卒業論文

各種ソート回路の設計
とハード/ソフト最適分割の検討

氏　　名：松田 英亮
学籍番号：2260050070-1
指導教員：山崎 勝弘 教授
提 出 日：2009年2月19日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

本論文では、ハードウェア記述言語Verilogを用いた各種ソートのハードウェア化を行った。また、それらのアルゴリズムを基にしてソフトウェアの開発を行い、ハード/ソフト最適分割化の検討を行った。

各種ソート回路は、その性能や特徴が相反しているクイックソートとバブルソートを採用した。ハードウェア面ではどちらもソフトと比較して高速化を実現することができ、またソフトウェアでの性能や特徴をハード面でも実現することができた。

ハード/ソフト最適分割化においては、ソフトウェアでハードウェアのアルゴリズムを実現しCPU負荷を測定、分割パターンの検討を行った。各分割パターンはそれぞれ目的に応じての重要性を考慮したパターンに分類化することができた。

目次

図目次

表目次

1.　はじめに

近年、特定の機能を実現する目的でコンピュータを組み込んだ組み込みシステムの必要性が高まってきている。組み込みシステムの開発には、ハード/ソフトの両方の知識が強く要求され、それらはテレビ、携帯電話、自動車、カーナビゲーションシステム、信号機などあらゆる機器に搭載されている。とりわけ自動車には多くの組み込みシステムが搭載されていて、高級車には１００個を超えるマイコンが搭載されている。

半導体集積技術の向上に伴い、LSIを使用して構築される組み込みシステムはハードウェア的にもソフトウェア的にも大規模化・複雑化している。このため，組み込みシステムの設計・開発効率や品質を向上させるには、ハードウェア設計とソフトウェア設計の協調設計が重要である。

ハード/ソフト協調設計は、設計にあたって予め目的や優先事項を決めておき、それに見合うようにシステムのハードウェア部分とソフトウェア部分を分けて設計する手法である。例えば、速度最優先で他の項目を無視できるならば全て、もしくはほとんどをハードウェアにするべきである。また技術の移り変わりに影響されやすい部分は柔軟性の高いソフトウェアにするべきである。このようにして設計の負担を軽くしようというのがハード/ソフト協調設計である［3］。

以上のような背景から、本研究ではソート処理をHDLによる回路設計を行い、回路におけるハード/ソフト最適分割の検討を行う。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ある。通常、ソフトウェア処理されることが多いソートにおいて、代表的なソートの１つであり、それぞれの長所・短所が特徴的であるクイックソートとバブルソートのハードウェア回路を設計する。さらに、データ数を64個と128個とした時のそれぞれの回路規模・遅延を測定し、その有用性と実用性がどれほどのものかを検証する。

ハード/ソフト最適分割においては、ソフトウェアでハードウェアのアルゴリズムを実現し、CPU負荷を測定、分割パターンの検討を行う。各分割パターンはそれぞれ目的に応じての重要性を考慮したパターンに分類化することを行う。

2.　ソート処理の概要

2.1　クイックソートのアルゴリズム

クイックソートの一連のアルゴリズムを図１に示す。

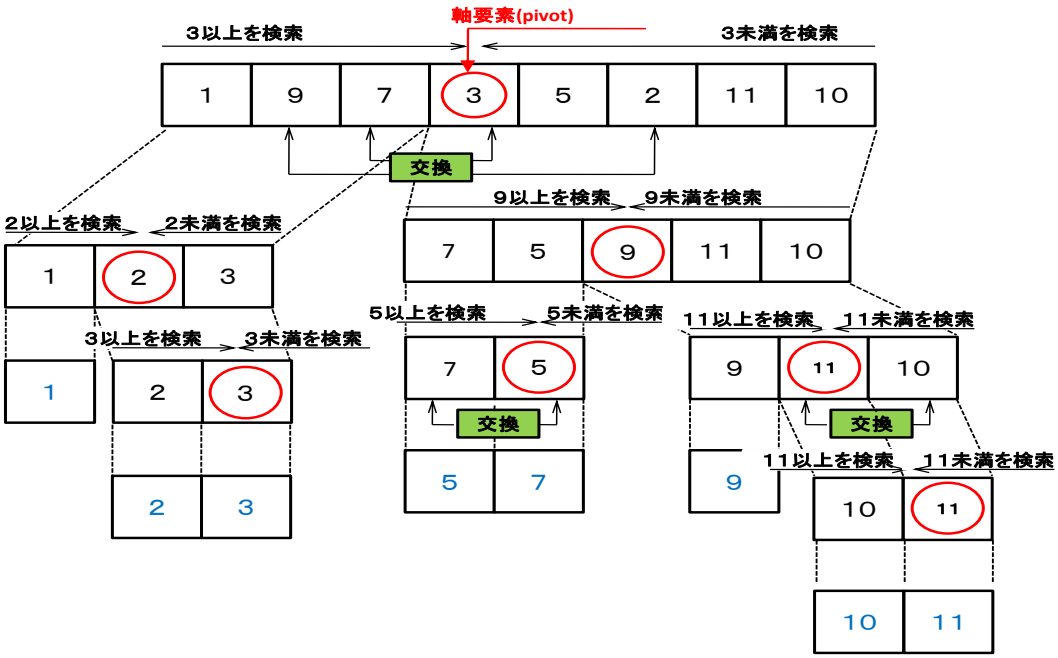

図 1：クイックソートのアルゴリズム

まず、「軸要素」と呼ばれるデータ値を決定する。この値は、データ全体を２つに分割するときのしきい値として使われる。軸要素は、分割が 均等に行われるように選ぶのが望ましいが、その選択に時間をかけると、かえって並べ替えの時間を大きくしてしまう。一般には、次のような方法がよく用いられている。

1. データの先頭の要素を軸要素とする
2. ランダムに１つ選ぶ
3. ランダムに３つ選んで、その中央値を取る
4. 左から見て最初に得られた２つの異なる値の大きいほうを取る

図１では、３の中央値を取る方法で軸要素を決定している。

次に、左端から軸までのデータで軸要素以上の値を、右端から軸までのデータで軸要素以下の値をそれぞれ探索していき、要素が見つかれば２つの値の交換を行う。この探索を交換する要素が無くなるまで行い、交換する要素が無くなれば軸までの値のデータと軸から最後までの値のデータに分割する。

そしてこれらの処理を要素が１つになるまで繰り返し行うことで、ソートが完了する。

クイックソートはデータの比較と交換回数が非常に少ないことが特徴で、一般的なバラバラなデータに対して最も効率よくデータのソートを行うことができる。だが、データの並び方によっては必ずしも高速であるということはなく、他のソート処理を行った方が高速である場合もある。

ここで n をソートすべきデータ要素数とすると、平均計算量は nlogn、最悪計算量は n^2 となる。性能は比較的優秀でありデータ数が多い処理に適しているクイックソートであるが、安定性はあまり良くなく使用には注意が必要である。

