修士論文

C ソースコード解析による

ハード／ソフト最適分割システムの構築

氏　　名	:	和田　智行
学 籍 番 号	:	6162070143-1
指 導 教 員	:	山崎　勝弘　教授
提　出　日	:	2009 年 2 月 13 日

立命館大学大学院　理工学研究科　創造理工学専攻

内要梗概

本論文では、ハード／ソフト協調設計において、設計の早期段階でハードとソフトの最適な分割パターンを見出す手法について述べる。我々は実際に回路を設計する前に、対象問題のハードウェアのリファレンスともなるCソースコードのプロトタイプ､ユーザ要求、及びシステムの特徴を解析して、ハードウェアとソフトウェアの設計空間を分割するハード／ソフト最適分割システムを試作した。本システムはモジュール毎に、ソフトウェアクロックサイクル数、ソフトウェア負荷割合、ハードウェアクロックサイクル数、回路規模、メモリ量、ハードウェア動作周波数を算出し、それを元に最終的なシステムの分割案を提案する。本システムをMisty1暗号、AES暗号、及びSHA-1の実用アプリケーションに適用し、評価・検証した。

実装環境には、評価ボードとして、Virtex4-MB LX60 Development Board revision3を用い､Xilinx社が提供するソフトコアCPUのMicroBlazeを使用した｡これにVerilog-HDLで記述したハードウェアを接続することでハードウェアとソフトウェアの協調動作を行う。これらの環境を使用することで実測結果を測定し、本システムの解析結果と比較した。Misty1暗号、AES暗号、及びSHA-1の違った処理内容を持つアプリケーションを用いることで、実験の内容を幅広いものにした。全てのアプリケーションを4～5種類のモジュール構成にし、解析値と実測値を比較することで本システムの評価を行った。これにより、本システムで提案された分割案、及び本システムで解析されたアプリケーションの全分割パターンの評価結果が妥当なものを示し、本システムが有効であることがわかった。

本論文では、アプリケーションのソフトウェアプロトタイプとなるC言語から協調設計システムにおける設計空間の探索を行う手法を構築し、検討した。また、ハードウェアとソフトウェアが協調動作を行える環境を構築し、アプリケーションを分割したシステムの実機検証を行った。そして、解析結果と実機での実験結果を比較し、本研究の有効性を見出した。

目次

図目次

表目次

1. はじめに

近年、半導体の微細化技術の向上は著しく、1 チップに集積可能な論理回路の規模は飛躍的に向上している。その結果、LSI に対して高度で多様な機能の実現をもたらし、ワンチップ上にプロセッサと論理機能ブロックとしての IP コアなどのシステム機能を搭載する SoC（System on s Chip）、またはシステム LSI と呼ばれるものまでが出現した。現在では、LSI はありとあらゆるものに用いられており、大規模計算機やパーソナルコンピュータだけでなく、多くの家電製品、車載製品、通信機器など様々なものに搭載され、その用途は多様化してきている。これに伴い、高性能化、高機能化、低消費電力化など LSI に対する要求も多様化しており、これらの要求のバランスをとった LSI の設計が望まれている。また、組込み機器の開発では競争が激しく、開発サイクルの短縮、製品開発の効率化が求められている。SoC およびシステム LSI 設計において、これらの厳しい要求を満たすための設計手法のひとつにハードウェア／ソフトウェア協調設計があげられる。ハードウェア／ソフトウェア協調設計とは、ハードウェアの処理速度の高速性や消費電力の低さ、ソフトウェアの再利用性や柔軟性といった両方の利点を生かした高性能なシステムを実現するためのものである。ハードウェアとソフトウェアのバランスを取った設計は厳しい制約の仕様を満たしたシステムには必須であり、両者のバランスを誤った場合、仕様を満たせなくなり、設計の手戻りが発生し、大きな損失となる可能性がある。

本研究では、ハードウェアとソフトウェアの最適な分割を実現するために、対象となる問題のハードウェアのリファレンスにもなる C 言語のプロトタイプが完成した段階でソースコードを解析して、設計の早期段階から、ハードウェアとソフトウェアの最適な分割を見つけ出すことを目的とする[1]。C 言語のプロトタイプは関数をハードウェアモジュールに対応させて記述してあるもので、ハードウェアの仕様となる記述をしたソースコードである。

ハード／ソフト最適分割システムは C ソースコードを解析してモジュール毎に性能を算出し、評価して最適なパターン案を提示する。モジュール性能は実装環境を考慮し、実装環境にあった性能を算出する。C ソースコードを解析する理由は、ソフトウェア・ハードウェア両方の設計者が扱えるプログラミング言語であり、現在でも組込み機器のプロトタイププログラムに利用されているからである[1]。C ソースコードからモジュール毎にパラメータを作成し、そこにシステムの実装環境を考慮することで、各モジュールの性能を導き出す。その後、速度重視、回路規模重視、バランス重視、及び複数の項目を重視するといったユーザ要求重視の 4 つの観点より最適なシステム案を提示する。

本システムを、様々なアプリケーションに適用し評価する。使用するアプリケーションは ISO の標準暗号に採択されている秘密鍵暗号方式である MISTY1[10][11]、NIST が標準暗号規格とした共通鍵暗号方式である AES[26]、及び組込み用途向けのベンチマークである MiBench[27][28]より選出したアプリケーションを使用する。本システムで分割案を出力

したのち、実際にシステムを FPGA ボード上へ実装し、出力結果が有効であるか各関数・モジュール、分割案の性能をシステムの解析値と実測値を比較することで評価する。実装には Xilinx 社の提供するソフトコアプロセッサ「MicroBlaze」を中心に構築した FPGA システムを使用する。専用のバスを用いて MicroBlaze にハードウェアを接続することで、FPGA ボード上でハードウェア／ソフトウェア協調動作を行わせる。これらの FPGA 検証システムを用いて、アプリケーションの分割実験を行い、本システムの評価を行う。解析値と実測値の比較・評価をする他、本システムによる出力パターンが最適であるか、FPGA への実装結果と経験則より検証・評価する。また、本システムをマルチコア環境にも対応させ評価を行う。

本論文では、第 2 章で本システムの構成について述べる。第 3 章では、本システムの解析手法を述べる。第 4 章では、本システムを様々なアプリケーションに対し適用し、その実験結果を示し、本システムの評価・考察を行う。アプリケーションは MISTY1、AES、及び Mi-Bench より数種類を対象とする。

2. ハード／ソフト最適分割システム

2.1. システムの概要

ハードウェアとソフトウェアを最適に分割する手法には現状では設計者による経験則からの試行が多い。それに対し、本研究では C 言語により記述されたプロトタイプを解析して、機械的かつ早期に分割案を提示することにより、設計者を支援することを目指す。ハード／ソフト最適分割システムの処理の流れを図 1 に示す。

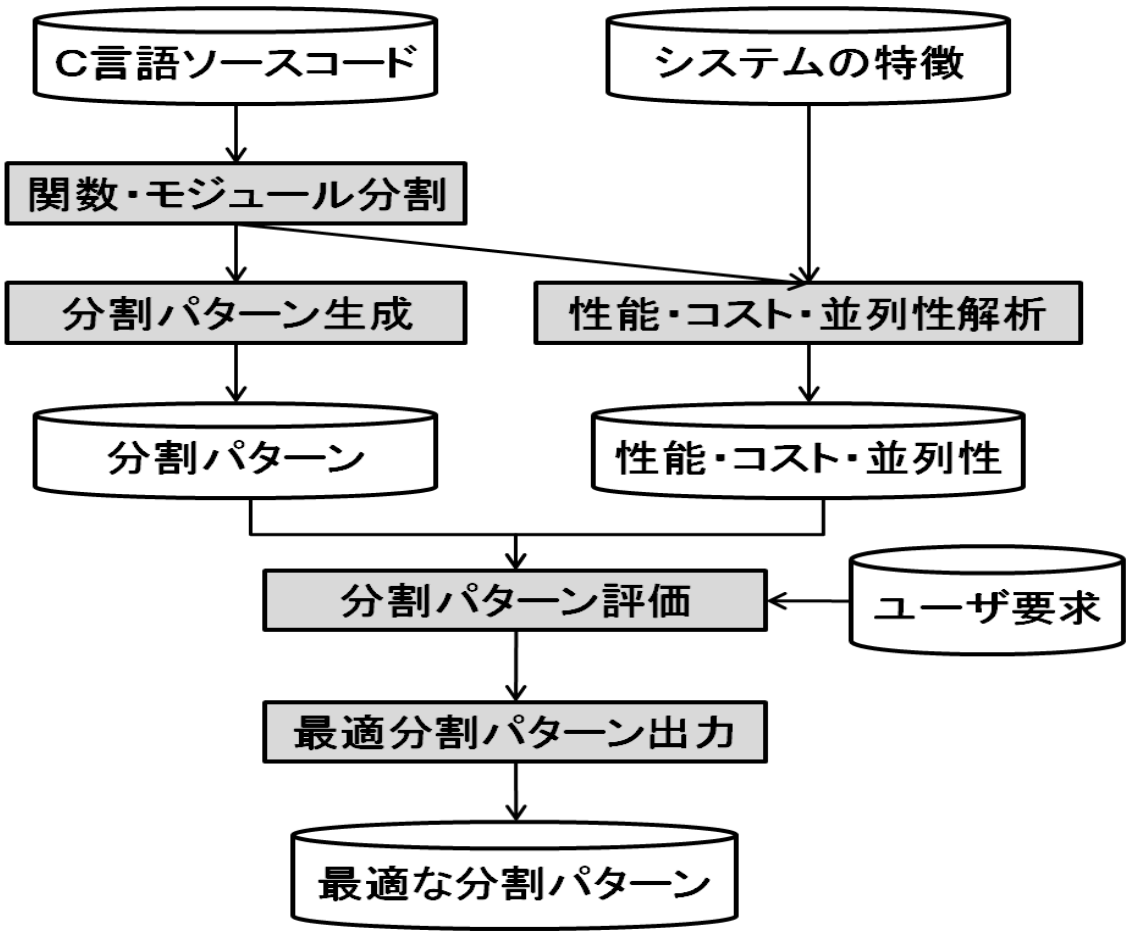

図 1　ハード／ソフト最適分割システムの流れ

本システムの入力は完成したハードウェアのリファレンスとなる C ソースコード、実験の対象の環境などのシステムの特徴を数値化したもの、及びユーザ要求を数値化したものである。システムの特徴は、主に実験対象のプロセッサの命令セットやアーキテクチャから予め抜き出したもので、命令毎のクロック数、パイプライン段数などである。ユーザ要求は処理速度、回路規模、並列性などの解析項目に対して重みを負荷したものと各性能に対する足きり項目である。まず、C ソースコードとシステムの特徴より、関数・モジュール

