l <= u) {

            /* 配列データのスワップ */
            tmp = a[l];
            a[l] = a[u];
            a[u] = tmp;

            l++;  u--;

            /* ソート中のデータ配列の表示 */
            printf("             %2d step : ", ++step);
            for (i = 0; i < SORT_MEMBERS; i++) {
                printf(" %2d", a[i]);
            }
            printf("\n");
        }
    } while(l < u); /* サンプル値を軸に振り分け完了したら
                                        ループ脱出 */

    /* 再帰処理 振り分けた中で再度同じことを行う */
    if (lower  < u) {
        step += quick_sort(a, lower , u);
    }
    if (l < upper) {
        step += quick_sort(a, l, upper);
    }

    return step;
}


int main()
{
    int i;                      /* ループ変数 */
    int step;                   /* 入れ替え回数 */
    static int a[SORT_MEMBERS]; /* データ配列 */

    /* 入力配列データの乱数による設定 */
    printf("ソート前のデータ配列 : ");
    for (i = 0; i < SORT_MEMBERS; i++) {
        a[i] = rand() % 100 ;
        printf(" %2d", a[i]);
    }
    printf("\n");
    printf("\n");

    /* クイック・ソート関数の呼び出し */
    step = quick_sort(a, 0, SORT_MEMBERS - 1);
    printf("\n");

    /* ソート後のデータ配列の表示 */
    printf("ソート後のデータ配列 : ");
    for (i = 0; i < SORT_MEMBERS; i++) {
        printf(" %2d", a[i]);
    }
    printf("\n");

    /* 入れ替えステップ数の表示 */
    printf("入れ替えステップ数は  %4d です\n", step);
    return 1;
}
```

```
ソート前のデータ配列 : 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26

             1 step : 26 86 77 15 93 35 86 92 49 21 62 27 90 59 63 83
             2 step : 26 27 77 15 93 35 86 92 49 21 62 86 90 59 63 83
             3 step : 26 27 21 15 93 35 86 92 49 77 62 86 90 59 63 83
             4 step : 26 27 21 15 49 35 86 92 93 77 62 86 90 59 63 83
             1 step : 15 27 21 26 49 35 86 92 93 77 62 86 90 59 63 83
             1 step : 15 26 21 27 49 35 86 92 93 77 62 86 90 59 63 83
             1 step : 15 21 26 27 49 35 86 92 93 77 62 86 90 59 63 83
             1 step : 15 21 26 27 49 35 86 92 93 77 62 86 90 59 63 83
             1 step : 15 21 26 27 35 49 86 92 93 77 62 86 90 59 63 83
             1 step : 15 21 26 27 35 49 83 92 93 77 62 86 90 59 63 86
             2 step : 15 21 26 27 35 49 83 63 93 77 62 86 90 59 92 86
             3 step : 15 21 26 27 35 49 83 63 59 77 62 86 90 93 92 86
             1 step : 15 21 26 27 35 49 62 63 59 77 83 86 90 93 92 86
             1 step : 15 21 26 27 35 49 59 63 62 77 83 86 90 93 92 86
             1 step : 15 21 26 27 35 49 59 62 63 77 83 86 90 93 92 86
             1 step : 15 21 26 27 35 49 59 62 63 77 83 86 90 93 92 86
             1 step : 15 21 26 27 35 49 59 62 63 77 83 86 86 93 92 90
             1 step : 15 21 26 27 35 49 59 62 63 77 83 86 86 90 92 93
             1 step : 15 21 26 27 35 49 59 62 63 77 83 86 86 90 92 93
```

〔図1〕クイック・ソートの実行結果

リ空間）や，整列にかかる時間が変わってきます．

(1) 選択整列法

データ列の最小値を選択し，未整列部分の先頭に置くことを順次繰り返すことにより整列を行う．

(2) 挿入整列法

新しい場所を用意し，そこに順に移動する際，整列させる．

(3) シェル・ソート

挿入整列法を行う際に効率のよい方法で行う．

(4) バブル・ソート

端から端まで隣り合う二つを比較交換することを整列するまで繰り返す．

(5) ヒープ・ソート（整列2分木法）

選択整列法を2分木構造を用いることにより効率よく