2.2　バブルソートのアルゴリズム

バブルソートの一連のアルゴリズムを図２に示す。

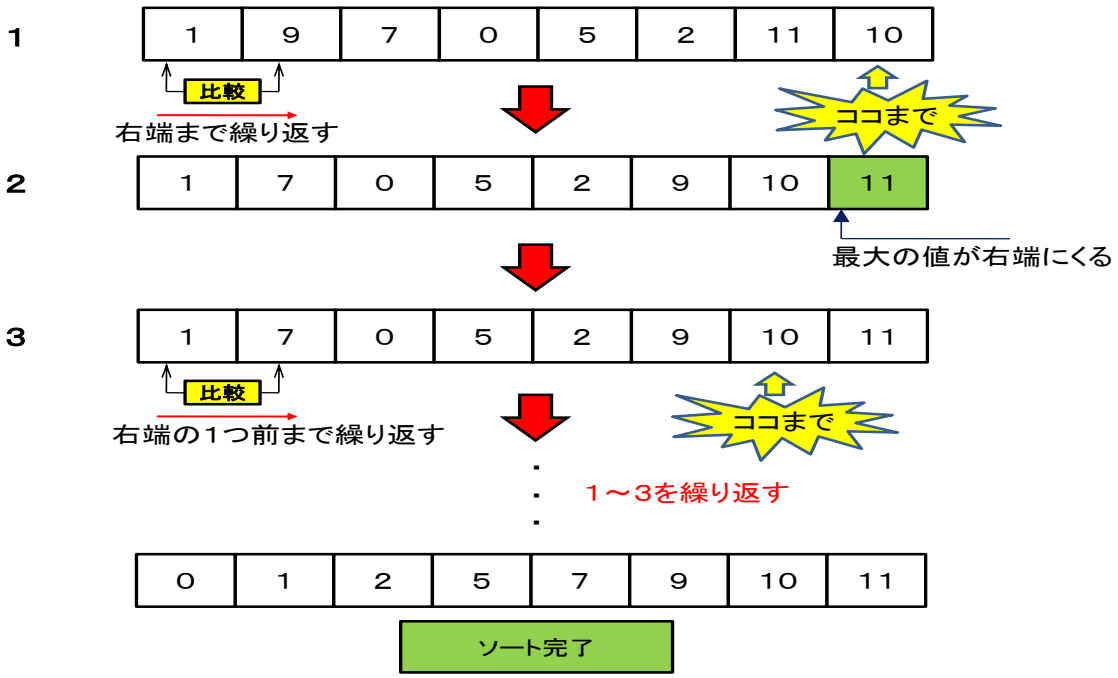

図 ２：バブルソートのアルゴリズム

バブルソートは、まず一番左の値とその隣の値を比較し、もし左の値が右の値より大きければ交換、左の値が右の値より小さければ交換しない。この動作を一番右端まで繰り返すと、一番大きな値が右端にくることになる。

そして次にまた一番左から順に同じ動作を繰り返し、右端の１つ手前まで比較を繰り返す。これで２番目に大きな値が右端から１つ手前にくることになる。

これらの動作を繰り返し行うことでソートが完了する。

バブルソートはデータの個数を n としたときに n（n-1）／２回の比較が行われることになる。そのためデータの個数が多くなれば多くなるほど処理速度も多くなるが、シンプルなアルゴリズムのため実装が容易でありデータ数が少ない時には手軽に実装できる。

ここで n をソートすべきデータ要素数とすると、最悪計算量は n^2 となる。速度は比較遅いバブルソートであるが、安定性は高くデータ数が少ない処理には適している。

3.　クイックソート回路の設計

3.1　全体のモジュール構成

クイックソートの全体のモジュール構成図を図 3 に示す。まず初期値が Memory モジュールに書き込まれる。そして比較するデータとそのアドレスをそれぞれのモジュールに送る。それらは Read_left、Read_right、Read_pivot を通り Compare_big と Compare_small で比較され、そこで比較された条件により Write_and_swap で交換が行われる。条件がそろわなければ交換は行われず、値とアドレスはそのまま Memory に返される。Write_and_swap ではデータ分割が必要な時にデータと同時に reflexive という信号を Memory に送る。また Memory モジュールではソート時に分割データの右端のアドレスが必要であり、reflexive の信号で Stack_address にそのアドレスを保存し、必要な時に取り出すことを行う。