に分割し、それぞれの性能・コスト・並列性の解析をそれぞれ行う。一方で C ソースコードより全分割パターンを生成し、解析結果とユーザ要求を合わせることで評価を行い、優れたパターン候補を出力する。

2.2. ハード／ソフト最適分割における着目点

本システムで分割するために着目する協調システムでの性能項目などについて述べる。分割するために重視する要素として以下の 5 つに着目する。

(1) 実行サイクル数

実行サイクル数は、処理速度を決定する上で重要な要素であり、ソフトウェアプロトタイプと比較していかに回路規模を増やさずに実行サイクル数が減らせるかを考慮し、分割案を評価する。

本システムの解析ではハードウェアとソフトウェア両方の実行サイクル数を算出する。ハードウェア実行サイクル数は、設計者への依存度が大きいため、一定の条件を設けサイクル数を算出する。ソフトウェア実行サイクル数は実装環境を考慮し実行サイクル数を算出する。

(2) 回路規模

回路規模は最終的な製造コストにかかる問題であり、重要な要素である。ここにはハードウェア側のメモリに関しても含まれることとなる。いかに回路規模を増やさずに性能をあげることができるかを考慮し分割案を評価する。

本システムの解析では回路規模はハードウェア設計者に依存するため、最低でも必要とされる数値とソフトウェアで使われるメモリをハードウェア化した場合の数値を足し合わせたものを、実装環境を考慮し算出する。単位は slice である。

(3) メモリ量

メモリ量は、ソフトウェアで使用するスタック領域・スタティック領域・コード領域の使用量である。分割案の評価には足きり項目として使用する。

本システムの解析ではスタティック領域のメモリをハードウェアにした場合に削減されるメモリ量を参考データとして算出する。また、全ての領域の合計を足きり項目として使用する。実装環境やユーザ要求でメモリ量の上限がある場合、その値を超えた場合、ソフトウェアでの実行、もしくは実装自体が不可であると判断する。算出するデータの単位は Byte である。

(4) 動作周波数

動作周波数は、処理速度を決定する上で重要な要素である。しかし、SoC などのように

チップ上にプロセッサが搭載されたシステムでは、ほとんどの場合がプロセッサの周波数に依存するため、分割案の評価には足きり項目として使用する。

本システムの解析では、システム設計のための参考データとして算出する。また、足きり項目として使用する。実装環境のプロセッサの周波数を大きく下回る場合、もしくはユーザが設定した足きり値を下回る場合はハードウェア化は不可であると判断する。算出するデータの単位は MHz である。

(5) 消費電力

消費電力は近年の SoC における最も大きなトレンドであり、非常に重要な項目である。本システムでは現在、解析項目として実装されていないが今後実装の必要がある。式(1)に消費電力の計算式を示す。P は総消費電力、α はゲーティッドクロック機構を採用した場合の機能モジュール毎の生起確率、C はキャパシタンス、F は周波数、V は電圧である。周波数と電圧は FPGA ボード上では変更することはないので考慮しない。生起確率は静的に計算することで求められ、キャパシタンスはゲート数を数えることで算出できる。このように簡略化した式で消費電力計算を行う。

$$P \propto \sum \alpha_i \times C_i \times F \times V^2 \qquad \cdots\text{式(1)}$$

2.3. ハード／ソフト分割手法

本システムの内容について説明する。まず、本システムの概要として、入力と出力の関係を示したものを図 2 に示す。入力は C ソースコード、システムの特徴、ユーザ要求の 3 つであり、ユーザは出力としてモジュール毎の性能一覧と最適な分割案を参照することが可能となる。

次に、解析の対象となる C ソースコードに必要となる主な条件・サポート範囲・条件について示す。これらを満たさなければ本システムの対象外となる。

- GCC コンパイラでエラーなくコンパイルできること
- 各関数がハードウェアに対応するように設計されていること
- モジュールの入力と出力が関数の引数と戻り値に対応していること
- for 文は評価式内部を 3 ヶ所とも埋めること
- システムの遷移状態をコントロールするモジュールは関数 1 つにまとめること
- マスク処理が必要な場合は全て main 関数内に記述すること
- 標準関数はサポートしない
- ポインタ操作は基本的には解析対象外

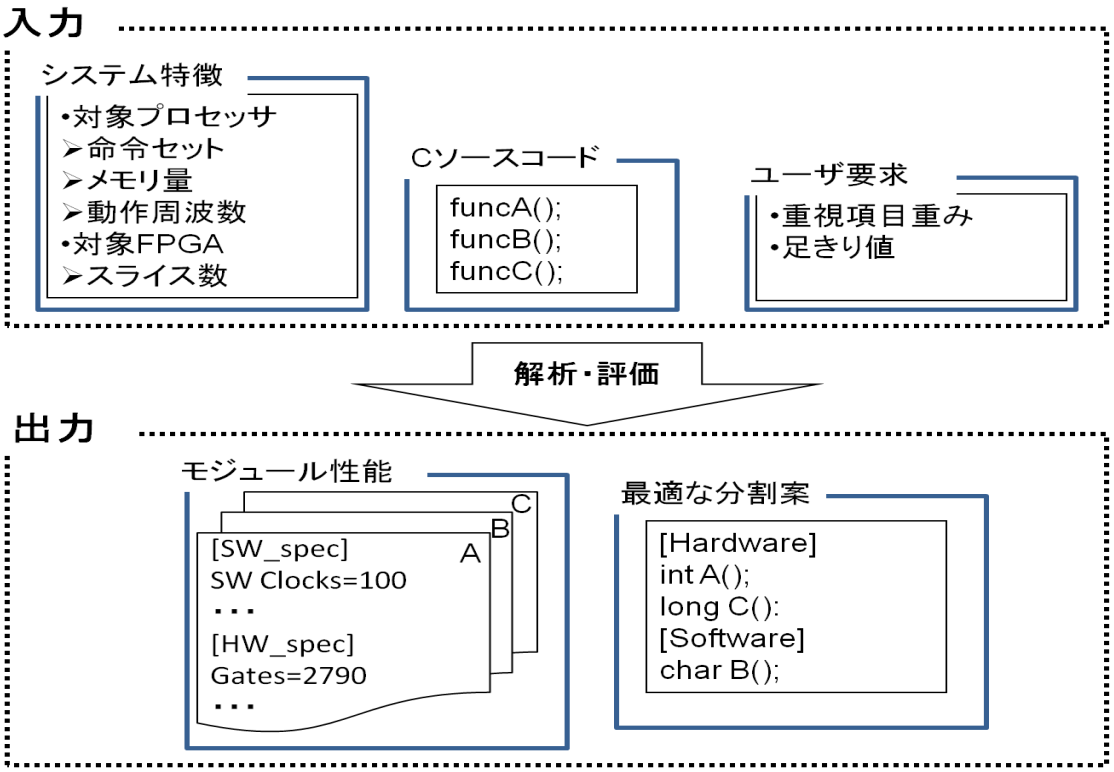

図 2　ハード／ソフト最適分割システムにおける入力と出力

2.4. MicroBlaze での検証方法

本システムの検証は Xilinx 社が提供するソフトコアプロセッサ“MicroBlaze”を中心として使用する。MicroBlaze とは FPGA 向けに構築されたソフトプロセッサコアである。MicroBlaze は MIPS に準拠した 32-bitCPU アーキテクチャになっている。3 段パイプラインアーキテクチャを採用しているため、ほとんどの命令が 1 クロックサイクルで実行することができる。パイプラインは 5 段まで拡張可能である。エンディアンはビッグエンディアン形式を使用している。また、浮動小数点ユニット・分岐命令高速化のための遅延スロットなどはオプションとして指定できるが本研究では付属しない。

MicroBlaze は Xilinx 社が提供する開発環境パッケージ EDK（Embedded Development Kit）を使用することで FPGA 上に構築可能である。EDK は XPS（Xilinx Platform Studio）・SDK という 2 つの環境で構築されており、このうち XPS を使用することでハードウェアの仕様を設定し FPGA 上に構築することが可能である。FPGA への論理合成、マッピング、配置配線などをするツールは XPS と協調関係のある Xilinx 社の ISE から呼び出して使用する。これらのプロセッサ・ツールを使用することでハードウェア／ソフトウェア協調動作が可能となる。