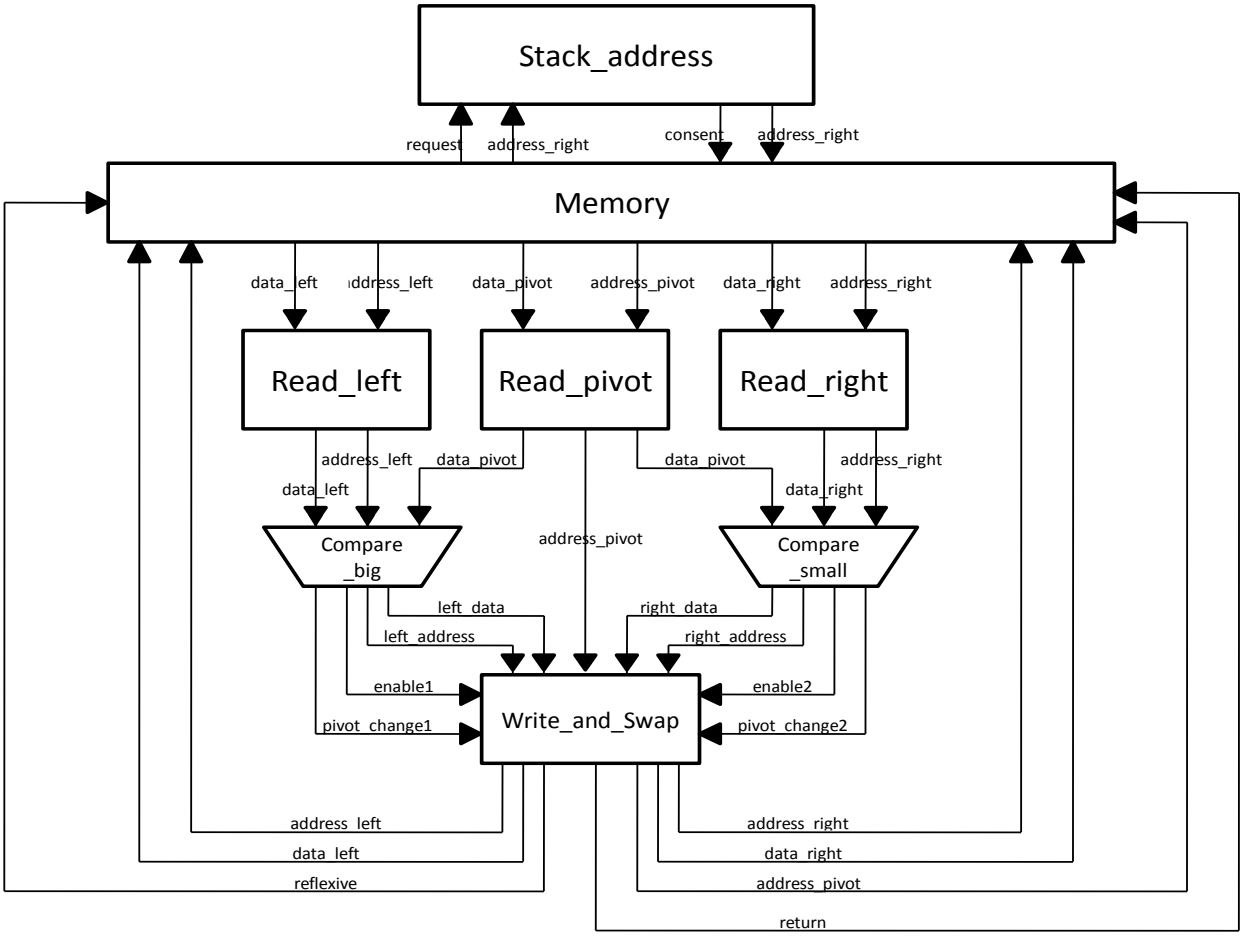

図 3：クイックソートの全体のモジュール

3.2　各モジュールの説明

3.2.1　Memory モジュール

図 4 に Memory の入出力インターフェースを、信号線の説明を表 1 に示す。

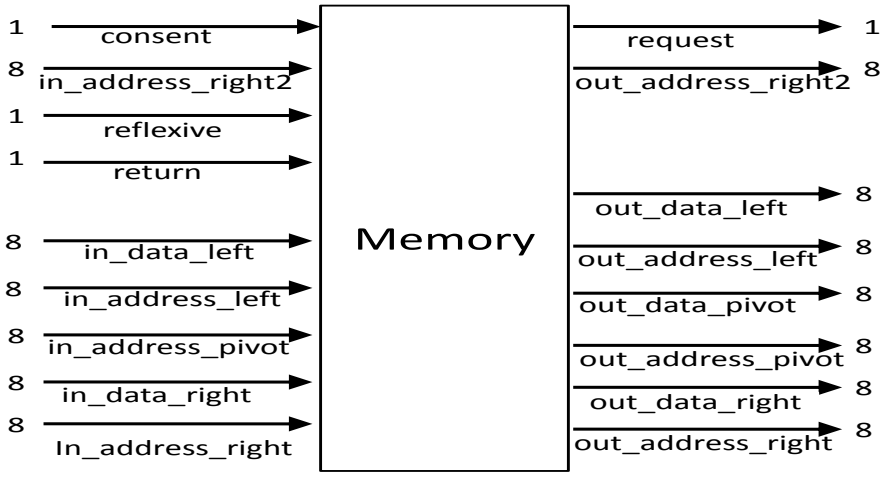

図 4：Memory モジュール

Memory モジュールは 9 つの入力と 8 つの出力から構成されている。比較データとそれぞれのアドレスの入出力、ソートするデータの書き込み、そしてデータ分割時のアドレスの入出力が主な役割である。データの初期値はこのモジュールで書き込まれ、それをそれぞれのモジュールに送り出し比較・交換が行われ、それらのデータの受け取り書き込みを行う。また、データ分割時の信号である reflexive が 1 の時に、データの分割を行う。その際に必要である分割前のデータの右端のアドレスを Stack_address モジュールに送り出し、そのアドレスが必要な時に取り出すことを行う。なお、最終的なデータもこのモジュールに書き込まれる。

表 1：Memory の信号線

信号名	方向	幅[bit]	詳細
in_data_left	input	8	左側から送り出す値
in_address_left	input	8	in_data_left のアドレス
in_data_right	input	8	右側から送り出す値
in_address_right	input	8	in_data_right のアドレス
in_address_pivot	input	8	軸要素である in_data_pivot のアドレス
return	input	1	スワップがおこなわれたサイン
reflexive	input	1	データが分割されたサイン
consent	input	1	stack_address からアドレスを受け取れる状態のサイン
in_address_right2	input	8	stack_address に送る
out_data_left	output	8	データが書き込まれる左側の値
out_address_left	output	8	out_data_left のアドレス
out_data_right	output	8	データが書き込まれる右側の値
out_addess_right	output	8	out_data_right のアドレス
out_data_pivot	output	8	軸要素であるピボットの値
out_address_pivot	output	8	out_data_pivot のアドレス
request	output	1	stack_address からのアドレスが必要であるサイン
out_address_right2	output	8	stack_address に送る右側のデータのアドレス

3.2.2　Read_left モジュール、Read_right モジュール、Read_pivot モジュール

図 5 に Read_left、Read_right、Read_pivot の入出力インターフェースを、信号線の説明をそれぞれ表 2、表 3、表 4 に示す。

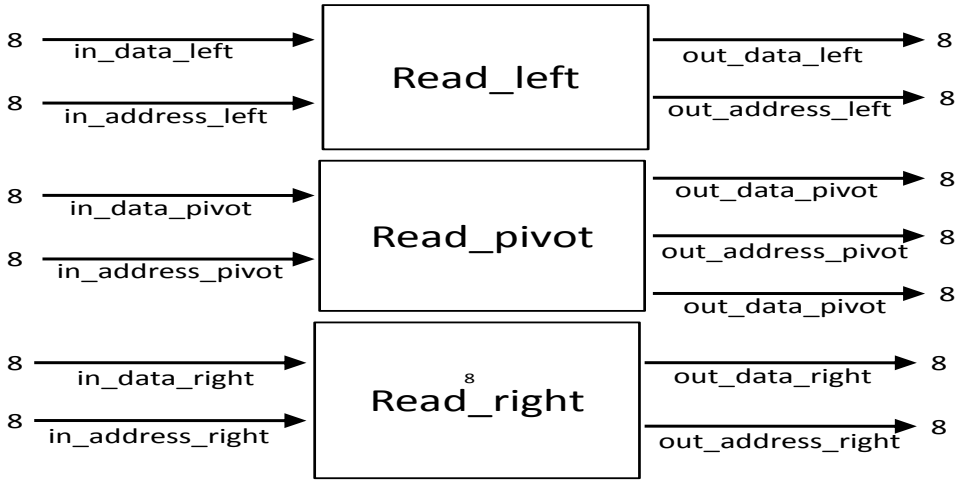

図 5：Read_left, Read_right, Raed_right モジュール

これら 3 つのモジュールはデータの流れを見やする為に作成したモジュールであり、これら自身は特別な動作を行うものではない。Read_left は Memory の左側から送られてきたデータとアドレスを、Read_right は Memory の右側から送られてきたデータとアドレスを、そして Read_pivot は Memory のデータの要素軸であるピボットデータとそのアドレスをそれぞれ比較のモジュールである Compare_big と Compare_small に送っている。Read_pivot のみアドレスを Write_and_swap に送っている。

表 2：Raed_left の信号線

信号名	方向	幅 (bit)	詳細
in_data_left	input	8	左側から送り出す値
in_address_left	input	8	in_data_left のアドレス
out_data_left	output	8	左側から送り出す値
out_address_left	output	8	out_data_right のアドレス

表 3：Read_right の信号線

信号名	方向	幅(bit)	詳細
in_data_right	input	8	右側から送り出す値
in_address_right	input	8	in_data_right のアドレス
out_data_right	output	8	右側から送り出す値
out_address_right	output	8	out_data_right のアドレス