図 3 に本研究で使用する MicroBlaze による FPGA システムの構成を示す。MicroBlaze

に BlockRAM を接続し、これに命令とデータを格納する。BlockRAM の接続は専用の LMB（Local Memory　Bus）で接続可能である。BlockRAM のサイズは FPGA によって制限されることはあるが、最大で 64KB の領域を確保できる。ユーザが作成したハードウェアは FSL（Fast Simplex Link）、または OPB（On-chip Peripheral Bus）というバスを使用することで接続可能である。以下に各専用バスの役割を記述する。

- Local Memory Bus（LMB）

  LMB は MicroBlaze の BlockRAM にアクセスするためのバスである。バス幅は 32bit で、BlockRAM に 1 クロックサイクルでアクセスができる。

- First Simplex Link（FSL）

  FSL は MicroBlaze に直結したバスであり、ユーザが作成したハードウェアを接続可能である。入出力ともに FIFO で構成されており、入出力間でポイントツーポイント通信が可能である。データ幅は入出力ともに 32bit で、MicroBlaze では、入出力をセットとし、8 個までの FSL インタフェースが使用可能である。

- On-chip Peripheral Bus（OPB）

  MicroBlaze 用に用意されている周辺 IP コアを接続するための MicroBlaze メインバスであり多くの機能が提供されている。シリアル転送回路 UART や、外部メモリへのアクセスなどは OPB によって通信可能となる。また、ユーザが作成したハードウェアも GPIO や IPIF といった IP によって接続可能である。汎用的で扱いやすく使用できる IP も豊富だが、データ転送のみで最低でも 4 クロックサイクルが必要である。

本論文では、FPGA は Xilinx 社の Virtex-4 LX60 を対象に、検証システムを構成する。MicroBlaze の構成は標準のものを使用する。MicroBlaze に接続された BlockRAM 上に、ソフトウェア実行に必要な命令とデータを置き、ソフトウェア実行を可能とする。また、設計したハードウェアは FSL によって接続し、MicroBlaze から直接呼び出すことによってハードウェア実行を可能とする。ハードウェアの接続には OPB は使用しない。この構成の FPGA システムを構築することで、ハードウェア／ソフトウェア協調動作を実行する。OPB は計測結果を得るために使用する。OPB シリアル転送用回路 UART を接続し、RS232C ポートを通し PC ターミナルで動作の確認をする。

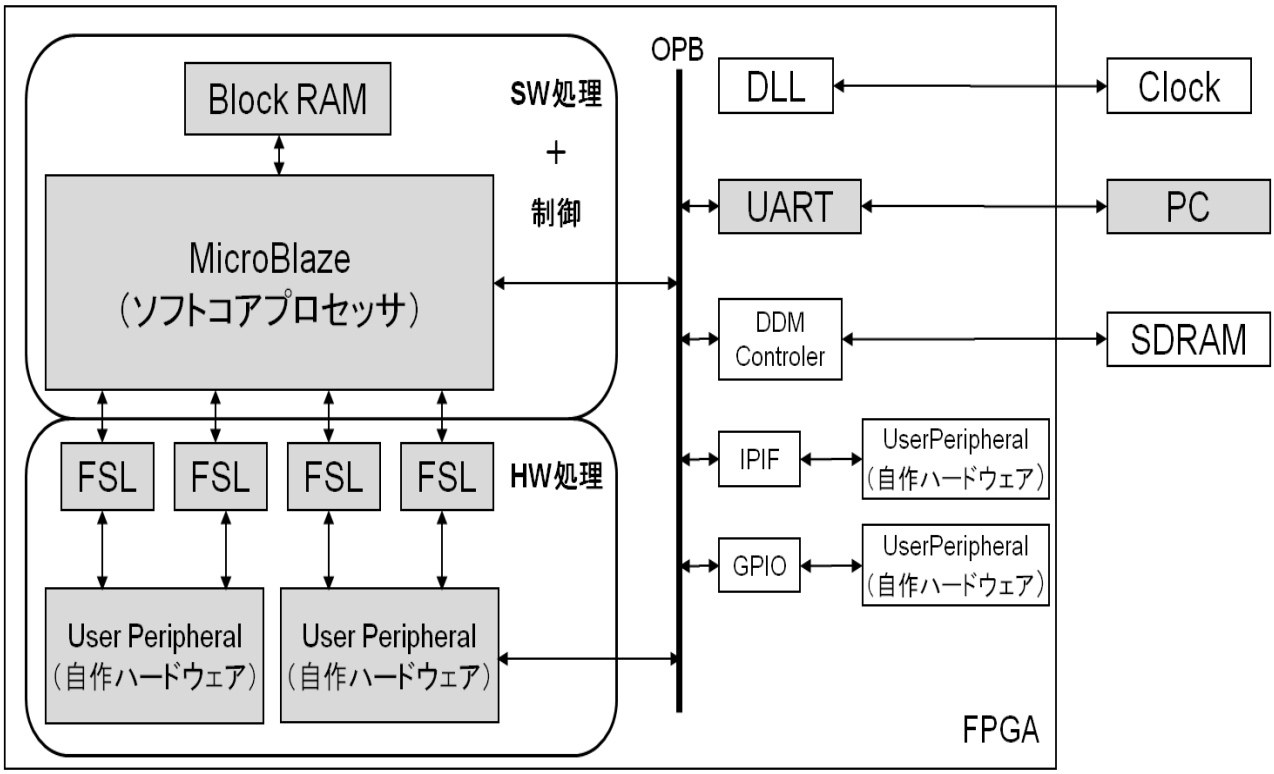

図 3　MicroBlaze による検証システムの構成

3. ハード／ソフト最適分割システムの実現

3.1. システムの構成

本章では本システムの詳細な実現方法について述べる。本システムは、オブジェクト指向スクリプト言語 Ruby により記述・実装する。Ruby を使用する理由は、Ruby は文字列操作が非常に強力であり、コード解析に適していること、オブジェクト指向に沿って設計することで、各オブジェクトの再利用性とシステムの拡張性を高めるためである。本システムの具体的な処理内容を UML クラスで表現したものを図 4 に示す。

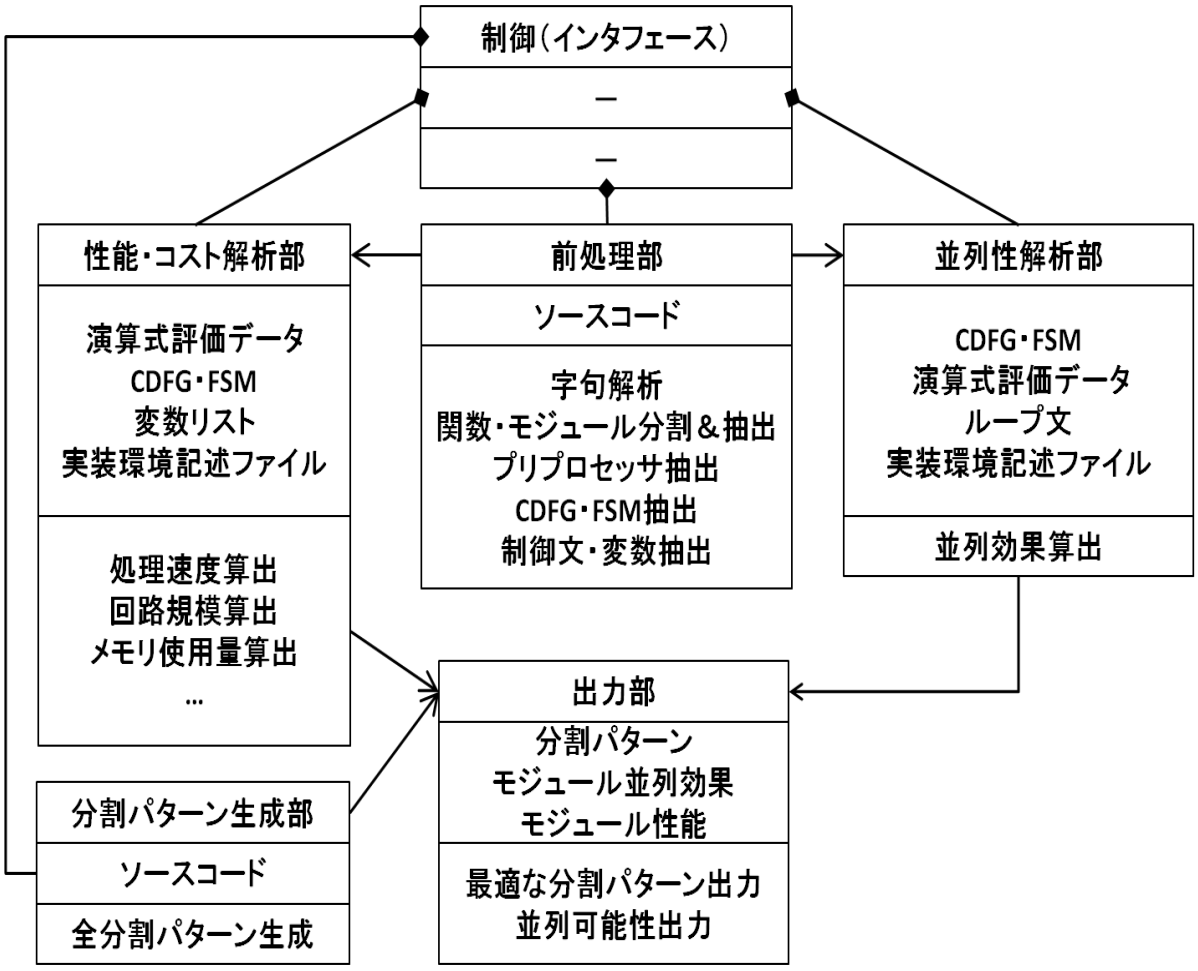

図 4　ハード／ソフト最適分割システムの UML クラス図

図 4 において本来属性を表す項目には入力を示している。本システムではまず、前処理部で関数・モジュール部分の抽出を行う。このとき、同時に変数、プリプロセッサ、CDFG（Control Data Flow Graph）、FSM（Finite State Machine）など解析に必要となる要素を抽出し、プログラム的な内部表現として保持しておく。前処理部より抜き出した情報を元に、性能・コスト解析部、並列性解析部で解析を行う。性能・コスト解析部では、ハー