表 4：Read_pivot の信号線

信号名	方向	幅(bit)	詳細
in_data_pivot	input	8	軸要素であるピボットの値
in_address_pivot	input	8	in_data_pivot のアドレス
out_data_pivot	output	8	軸要素であるピボットの値
out_address_pivot	output	8	out_data_pivot のアドレス

3.2.3　Compare_big モジュール、Compare_small モジュール

図 6 に Compare_big、Compare_small の入出力インターフェースを、信号線の説明をそれぞれ表 5、表 6 に示す。

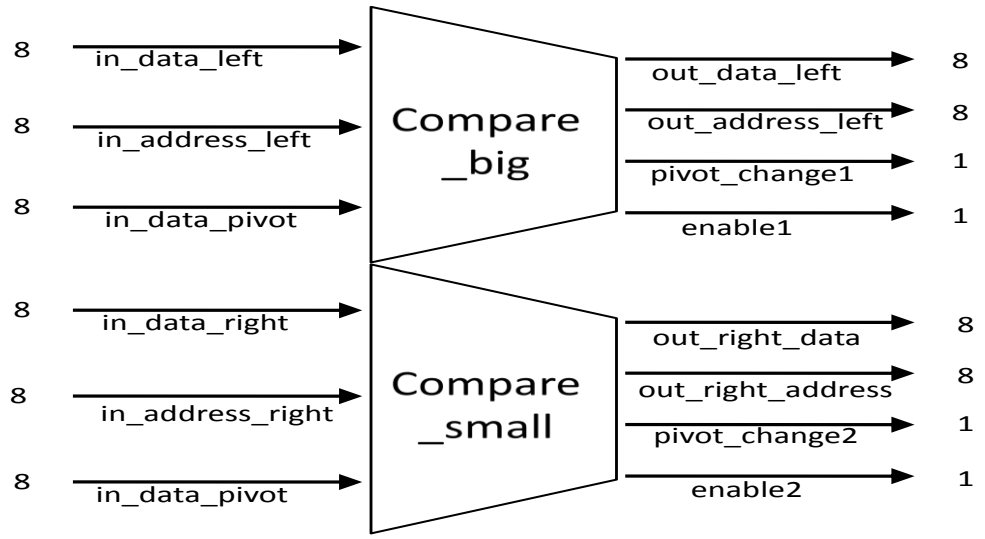

図 6：Compare_big, Compare モジュール

Compare_big モジュールは、Read_left モジュールからのデータとアドレス、そしてピボットデータを受け取り、それら２つのデータの比較を行う。もし data_left よりも data_pivot のデータの方が大きければ enable1 に 1 を送る。そうでなければ enable1 には 0 を送る。比較の結果にかかわらず、data_left と address_left はそのまま次のモジュールへと送られる。また、data_left と pivot_data が同じ値のときは軸要素である pivot_data 自身が交換され、軸要素の位置が移動するという信号である pivot_change1 に 1 が送られる。

Compare_small モジュールも同様に、Read_right モジュールからのデータとアドレス、そしてピボットデータを受け取り、それら２つのデータの比較を行う。もし data_left よりも data_pivot のデータの方が小さければ enable2 に 1 を送る。そうでなければ enable2 には 0 を送る。こちらのモジュールも比較の結果にかかわらず、data_right と address_right はそのまま次のモジュールへと送られる。また、data_right と pivot_data が同じ値のときは軸要素である pivot_data 自身が交換され、軸要素の位置が移動するという信号である pivot_change2 に 1 が送られる。

表 5：Compare_big の信号線

信号名	方向	幅(bit)	詳細
in_data_left	input	8	左側から送り出す値
in_address_left	input	8	in_data_left のアドレス
in_data_pivot	input	8	軸要素であるピボットの値
out_data_left	output	8	左側から送り出す値
out_address_left	output	8	out_data_left のアドレス
enable1	output	1	スワップが可能というサイン
pivot_change1	output	1	軸要素が変わるというサイン

表 6：Compare_small の信号線

信号名	方向	幅(bit)	詳細
in_data_right	input	8	右側から送り出す値
in_address_right	input	8	in_data_right のアドレス
in_data_pivot	input	8	軸要素であるピボットの値
out_data_right	output	8	右側から送り出す値
out_address_right	output	8	out_data_right のアドレス
enable2	output	1	スワップが可能というサイン
pivot_change2	output	1	軸要素が変わるというサイン

3.2.4 Write_and_swap モジュール

図 7 に Write_and_swap の入出力インターフェースを、信号線の説明を表 7 に示す。

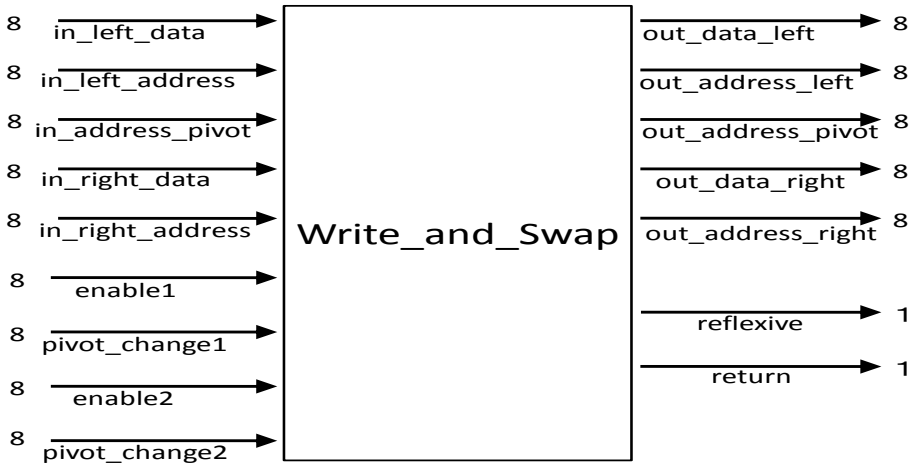

図 7：Write_and_swap モジュール

Write_and_swap モジュールでは主にデータの交換であるスワップと Memory モジュールへ書き込むデータとアドレスの決定を行う。入力信号である enable1 と enable2 が 1 の時にスワップを行う。enable1 と enable2 のどちらか片方だけが 1 の時は 1 の方のアドレスは進めずもう片方に 1 がくるまで待機しておく。スワップが行われた時には return 信号を Memory に送り、初めのアドレスに戻る。また交換するデータが無くなったときは、reflexive に 1 を送りデータの分割を行う信号を送る。

表 7：Write_and_swap の信号線

信号名	方向	幅(bit)	詳細
in_data_left	input	8	左側から送り出す値
in_address_left	input	8	in_data_left のアドレス
in_data_right	input	8	右側から送り出す値
in_address_right	input	8	in_data_right のアドレス
in_address_pivot	input	8	軸要素であるピボットのアドレス
enable1	input	1	スワップが可能というサイン
pivot_change1	input	1	軸要素が変わるというサイン
enable2	input	1	スワップが可能というサイン
pivot_change2	input	1	軸要素が変わるというサイン
out_data_left	output	8	データが書き込まれる左側の値
out_address_left	output	8	out_data_left のアドレス
out_data_right	output	8	データが書き込まれる右側の値
out_addess_right	output	8	out_data_right のアドレス
out_address_pivot	output	8	軸要素であるピボットのアドレス
return	output	1	スワップがおこなわれたサイン
reflexive	output	1	データが分割されたサイン

3.2.5　Stack_address モジュール

図 8 に Stack_address の入出力インターフェースを、信号線の説明をそれぞれ表 8 に示す。

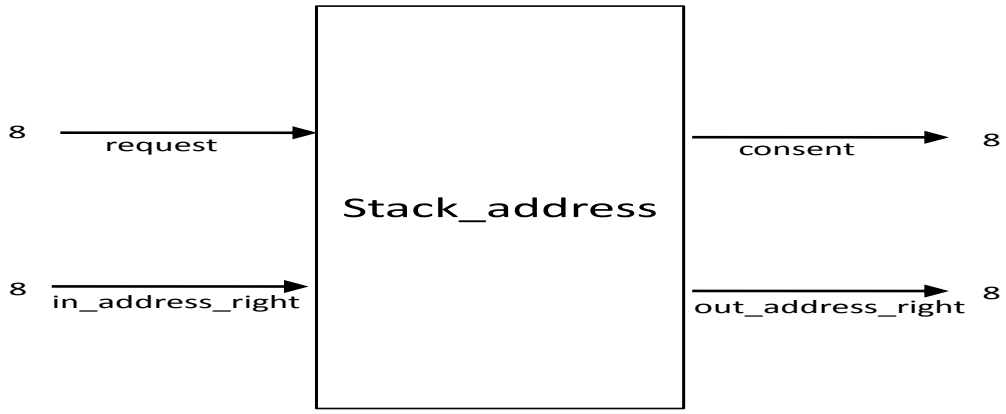

図 8：Stack_address

このクイックソートはデータ分割時に分割された左側のデータのソートを先に行い、左橋のデータの要素が 1 つになるまでデータ分割とソートを行う。その後に残されたデータをソートしていくものであり、そのためソート時に分割時のデータの右端のアドレスが必要になる。Stack_address モジュールでは、Memory とそれらのアドレスの受け渡しを必要に応じて行うモジュールである。Memory にデータ分割の信号である reflexve がくると、同時に右側のデータの右端のアドレスを Stack_address に送る。このモジュールはスタック構造から成っており、reflexive 信号がくる度にアドレスは新しいものが一番上にくるように保存される。もし、Memory モジュールで分割するデータが 1 つであれば request 信号が入り、データの受け渡しの準備ができれば consent 信号と同時にストックされているアドレスである address_right を Memory に返す。

表 8：Stack_address の信号線

信号名	方向	幅(bit)	詳細
request	input	1	データ分割時の右側のアドレスの要求
in_address_right	input	8	データ分割時の右側のアドレス
consent	output	1	アドレスを返す許可
out_address_right	output	8	データ分割時の右のアドレス

3.3　実験と考察

作成したクイックソートのハードウェアで実験を行い、その実験環境・実験データ・実験結果・ハードウェア性能をそれぞれ下に示す。

実験環境

名称：	Dell　Optiplex	動作周波数：	3.16 [GHz]
CPU：	IntelCore2　Duo	動作環境：	Modelsim

実験データ 1

ソート数：　64 [個]

ソート前データ：　1, 13, 0, 9, 7, 11, 12, 8, 6, 2, 3, 10, 14, 4, 15, 5, 16, 25,
31, 22, 28, 17, 30, 21, 26, 18, 23, 20, 27, 19, 29, 24, 32
38, 42, 45, 35, 40, 41, 33, 37, 46, 43, 47, 36, 39, 44, 34,
48, 49, 57, 53, 58, 50, 60, 54, 61, 51, 59, 55, 52, 62, 63, 56

実験結果 1

実行クロック数：　1560 [clk]

ソート後データ：　0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 19, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63

実験データ 2

ソート数：　128 [個]

ソート前データ：　1, 13, 0, 9, 7, 11, 12, 8, 6, 2, 3, 10, 14, 4, 15, 5, 16, 25,
31, 22, 28, 17, 30, 21, 26, 18, 23, 20, 27, 19, 29, 24, 32
38, 42, 45, 35, 40, 41, 33, 37, 46, 43, 47, 36, 39, 44, 34,
48, 49, 57, 53, 58, 50, 60, 54, 61, 51, 59, 55, 52, 62, 63, 56,
64, 68, 73, 74, 66, 76, 92, 93, 80, 89, 87, 91, 92, 88, 86, 82,
99, 109, 96, 105, 103, 103, 107, 108, 104, 102, 98, 147, 141,
112, 137, 151, 123, 156, 136, 118, 130, 131, 161, 173, 160,
169, 167, 171, 172, 168, 166, 162, 163, 176, 185, 183, 187,
188, 184, 182, 178, 179, 205, 208, 233, 199, 219, 236, 248

実験結果２

実行クロック数： 4090 [clk]

ソート後データ： 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 19, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 68, 73, 74, 76, 80, 82, 86, 87, 88, 89, 91, 92, 93, 96, 98, 99, 102, 103, 104, 105, 107, 108, 109, 112, 118, 123, 130, 131, 136, 137, 141, 147, 151, 156, 160, 161, 162, 163, 166, 167, 168, 169, 171, 172, 173, 176, 178, 179, 182, 183, 184, 185, 187, 188, 199, 205, 208, 219, 233, 236, 248

表 9：クイックソートのハードウェア性能

モジュール	回路規模 ［スライス］		遅延 ［ns］	
	データ数 64[個]	データ数 128[個]	データ 64[個]	データ数 128[個]
Top	0	0	×	×
Memory	1439	3242	7.298	7.786
Stack_address	500	500	6.293	6.293
Read_left	0	0	4.275	4.275
Read_right	0	0	4.275	4.275
Read_pivot	0	0	4.275	4.275
Compare_big	7	7	6.105	6.105
Compare_small	7	7	6.105	6.105
Write_and_swap	32	32	6.871	6.871

まず、実験結果についてはデータ数を 64 個とした場合、1560 [clk] という結果になり、これは例えば 1 [clk]を 100[ps]とした場合、156 [ns] という非常に高速な時間である。そして、データ数を 2 倍にした 128 個をソートする場合、4090 [clk] となり、ソート数が 64 個の場合と比べて約 2.6 倍の処理速度となっている。

またこれは Modelsim での実行時間であり、実際に FPGA ボード上で動かした場合の処理速度は ISE での論理合成時に回路が最適化されるため、さらに高速なものとなると予想される。

ハードウェア性能においては、Xilinx 社の提供する ISE を用いて測定した。レジスタを多く使用した Memory モジュールと Stack_address モジュールは回路規模が大きくなっている。遅延についても大量のデータの書き込み動作が行われる Memory モジュールではやはり処理に時間がかかっていると考えられる。

4.　バブルソート回路の設計

4.1　全体のモジュール構成

バブルソートの全体のモジュール構成図を図 9 に示す。
まず、初期値が Memory モジュールに書き込まれる。そして一番左端のデータとその隣のデータ、そしてそれらのアドレスが同時に Compare_and_exchange モジュールに送られる。Compare_and_exchange モジュールでは比較と交換が行われ、必要に応じてスワップを行い、Memory にそれぞれのデータとアドレスを返す。データの比較と交換のデータ列の右端まで行われると、Memory のデータ列の一番右端にデータの最大値がくる。そして、データの右端を最大値の 1 つ手前として同様に比較と交換を繰り返す。
以上の動作を繰り返し行うことでソートが完了する。