ドウェアクロック数、ソフトウェアクロック数、回路規模、メモリ量、動作周波数、及び各モジュールの負荷割合を算出する。並列性解析部ではデータ並列性の検出を行い、適切な並列数も解析する。分割パターン生成部では全分割パターンの生成を行う。出力部では、解析された性能・コスト・並列性と全分割パターンを元に分割パターン毎の評価を行い、分割パターンの中で最も優れているものを最適な分割パターンとして出力する。

3.2. 前処理部

前処理部では、C ソースコードより、各解析に必要な情報を抽出する。前処理部で抜き出す情報と抜き出し方・表現方法を以下に示す。

(1)プリプロセッサ

プリプロセッサとは、ソースプログラムに対して行われる前処理部のことである。マクロ定義やヘッダファイル読み込みの指定がある。プリプロセッサ記述部を抽出し、マクロ定義は、検索文字列と置換文字列をシステムの内部情報として、ヘッダファイルの読み込みは、ヘッダファイル名とライブラリ名をハッシュ情報として抽出・保持する。

(2)関数・モジュール

本システムでは、関数・モジュール毎に性能や特徴の解析を行い、ハードウェアとソフトウェアどちらで実装することが望ましいか解析するため、関数とモジュールを抽出する。関数名をシステムの内部情報としてまとめると同時に、ソースコードそのものを抜き出し個別に解析できるように分割したコードを生成する。以後、本論文では、関数とはソフトウェアに対してのもの、モジュールはハードウェアに対してのものとして表記する。

(3)FSM・CDFG

コントロール・データ・フロー・グラフ（CDFG）はデータの流れをその制御とともにグラフ化したものである。システムの構造化分析においてよく出てくる手法であり、ハードウェアの高位合成などを行う動作合成ツールでは必ず使用される手法である。代表的な例を図 5 に示す。本研究においては、図 5 のように処理の分岐だけでなく、その命令の構造の深度と重みといった情報も付加している。ここでいう深度はプログラムの入れ子構造（ネスト）の階層であり、重みは文に対する処理負荷、または回数である。FSM（有限状態機械）とは有限個の状態と遷移と動作の組合せからなるふるまいのモデルである。本システムでは、ハードウェア設計でよく使用されるミーリ型オートマトンを使用する。ミーリ型オートマトンとは､内部状態および入力条件により出力と内部状態が決定する FSM である。なお、本システムで使用される FSM は決定性を持つ。本システムで使用される FSM の例を図 6 に示す。

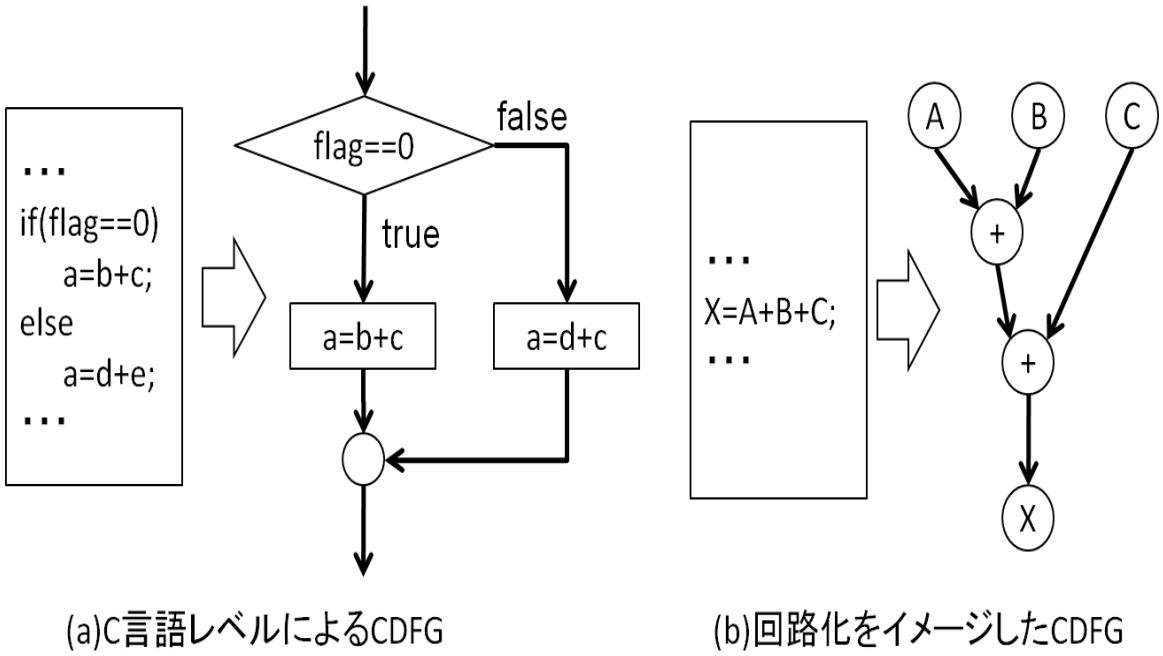

図 5　CDFG の例

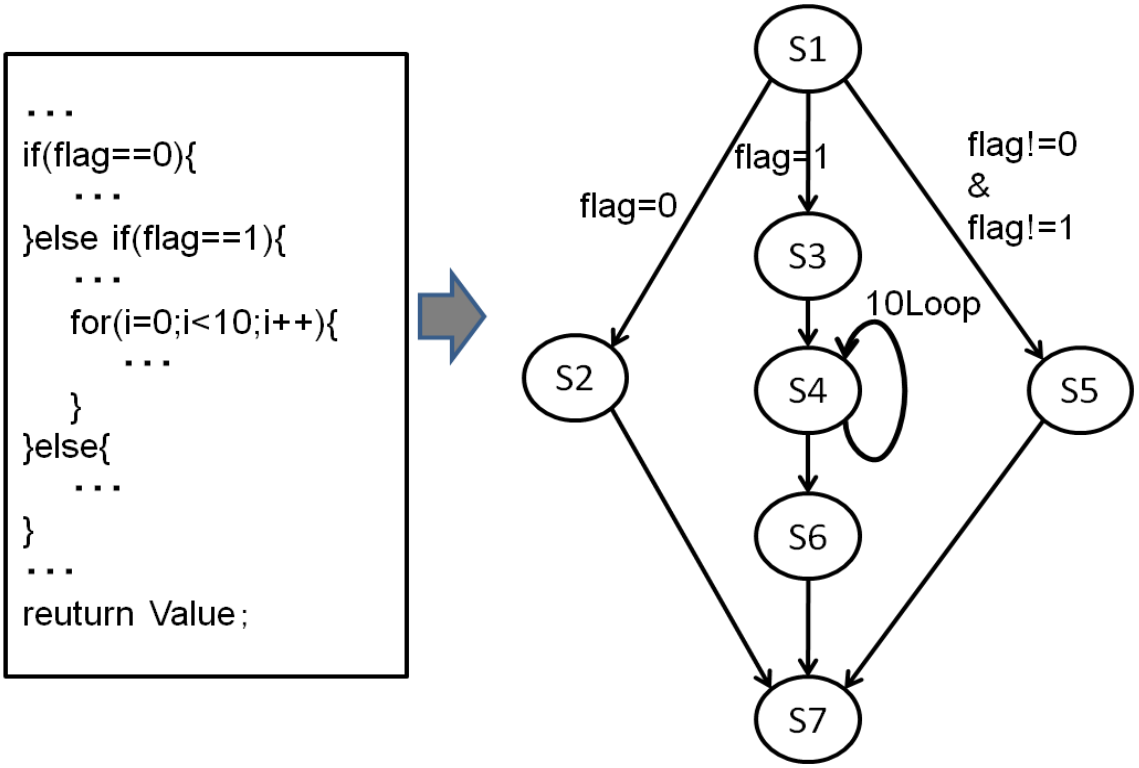

図 6　FSM の例

現在の本システムでは、FSM は、ループ文、条件文などにより状態が遷移する箇所のみ検出しシステムの内部情報として記憶している。また、ソースコードに状態を示すタグを

付加した解析系統を構築している。CDFG については、直接的な表現は生成せず、ソースコードを参照し直接構造解析し、FSM で生成した状態に対して、深度と処理負荷となる重みを加えている。これらは、演算子・命令リストの生成やハードウェアクロックサイクル数の算出に使用される。