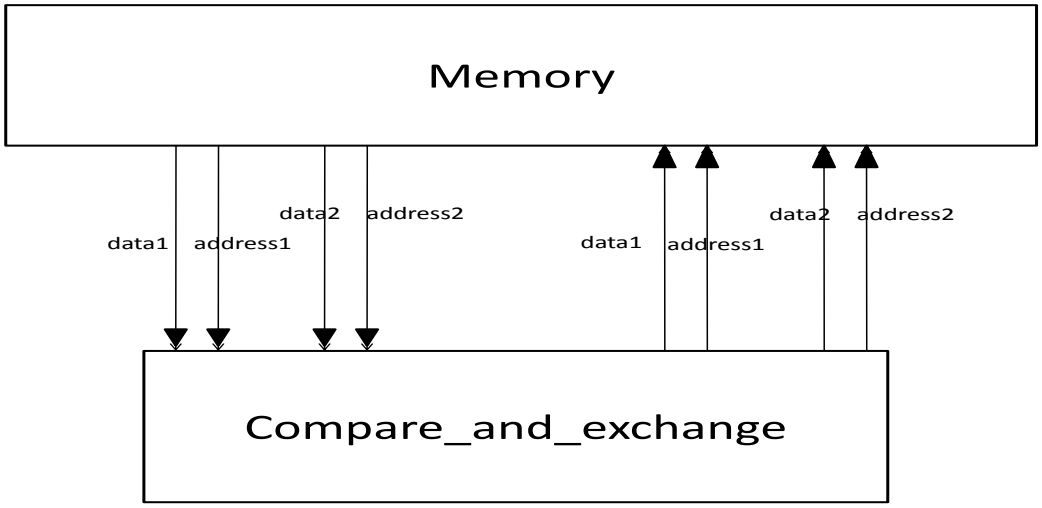

図 9：バブルソートの全体のモジュール

4.2　各モジュールの説明

4.2.1　Memory モジュール

図 10 に Memory モジュールの入出力インターフェースを、信号線の説明を表 9 に示す。

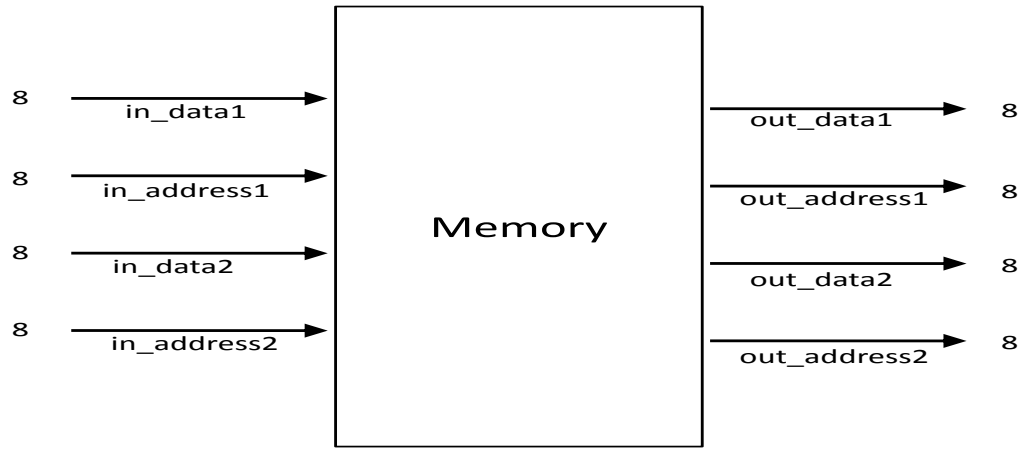

図 10：Memory モジュール

Memory モジュールは、主にデータの書き込みとデータ・アドレスの受け渡しを行う。最初に一番左のデータとアドレス、そしてそのとなりのデータとアドレスを同時に Compare_and_exchange に送る。そして Compare_and_exchange から帰ってきたデータとアドレスを受け取り、データの書き込みを行う。そしてアドレスを１つ右にそれぞれずらして、再びデータとアドレスを Compare_and_exchange へと送る。この動作を繰り返すと右端にデータの最大値がくるので、その時にアドレスを 0 に戻して最大のデータの１つ手前まで同様の動作を繰り返し行う。

表 10：Memory の信号線

信号名	方向	幅(bit)	詳細
in_data1	input	8	受け取るデータ 1
in_address1	input	8	in_data1 のアドレス
in_data2	input	8	受け取るデータ 2
in_address2	input	8	in_data2 のアドレス
out_data1	output	8	送り出すデータ 1
out_address1	output	8	in_data1 のアドレス
out_data2	output	8	送り出すデータ 2
out_address2	output	8	in_data2 のアドレス

4.2.2　Compare_and_exchange モジュール

図 11 に Compare_and_exchange モジュールの入出力インターフェースを、信号線の説明を表 10 に示す。

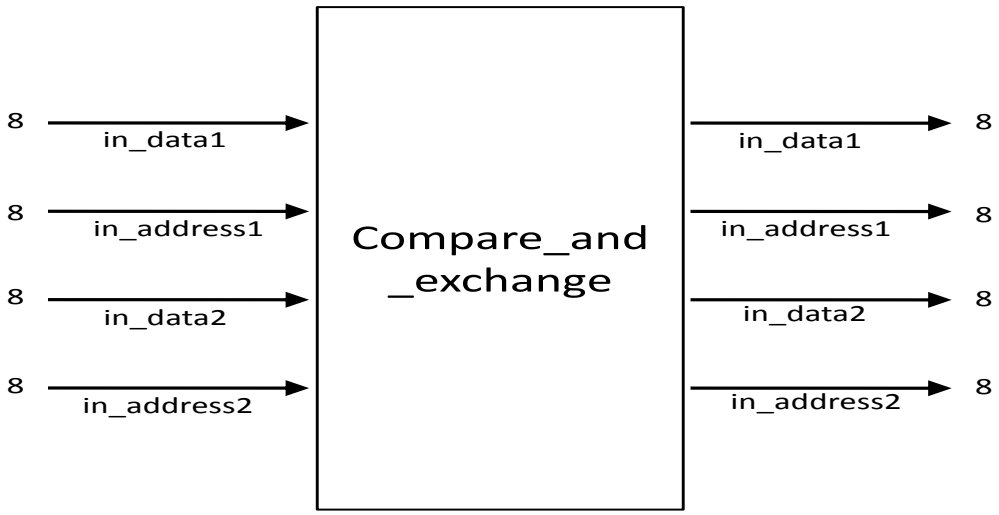

図 11：Compare_and_exchange モジュール

Compare_and_exchange モジュールでは、Memory モジュールから送られてきたデータの比較・交換を行う。もし左側のデータが右側のデータより大きかったらそれらを交換し、そうでなければ交換しない。それらのデータとアドレスを Memory モジュールに戻す作業を繰り返し行う。