(4)変数リスト

変数リストは、ソフトウェア実行時のメモリ量、ハードウェアの回路規模を算出するために必要になる。各関数・モジュールで使用される変数をリストにしたものである。宣言型、変数名、配列数リストにしてシステムの内部で表現される。

(5)演算子・命令リスト

演算子・命令の種類と数は、ソフトウェアクロックサイクル数を算出するために必要である。使用されている演算子・命令のリストを作成し、演算子・命令の種類と出現回数をシステムの内部情報としてまとめる。演算子・命令の出現回数は、ソースコード中の出現回数を算出したものと、実際の実行時の使用回数を算出したものをリストとしてまとめる。前者は、ハードウェア動作を考慮し、後者はソフトウェア動作を考慮したものである。ソースコード中の出現回数は単に演算子・命令をカウントしただけのものであるが、使用回数は、演算子をカウントする際に、CDFG などで決定された重みを掛け合わせることで算出している。

(6)左辺式評価データ

左辺式評価データとは、左辺を計算するまでに必要な演算子の数と変数のリストをリスト化したものである。データパスの遅延が最大となる箇所を検出するために使用する。

3.3. 性能・コスト解析部

性能・コスト解析部では、前処理部で抽出した情報を基に各関数・モジュールの性能・コストを算出する。算出する性能・コストは、ソフトウェアクロックサイクル数、ソフトウェア負荷割合、ハードウェアクロックサイクル数、回路規模、メモリ量、ハードウェア動作周波数である。ハードウェアに関する性能・コストは各設計者の設計能力・手法への依存度が高い。組み合せ回路か順序回路で設計するかの違いだけでもクロックサイクル数・回路規模などに生じる差は大きくなる。本システムでは、順序回路で必ず保持すべき値のみをレジスタとした場合に出来るデータパスを想定して算出する。各性能・コストの算出方法を以下に示す。

(1)ソフトウェアクロックサイクル数

ソフトウェアクロックサイクル数は、ソフトウェアで関数を 1 度実行した場合のクロッ

クサイクル数を算出する。演算子・命令リストの中で、ソースコードの実行時の使用回数を算出したリストに、実装環境のプロセッサの命令セットに合わせて決定した重みを掛け合わせ、全ての総和をとることで算出する。重みはシステムの特徴を記述したファイルに記述しておき、そこから抜き出す。現在、本システムは MicroBlaze にのみ対応している。MicroBlaze における演算命令に対する重みを表 1 に示す。

表 1　MicroBlaze での各演算に対する重み

演算子種類	重み
論理演算	1
四則演算（除算・剰余算以外）	1
除算・剰余算	32
分岐	2
型変換	5
Break	3
関数参照	2
配列インデックス計算	2
右シフト	シフト量
左シフト	1
代入命令	1

(2)ソフトウェア負荷割合

ソフトウェア負荷割合は、ソフトウェア実行時に各関数が占める負荷の割合である。C ソースコード中の main 文を解析し、各関数の使用頻度を算出する。使用頻度と、算出されたソフトウェアクロックサイクル数を掛け合わせ、各関数がシステム全体で実行するクロックサイクル数を算出し、さらに、システム全体での割合を求め、それをソフトウェア負荷割合とする。

(3)ハードウェアクロックサイクル数

レジスタとして必ず保持すべき値として考えられるものは、ループ文内の変数とポインタで指示した先のアドレスに置かれている値である。前処理で作成した FSM・CDFG 表現より、状態がループしている箇所を検出し、ループ文のループ回数を評価することでハードウェアクロックサイクル数を算出する。現段階ではポインタ表現は制限しており、未実装であるが、算出したハードウェアクロックサイクル数にポインタ変数の出現回数を加えることで、より詳細なハードウェアクロック数が検出できるであろうと考えられる。

(4)回路規模

回路規模は、FPGA スライス数を算出する。FPGA スライス数は、データパス中の演算器の数と、レジスタ数を考慮すべきである。前処理部で得た演算子・命令リスト、変数リストなどを利用し、演算器の数とレジスタになり得る部分を検出し、その総和をとり対象の FPGA に対応した重みを掛け合わせることで算出する。本研究で実験する環境“Virtex4 LX60”における演算器・レジスタへの回路規模を表 2 に示す。なお表 2 における重みは、演算器は 32bit 出力し、レジスタは 32bit の場合のスライス数である。

表 2　Virtex4 LX60 での各演算子に対する回路規模

種類	論理演算	加算・減算	レジスタ
スライス数(slices)	18	16	16

(5)メモリ量

ソフトウェアの実行時に使用するメモリ量と、実行コードをプロセッサに置くためのメモリ量の算出を行う。変数リスト内の全ての変数の bit 総和からソフトウェアの実行時に使用するメモリ量の算出を行う。また、実行コードをプロセッサに置くためのメモリ量は、演算子・命令リストより命令数を算出し、命令数にプロセッサの命令コードのビット長を掛け合わせ算出する。

(6)ハードウェア動作周波数

前処理部で得た左辺式評価データを使用し、データパスが最長になる箇所で最終出力までに使用した演算器の種類と数を算出し、実際に FPGA ボード上で測定した重みと掛け合わせ総和をとることで最大遅延時間を算出する。総和には、どんなハードウェアにも付与される IOBUF のレイテンシも加える。その遅延速度の逆数を算出することでハードウェア動作周波数を算出する。本研究で対象とする FPGA の各演算子に対する遅延時間を表 3 に示す。

表 3　Virtex4 LX60 での各演算子に対する遅延時間

演算子	論理演算	加算・減算	IOBUF
レイテンシ(ns)	0.6	2.6	5.5

3.4. 並列性解析部

並列性解析部では、マルチコア環境を考慮し、各関数・モジュールの並列性を解析する。並列性にはデータ並列性とタスク並列性が挙げられるが、本システムではデータ並列性の

解析を行う。データ並列とは、処理されるデータ領域を分割し、分割されたデータ領域に対し同一の処理を行う並列手法である。プログラムのループが本質的に備えている並列性であり、ループの各周回が各ノードで並列に処理できるようにデータを配布するかが中心となる。よって、データ並列性は大きなデータ構造に対してループ処理を繰り返す処理に見ることができる。

本システムでは、ループ文とデータ依存性を検出することで並列性の有無を検出する。データ依存性がある場合はデータ並列性がないと判断する。データ依存性がある例を図 7 に示す。図 7 に示すように、先の文で得た出力を後の文の入力に使用する場合、先の文に必要な入力値が後の文に書き換えられる場合、及び同じ変数に対して複数回の書き換えをするときの 3 パターンがある。このうちデータにループ伝搬依存が見られる A・B の場合のデータ依存性が検出されなかった場合、データ並列性を検出する。

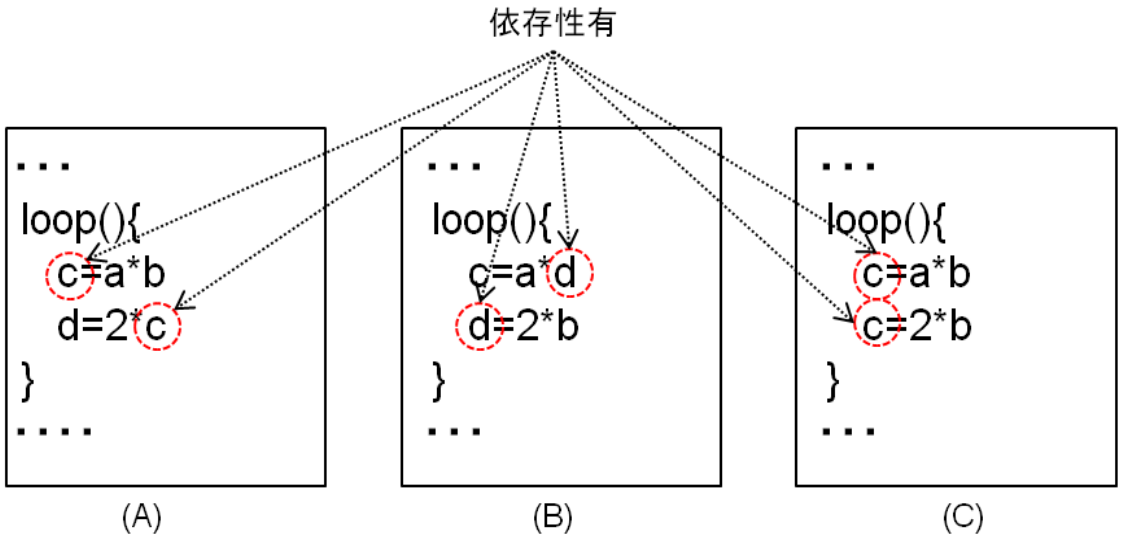

図 7　データ依存性が検出されるパターン

データ並列性が検出できた場合、分割できるデータ構造を持っているかをデータが配列かどうかから判断し、性能・コスト解析で得たソフトウェアクロックサイクル数より並列効果を算出する。