表 11：Compare_and_exchange の信号線

信号名	方向	幅(bit)	詳細
in_data1	input	8	受け取るデータ 1
in_address1	input	8	in_data1 のアドレス
in_data2	input	8	受け取るデータ 2
in_address2	input	8	in_data2 のアドレス
out_data1	output	8	送り出すデータ 1
out_address1	output	8	in_data1 のアドレス
out_data2	output	8	送り出すデータ 2
out_address2	output	8	in_data2 のアドレス

4.3　実験と考察

作成したクイックソートのハードウェアで実験を行い、その実験環境・実験結果・ハードウェア性能をそれぞれ下に示す。

実験環境

名称：	Dell　Optiplex	動作周波数：	3.16 [GHz]
CPU：	IntelCore2　Duo	動作環境：	Modelsim

実験データ 1

ソート数：　64 [個]

ソート前データ：
1, 13, 0, 9, 7, 11, 12, 8, 6, 2, 3, 10, 14, 4, 15, 5, 16, 25,
31, 22, 28, 17, 30, 21, 26, 18, 23, 20, 27, 19, 29, 24, 32
38, 42, 45, 35, 40, 41, 33, 37, 46, 43, 47, 36, 39, 44, 34,
48, 49, 57, 53, 58, 50, 60, 54, 61, 51, 59, 55, 52, 62, 63, 56

実験結果 1

実行クロック数：　2070 [clk]

ソート後データ：
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 19, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63

実験データ 2

ソート数：　128 [個]

ソート前データ：
1, 13, 0, 9, 7, 11, 12, 8, 6, 2, 3, 10, 14, 4, 15, 5, 16, 25,
31, 22, 28, 17, 30, 21, 26, 18, 23, 20, 27, 19, 29, 24, 32
38, 42, 45, 35, 40, 41, 33, 37, 46, 43, 47, 36, 39, 44, 34,
48, 49, 57, 53, 58, 50, 60, 54, 61, 51, 59, 55, 52, 62, 63, 56,
64, 68, 73, 74, 66, 76, 92, 93, 80, 89, 87, 91, 92, 88, 86, 82,
99, 109, 96, 105, 103, 103, 107, 108, 104, 102, 98, 147, 141,
112, 137, 151, 123, 156, 136, 118, 130, 131, 161, 173, 160,
169, 167, 171, 172, 168, 166, 162, 163, 176, 185, 183, 187,
188, 184, 182, 178, 179, 205, 208, 233, 199, 219, 236, 248

実験結果２

実行クロック数： 8250 [clk]

ソート後データ： 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 19, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 68, 73, 74, 76, 80, 82, 86, 87, 88, 89, 91, 92, 93, 96, 98, 99, 102, 103, 104, 105, 107, 108, 109, 112, 118, 123, 130, 131, 136, 137, 141, 147, 151, 156, 160, 161, 162, 163, 166, 167, 168, 169, 171, 172, 173, 176, 178, 179, 182, 183, 184, 185, 187, 188, 199, 205, 208, 219, 233, 236, 248

表 12：バブルソートのハードウェア性能

モジュール	回路規模 ［スライス］		遅延 ［ns］	
	データ数 64[個]	データ数 128[個]	データ数 64[個]	データ数 128[個]
Top	0	0	×	×
Memory	955	1859	8.834	9.819
Compare_and_exchange	13	13	6.465	6.465

まず、実験結果１については処理速度が 2070［clk］という結果になり、これは 1 [clk]を 100[ps]とした場合、207［ns］という非常に高速な時間である。そして、ソート数を 2 倍の 128 個とした場合、処理速度は 8250［clk]となりソート数が 64 個の場合と比べて約 4 倍の処理速度が必要とされている。これはもちろんデータの内容にも関係するが、クイックソートが 2.6 倍だったことを考慮すると、やはりデータ数を増加させた場合はクイックソートの方が優れていると思われる。

ハードウェア性能には、レジスタを多く使用した Memory モジュールの回路規模が大きくなっている。処理速度はクイックソートと比較して劣っているバブルソートだが、回路規模をみると約 2 倍の回路規模を必要としているためソート数が少ないを行う場合にはバブルソートの方がコストパフォーマンスは優れていると考えられる。

5.　ハード/ソフト最適分割の検討

5.1　CPU 負荷部の調査

ハードウェアとソフトウェアの分割探索のために、まずハードウェアで作成したアルゴリズムをソフトウェアで実現するシステムの設計を行った。ハードで作成したモジュールをそれぞれ関数ごとに分け、その関数ごとのCPU負荷を計測した。結果は図12、図13の通りである。

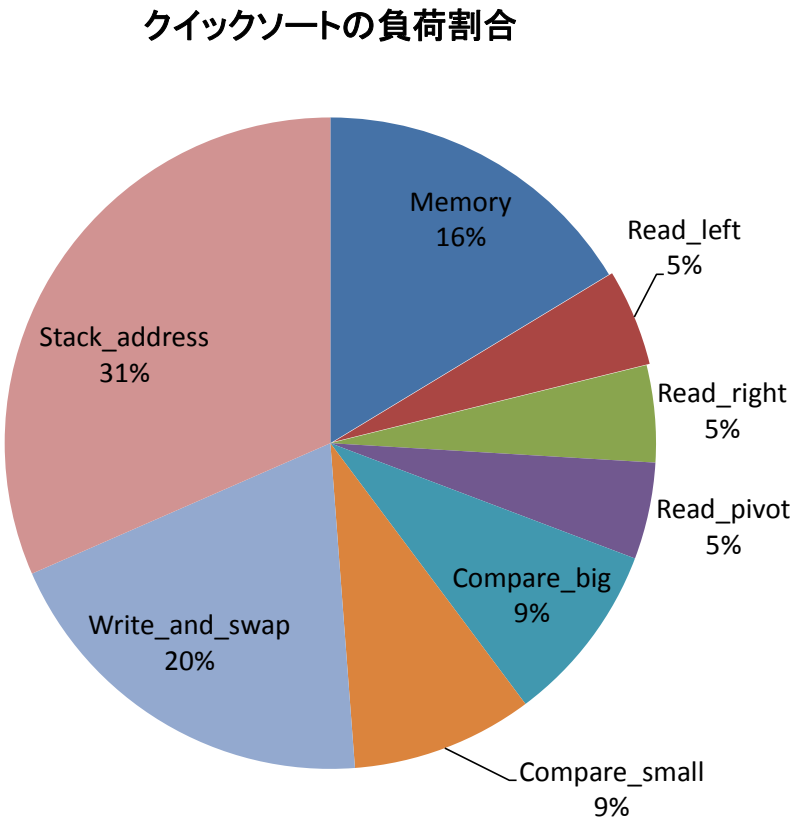

図 12：クイックソートの負荷割合

クイックソートについては、アドレスの一部を保存する Stack_address モジュールが一番大きな負荷となっていることがわかる。そして次にデータの交換を行う Write_and_swap モジュール、続いてデータの読み書きを行う Memory モジュールとなっている。
一番処理が大きな Stack_address では、配列へのアクセスが頻繁に行われるためデータの読み出しと書き込みの処理に時間がかかると思われる。次に、Write_and_swap モジュール