3.5. 分割パターン生成部・出力部

分割パターン生成部では、前処理部より抜き出した関数・モジュールを利用し、全分割パターンの生成を行う。

出力部では、システムが判断した最適な分割パターンとその性能・コスト、各関数・モジュールの性能・コストを出力する。

最適な分割パターンの出力は、まず、分割パターン生成部で作成した全パターンと、性能・コスト解析部で算出した性能・コストを組み合わせ、分割パターン毎にシステム全体

の性能・コストを算出する。このとき、性能・コストに対して足きり値が設けられている場合、足きり判定を行い、該当するパターンをリスト化しておく。その結果より、システム全体の評価値の算出を行い、足きりされていないパターンの中で評価値が最も優れているものを最適な分割パターンとして出力する。評価は、速度重視、回路規模重視、バランス重視、ユーザ要求重視の 4 つの評価を行い出力する。速度重視ではクロックサイクル数が最小のものを、回路規模重視ではスライス数が最小のものを、バランス重視・ユーザ要求重視は、評価式を用意し、最も評価が良かったものを最適なパターンとする。評価式は式(2)に示すものを使用する。Eval は各パターンの評価値であり、数値が大きいほど優先度が高くなる。W は性能・コスト項目に対する重みであり W は全体で 1 になるように重みを付ける。Val は各性能の偏差値である。worst は全パターン中の最悪値を、best は全パターン中の最良値を、this は評価するパターンの値を示す。バランス重視では重みをクロック数・回路規模ともに 0.5 にして計算する。

$$Eval = \sum \left[W \times \frac{Val_{worst} - Val_{this}}{Val_{worst} - Val_{best}} \right]_{pattern} \cdots \text{式(2)}$$

3.6. 制御部

制御部では、主に本システム内の各クラス・メソッドを使用することで処理フローの制御を行う。関数・モジュール毎にオブジェクトを生成し、各メソッドに受け渡すことで処理を行う。

補助的な機能として、C 言語におけるコメント文の削除、マクロ定義の置き換え、解析時に生成されるディレクトリやファイルを管理する機能を実装している。

4. ハード／ソフト最適分割システムの適用と評価

4.1. Misty1 への適用

4.1.1. Misty1 暗号アルゴリズム

本システムを Misty1 暗号へ適用した。Mity1 暗号は 128bit の暗号化鍵をもつ 64bit ブロック暗号アルゴリズムである[10][11]。Misty1 暗号の処理は、主に、暗号処理を行うデータランダマイズ部と、秘密鍵から拡大鍵を導き出す、鍵スケジューリング部に分かれる。図 8 にデータランダマイズ部の処理フローを、図 9 に図 8 中で使用される副関数の処理フローを、図 10 に鍵スケジューリング部の処理フローを示す。

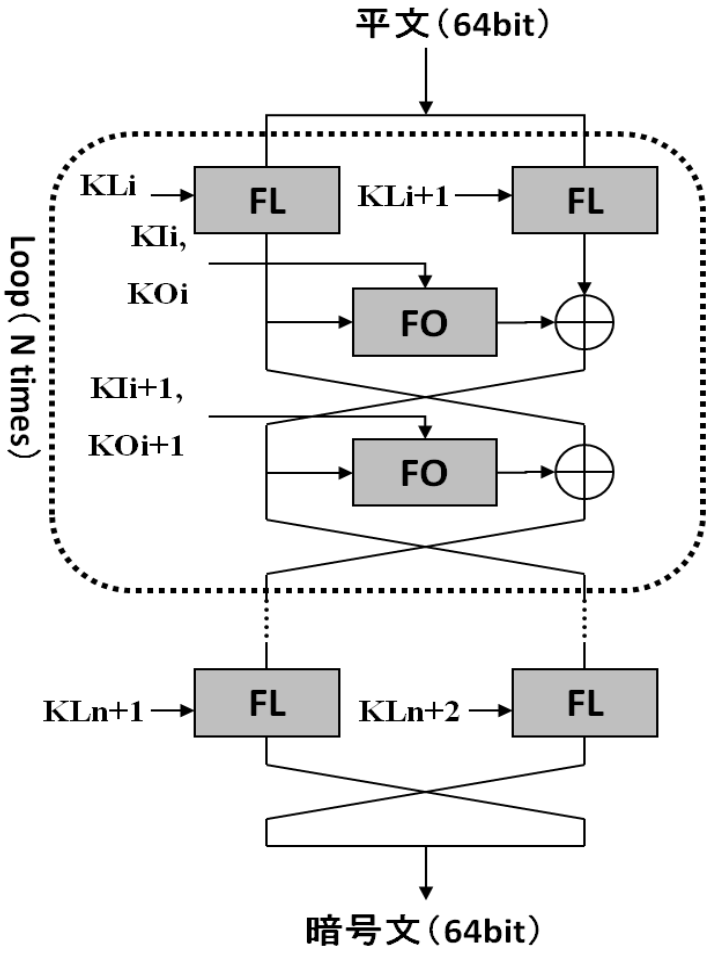

図 8　Misty1 暗号データランダマイズ部処理フロー

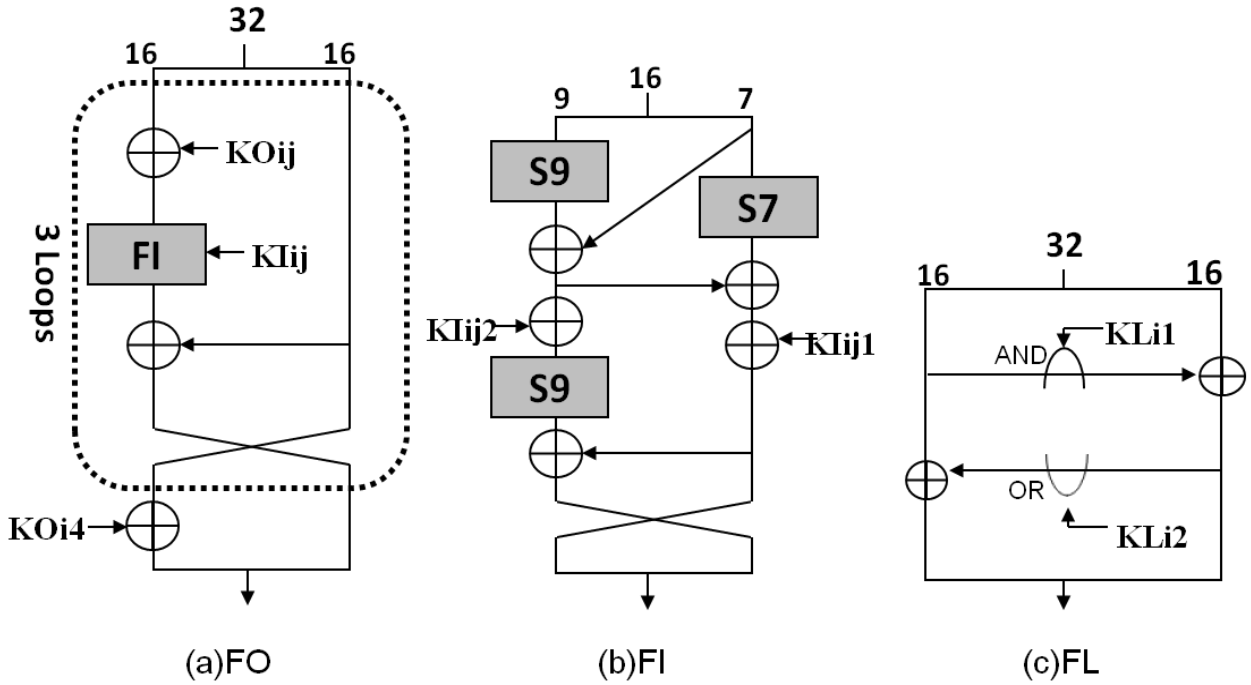

図 9　各副関数の処理フロー

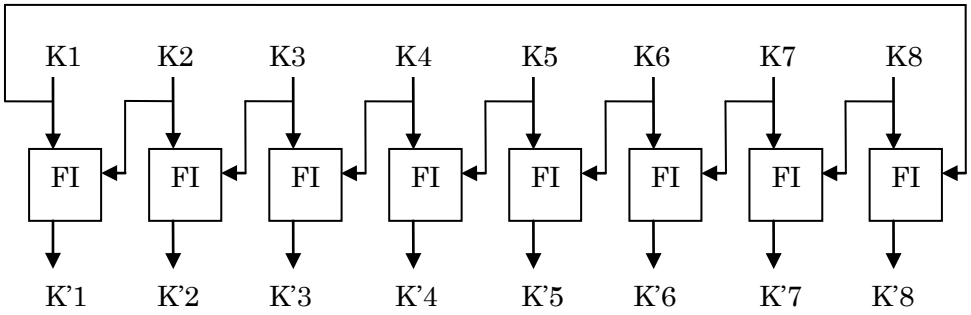

図 10　Misty1 暗号鍵スケジューリング部の処理フロー

表 4　Misty1 暗号の鍵の対応表

	KOi1	KOi2	KOi3	KOi4	KIi1	KIi2	KIi3	KLi1	KLi2
Key	K_i	K_i	K_{i+7}	K_{i+4}	K'_{i+5}	K'_{i+1}	K'_{i+3}	(odd)$K_{(i+1)/2}$ (even)$K'_{i/2+2}$	(odd)$K'_{(i+1)/2+6}$ (even)$K_{i/2+4}$

データランダマイズ部はループ構造を持っており、ループ回数は 2 の倍数をとる限りで可変である。推奨値は 4 ループである。入力データ 64bit を 32bit に二分割し、排他的論理和と副関数 FL、FO の処理を繰り返すことで変換を行う。副関数 FO では、データ分割、排他的論理和、及び副関数 FI によって変換を行う。ループ構造を持っており、3 回同じ処

理をした後、処理を終了する。FO 中の FI は、データ分割、排他的論理和、副関数 S9・S7 によって変換を行う。S9・S7 は置換表に基づくテーブル演算処理である。副関数 FL では、データ分割した後、排他的論理和、論理和、論理積を行うことでデータ変換を行う。

鍵スケジューリング部（KS）では、入力された 128bit の秘密鍵を 16bit ずつに分割し、それぞれを副関数 FI によってデータ変換することで拡大鍵を生成する。

秘密鍵と拡大鍵はそれぞれ、16bit ずつに分割し、左から i 番目のデータをそれぞれ Ki（1≦i≦8）とし、鍵の対応表に基づきデータランダマイズ部へ割りつけられる。i が 8 を超える場合、i は 8 の剰余となる。表 4 に Misty1 暗号における鍵の割付時の対応表を示す。KOi1、KOi2、KOi3 はそれぞれ FO 内の KOij にループ回数に対応して割りつけられる。KOi4 は FO 内の KOi4 に割りつけられる。KIi1、KIi2、KIi3 は FO 内の KIij にループ回数に対応して割りつけられる。KIij は FI 内では左 7bit、右 9bit に分割され、それぞれ KIij1、KIij2 として使用される。KLi1、KLi2 は FL 内の KLi1、KLi2 に割りつけられる。KLi1、KLi2 はループ回数が奇数のとき odd の値が、偶数のとき even の値が使用される。

4.1.2. 適用結果

段数が 8 の場合の Misty1 暗号を FL・FO・FI・KS（鍵スケジューリング部）の 4 つの関数・モジュールに分割し、本システムに適用した。表 5、表 6、及び表 7 に Misty1 暗号における本システムの解析結果と、図 3 のシステムで計測した実測値を示す。回路規模、動作周波数などのハードウェアに関する性能は、MicroBlaze は使用せず、MicroBlaze と協調関係のある ISE9.2i による測定結果とそれを元に算出したものである。表 5 は本システムによる Misty1 暗号の各関数・モジュールの性能である。表 6 は本システムによる Misty1 暗号の全分割パターンの性能、及び評価値である。表 7 は、Misty1 暗号に対して本システムが出力した最適な分割案のパターンと、そのときの Misty1 暗号システムの処理性能、及び評価値を示している。表 6 と表 7 中の H と S はハードウェアとソフトウェアである。

表 5 に示す各関数・モジュールにおける性能・コストの解析値と実測値の比較を行うと、実測値を 1 とした場合の解析値の比率の平均は 1.1 であり、整合性は高い。並列性は全ての関数・モジュールに並列性がないと出力された。Misty1 はデータを分割した処理をするが、分割されたデータ間に依存関係がある。また、大きなデータ構造を持っているというわけではなく、入力から出力までひとつのデータに関して処理を行うため分割は難しい。そのため、この結果は妥当であるといえる。

表 6 に示す解析結果は表 5 の結果を元に解析されている。速度重視、回路規模重視共に KS がソフトウェアのとき全ての評価値が高い。また、バランス重視では KS がソフトウェアで、FI がハードウェアのときのパターンの評価値が全体的によい結果となっている。表 5 より、KS は負荷割合が低く、FI は負荷割合が高いことがわかる。KS は全ての処理の中で一度しか使用されず、ハードウェアにしたときの効果も高くはない。FI はハードウェアにしたときの回路規模の増加量あたりの SW 実行サイクル数の減少量が一番多い。これより、

この結果は妥当であると考えられる。また、FO だけに注目してみると、全体的に FO がハードウェアのときのパターンの評価がよい。FO は FI を 3 つ含んでおり、FO 中の FI の処理の割合も多いため、ハードウェア化の効果は高いといえる。

表 7 に示す分割案は本システムの最終出力であり、表 6 の結果を元に解析されている。速度重視のパターンはハードウェアのみ、回路規模重視のパターンはソフトウェアのみとなっており妥当な結果となっている。また、バランス重視のパターンは、KS のみがソフトウェア、他の 3 つがハードウェアパターンとなっている。KS は処理中に一度しか出てこないので、ハードウェア化をした場合のハードウェア資源の使用頻度が低いことがいえる。また、FI は全処理中で最も使用頻度が高く、処理負荷も高くなっている。そのため、FI と、FI を包含する FO を優先的に、ハードウェア化することが望ましいと考えられる。FL について、処理負荷は低いが、ハードウェア化の効果が他の関数・モジュールに比べ非常に高いため、ハードウェア化が望ましいと考えられる。これらのことより、KS のみがソフトウェアで、他の 3 つがハードウェアのパターンが最もコストパフォーマンスがよく、バランス重視のパターンとして選出されていることは妥当であると考えられる。

表 5　本システムによる Misty1 暗号の各関数・モジュールの性能

	FI		FO		FL		KS	
	解析値	実測値	解析値	実測値	解析値	実測値	解析値	実測値
SW 実行サイクル数[Cycles]	47	36	247	199	48	41	576	551
SW 負荷割合[%]	45	46	47	52	10	5	12	17
HW 実行サイクル数[Cycles]	1	1	4	3	1	1	12	8
回路規模[Slices]	256	232	784	1026	16	16	2064	2489
メモリ量[Bytes]	28	39	140	43	24	36	232	228
最高動作周波数[MHz]	88	63	20	34	108	155	35	59.6
並列性	なし	—	なし	—	なし	—	なし	—

表 6　本システムによる Misty1 暗号の全分割パターンの解析結果

パターン					評価項目				
パターン番号	FI	FO	FL	KS	回路規模 [slices]	実行サイクル数 [cycles]	評価値		
							速度	回路規模	バランス
1	S	S	S	S	0	4841	0.000	1	0.500
2	S	S	S	H	2064	4253	0.184	0.338	0.253
3	S	S	H	S	16	4371	0.158	0.995	0.579
4	S	S	H	H	2080	3783	0.316	0.333	0.319
5	S	H	S	S	784	3842	0.316	0.749	0.533
6	S	H	S	H	2848	3254	0.474	0.087	0.273
7	S	H	H	S	800	3372	0.447	0.744	0.599
8	S	H	H	H	2864	2784	0.605	0.082	0.338
9	H	S	S	S	256	3575	0.395	0.918	0.662
10	H	S	S	H	2320	2987	0.553	0.256	0.401
11	H	S	H	S	272	3105	0.526	0.913	0.727
12	H	S	H	H	2336	2517	0.684	0.251	0.467
13	H	H	S	S	1040	2576	0.684	0.667	0.681
14	H	H	S	H	3104	1988	0.842	0.005	0.421
15	H	H	H	S	1056	2106	0.816	0.662	0.747
16	H	H	H	H	3120	1518	1.000	0.000	0.500

表 7　本システムによる Misty1 暗号の分割案

重視項目	FI	FO	FL	KS	回路規模 [Slices]		実行サイクル数[Cycles]		評価値		
					解析値	実測値	解析値	実測値	速度	回路規模	バランス
速度	H	H	H	H	3120	3436	1518	1467	1	0	0.5
回路規模	S	S	S	S	0	0	4841	3918	0	1	0.5
バランス	H	H	H	S	1056	1093	2106	1782	0.81	0.66	0.75

次にユーザ要求を考慮した解析結果を表 8 に示す。ここでは例として、回路規模が 3000slices 以下、実行サイクル数が 4500clock 以下、システムの動作周波数が 30MHz 以上、メモリが 10KByte 以下という条件・制約と、評価での重みづけが回路規模 0.1、速度 0.9 というユーザ要求を指定している。表 8 における速度・回路規模・バランスは、上記の条件・制約を満たす解析であり、重みづけはこれまでと同様である。

表 8　ユーザ要求を考慮した本システムの最適な分割パターン

重視項目	FI	FO	FL	KS	回路規模[Slices]	実行サイクル数[Cycles]	最高動作周波数[MHz]	メモリ量[KByte]	評価値
速度	H	S	H	H	2336	2517	88	1.502	0.64079
回路規模	S	S	H	S	16	4371	35	1.736	0.24211
バランス	H	S	H	S	272	3105	35	1.736	0.56654
ユーザ要求	H	S	H	H	2336	2517	88	1.502	0.64079

ユーザ要求を考慮し条件・制約が増えたことで、表 7 の出力結果と比べ、速度重視、回路規模重視、バランス重視の分割案の出力が変化した。ユーザ要求による条件・制約は全て満たしており、表 6 の結果と照合し確認してゆくと、速度重視、回路規模重視、バランス重視では、制限内で実行サイクル数が最小のもの、回路規模が最小のもの、評価値が最大のものがそれぞれ選ばれている。ユーザ要求重視では、速度に大きな重みがおかれているため、実行サイクル数が最小のものが選ばれている。今回の場合は、動作周波数が 30MHz 以上必要と設定したため、FO をハードウェア化するパターンが条件・制約を満たしていなおらず評価されていない。しかし、実測値では、34MHz と 30MHz 以上の条件に適合しているため評価されるべきである。ユーザ要求を考慮する場合、解析精度が重要であり、解析性能の向上が課題となる。

動作周波数の制限をなくし、FO を含めたハード／ソフト最適分割の結果を表 9 に示す。表 8 の結果よりも表 9 の方が本来設定した制限下での最適な分割パターンに近いこととなる。速度重視、バランス重視、ユーザ要求重視では KS のみソフトウェアでの実装がよいと解析され、回路規模重視では、FL のみをハードウェア化することが望ましいと解析された。これより KS のみソフトウェアのパターンが優秀だということがわかる。