は配列へのアクセスにポインタを使って処理しているために時間がかかると思われる。そして、Memory モジュールは Stack_address モジュールと同様に配列への読み書きの処理が多いためと考えられる。

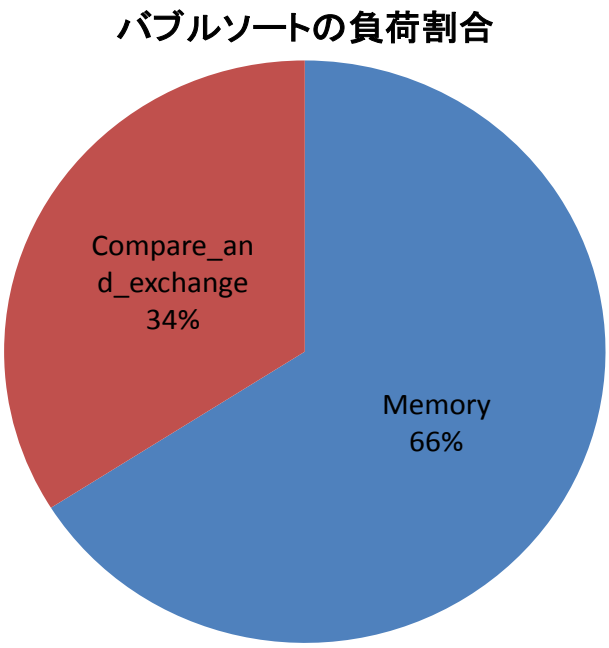

図 13：バブルソートの負荷割合

バブルソートについては、Memory モジュールの方が Compare_and_exchange モジュールより負荷が大きいと思われる。これはデータの書き込みと読み出しを行う Memory モジュールでは、比較と交換を行うだけの Compare_and_exchange モジュールより処理が重いためであると考えられる。

5.2　ハード/ソフトの分割パターンと考察

クイックソートのハード/ソフトの分割パターンを表 13 に、バブルソートのハード/ソフト示す。分割パターンを表 14 に示す。

表 13：クイックソートのハード/ソフト分割パターン

分類	ハード	ソフト	回路規模	クロック数
A	Memory, Raed_left, Read_right, Read_pivot, Compare_big, Compare_small, Write_and_swap, stack_address		1985	1560[clk]
B	Memory	Raed_left, Read_right, Read_pivot, Compare_big, Compare_small, Write_and_swap, stack_address	1439	×
C	Memory, Stack_address	Raed_left, Read_right, Read_pivot, Compare_big, Compare_small, Write_and_swap	1939	×
D	Raed_left, Read_right, Read_pivot, Compare_big, Compare_small, Write_and_swap	Memory, Stack_address	46	×
E		Memory, Raed_left, Read_right, Read_pivot, Compare_big, Compare_small, Write_and_swap, stack_address	0	780000[clk]

A については、すべての処理をハードウェア化しているため速度においては最も高速な性能を示す。しかし、すべてをハードウェアで作成しなければならないのでコストもかかり、また回路規模もすべての分類の中で一番おおきくなる。

B については、処理の重い部分をできるだけハードウェア化している。そのため速度は A には劣るものの高速な処理を実現できる。しかし、ハードウェア化する部分も多く、回路規模も比較的大きいにもかかわらず処理速度も A に比べて劣るためにあまりバランスのとれたものではない。

C については、比較的処理の重い部分がバランスよくハードウェア化されるために速度もある程度早くなる。さらにハードウェア化する部分も多くないために回路規模も比較的コンパクトになりコストもあまりかからない。

D については、ハードウェア化する部分を処理の軽い部分でおさえるため、回路規模は小さくすることができるが処理速度は期待できない。この分割パターンも C に比べてあまりバランスのとれたものではない。

E については、すべての処理をソフトウェアで行っているため、処理速度については遅いものの、ハードウェア回路を別で設計する必要がないために回路規模も 0 でコストを抑えることができる。

表 14 ：バブルソートのハード/ソフト分割パターン

分類	ハード	ソフト	回路規模	クロック数
A	Memory, Comapre_and_exchange		968	2070[clk]
B	Memory	Compare_and_exchange	955	×
C	Compare_and_exchange	Memory, Stack_address	13	×
D		Memory, Compare_and_exchange	0	828000[clk]

A については、こちらもすべての処理をハードウェアで行っているため速度においては最も高速な性能を示す。しかし、すべてをハードウェアで作成しなければならないのでコストもかかり、また回路規模もすべての分類の中で一番大きくなる。

B については、処理の重い Memory の部分をハードウェア化しており、速度は A には劣るものの高速な処理を実現できる。比較的バランスのとれた分割パターンである。

C については、Memory より処理の少ない Comapre_and_exchange の部分をハードウェア化しているので B よりもコストパフォーマンスは悪いが、速度はある程度のものが期待できる。

D については、すべての処理をソフトウェアで行っているため、処理速度については遅いものの、ハードウェア回路を別で設計する必要がないために回路規模も 0 でコストを抑えることができる。

6. おわりに

本研究では、現在のシステムLSIの設計について考察し、クイックソート回路とバブルソート回路を設計した。さらにハード/ソフト協調設計における最適分割を検証するために、ソフトウェアでそれらのアルゴリズムを実現しCPU負荷を測定、分割パターンの検討を行った。

本論文では、ソフトウェア設計のアルゴリズムをハードウェア設計に用いることを試みたが、実現が困難であるためにハードウェアで実現可能な別のアルゴリズムを考え、ハードウェアでのアルゴリズムを基にソフトウェア設計を行った。今研究を通して、実行時のクロック数を減らすアルゴリズムの設計の重要性とともに、ハード/ソフトの協調設計の必要性を強く感じることになった。ハードウェア設計においては、クイックソート・バブルソート共にハードウェアでソフトウェア時の特徴・アルゴリズムを実現することができたことは非常に意義があることであったと思われる。特にクイックソートにおいては非再帰的な処理を行うアルゴリズムを考え出しハードウェア化することができた。

ハードウェア化した今後の課題として、Xilinx社の提供するソフト・マクロCPUであるMicroBlazeを用いて、FPGAボードの上に両システムを実装して実際に提案した分割パターンの実測値の測定を行い、その実用性・有用性がどれほどであるのかを検証することがあげられる。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導をいただきました山崎勝弘教授に深く感謝いたします。また、本研究の共同研究者である高性能計算研究室の皆様に心より深く感謝いたします。

参考文献

[1] 小林優:ハードウェア記述言語の速習＆実践 入門 Verilog-HDL 記述,CQ 出版社,2002.

[2] 並木秀明,前田智美,宮尾正大:実用入門ディジタル回路と Verilog-HDL,技術評論社,2004.

[3] 梅原直人：ハード/ソフト最適分割を考慮した AES 暗号システムと JPEG エンコーダの設計と検証, 立命館大学理工学部情報学科卒業論文, 2005.

[4] 和田智行：Misty1 暗号回路の設計とハード/ソフト最適分割の検討：ハード/ソフト最適分割を考慮した AES 暗号システムと JPEG エンコーダの設計と検証, 立命館大学理工学部情報学科卒業論文, 2005.