表 9　動作周波数を考慮しない場合の分割パターン

重視項目	FI	FO	FL	KS	回路規模[Slices]	実行サイクル数[Cycles]	評価値
速度	H	H	H	S	1056	2106	0.80207
回路規模	S	S	H	S	16	4371	0.31635
バランス	H	H	H	S	1056	2106	0.80207
ユーザ要求	H	H	H	S	1056	2106	0.80207

4.2. AES への適用

4.2.1. AES 暗号アルゴリズム

本システムを AES 暗号へ適用した、AES とはアメリカ合衆国の新暗号規格（Advanced Encryption Standard）として規格化された共通暗号方式である。AES 暗号は Rijndael アルゴリズムを採用しており[26]、入力データに一定回数のデータ変換を行うことで暗号化す

る。暗号化ブロック長は 128bit であり、鍵長は、128bit・192bit・256bit の 3 種類のビット長のものが利用できる。

AddRoundKey 変換、SubBytes 変換、ShiftRows 変換、MixColumns 変換の 4 種類のデータ変換があり、これに鍵拡張部の KeyExpantion 変換が加わり、大きく 5 つの機能ブロックに分けることができる。図 11 に AES 暗号全体の処理フローを示す。鍵長によってループ数が決定するが、本研究で 128bit 鍵長のもので実験を行う。128bit 鍵長のとき、図 11 のループ数は 10 となる。

AES 暗号のデータの扱い方は 1 バイトを 1 ブロックとして 4 つのブロックを繋げて 1 列とした 2 次元配列上のデータを処理する。データの並べ方を図 12 に示す。AES ではほとんどの場合が列単位で処理を行う。

KeyExpantion 変換は、暗号化に使用する鍵を拡張する処理である。AddRoundKey 変換は入力データと鍵データの排他的論理和をとる。SubByte 変換は、2 の 8 乗のガロア体の逆元計算と行列計算を行う。数学的には複雑な演算が必要となるが、入力によって出力が一意に決まるため、ハッシュテーブルを用意しておけば、テーブル参照をすることで実現可能になるため、この方法で実現している。ShiftRows 変換は入力データを行ごとにシフトする。ただし、シフト量は行ごとに違う。MixColumus 変換は、入力データを列単位で、ガロアフィールド理論に基づく演算によって変換する。

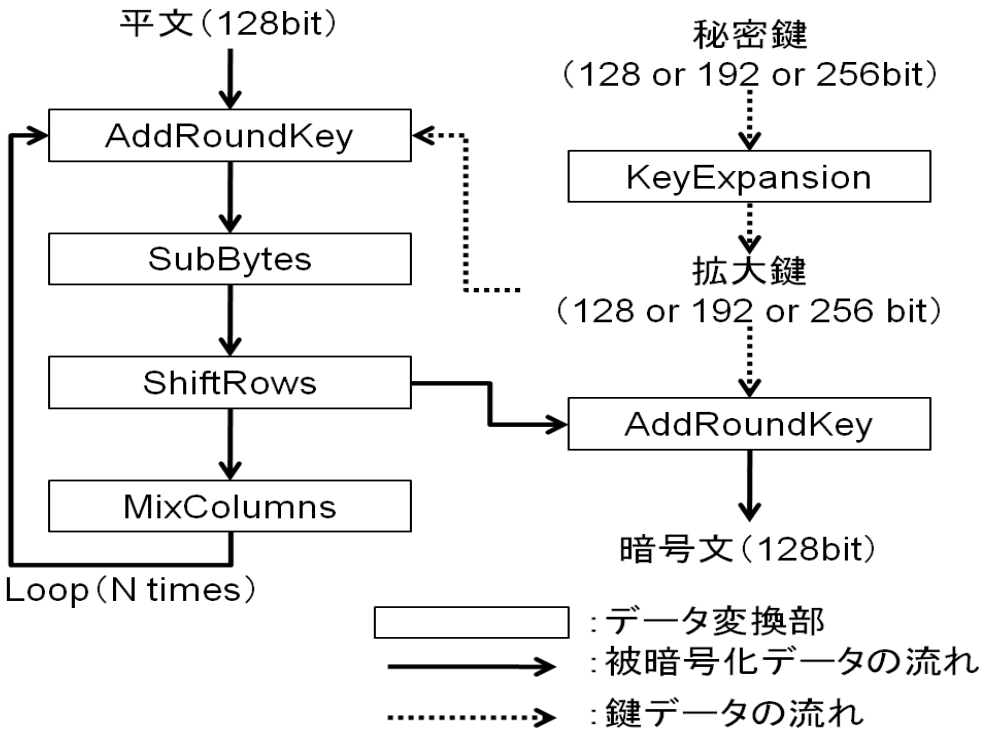

図 11　AES 暗号の処理フロー

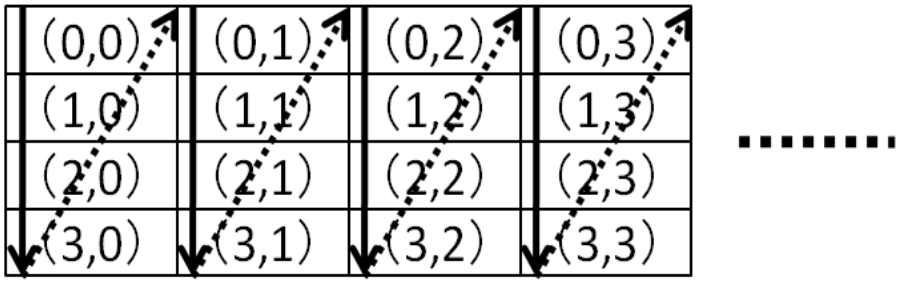

図 12　AES でのデータの並び

4.2.2.　適用結果

鍵が 128bit 長の AES を、KeyExpansion（KE）、AddRound（Add）、SubBytes（Sub）、ShiftRows（Shift）、MixColumns（Mix）の 5 つの関数・モジュールに分割し、本システムを適用した。表 10、表 11、及び表 12 に AES 暗号における本システムの解析結果を示す。表 10 では本システムによる AES 暗号の各関数・モジュールの性能を解析値と、図 3 のシステムで計測した実測値を示す。回路規模、動作周波数などのハードウェアに関する性能は、MicroBlaze は使用せず、MicroBlze と協調関係のある ISE9.2i による測定結果とそれを元に算出したものである。表 11 は本システムによる AES 暗号の全分割パターンの性能、及び評価値である。表 12 は AES 暗号に対して本システムが出力した最適な分割案のパターンと、そのときの AES 暗号システムの処理性能、及び評価値を示している。表 11、表 12 中の H と S はハードウェアとソフトウェアを表す。

表 10 に示す各関数・モジュールにおける性能・コストの解析値と実測値の比較を行うと、解析値を 1 とした場合の解析値の比率の平均は 1.4 であるが、範囲は 0.3～3.5 となり、やや精度に問題がある。これは、KeyExpansion、AddRoundKey、ShiftRows の回路規模、KeyExpansion、ShiftRows の最高動作周波数が原因となっている。回路規模については、C ソースコードの段階での設計思想と実際の設計が異なっているためである。しかし、全体を通した数値の揺れ幅だけを見ると KeyExpansion 以外の関数・モジュールに関しては微差であるといえるため、今回の解析には問題ない。最高動作周波数は、回路規模で挙げられる理由に加え、ハードウェア化する箇所以外の演算部もハードウェア化するとみなし、動作周波数の演算に含めているためである。回路規模は最終的な分割案の性能結果に影響し、最高動作周波数は足きりに関する項目であるため、ユーザ要求で足きり値を設けた場合に影響する。これより、ユーザ要求を考慮しない場合の解析結果は妥当であるといえる。並列性について、KeyExpansion に並列性があると判断されている。KeyExpansion はデータの依存がなく、大きなデータ構造を使用した演算が行われる。また、全く同じ構造の文が複数連続で記述されている。このため、データ並列による並列実行が可能である。他の

関数・モジュールについてはデータ依存が見られるため並列実行は不可能である。これより、並列性の解析においても、解析結果は妥当であるといえる。

表 11 の解析結果は、表 10 の結果を元に解析されたものである。速度重視では、表 10 の結果で SW 負荷割合が高かった MixColumns と ShiftRows をハードウェア化するパターンの評価が高い。回路規模重視では表 10 で回路規模が小さな AddRoundKey と ShiftRows がハードウェアのパターンの評価が高い。速度重視、回路規模重視ともに、KeyExpansion と SubBytes がハードウェアのパターンの評価は低い。また、バランス重視において全分割パターンの評価値が高いもののパターン番号を順に 5 つ挙げると、15、7、13、31、5 になる。これらには、KeyExpansion と SubBytes がソフトウェアのパターンであるという共通点が挙げられる。また、逆に評価値が低いものからパターン番号を 5 つ挙げると、18、26、20、2、28 となる。これらには、KeyExpansion と SubBytes がハードウェアのパターンである。これより、KeyExpansion と SubBytes 以外の関数・モジュールを優先的にハードウェア化することが効果的であることがわかる。

表 10　本システムによる AES 暗号の各関数・モジュールの性能

	KE		Add		Shift		Mix		Sub	
	解析値	実測値	解析値	実測値	解析値	実測値	解析値	実測値	解析値	実測値
SW 実行サイクル数[Cycles]	2910	2181	256	228	816	1178	912	1260	288	204
SW 負荷割合[%]	11	7	11	7	32	38	32	41	11	7
HW 実行サイクル数[Cycles]	60	44	16	16	16	16	96	16	16	16
回路規模[Slices]	714	334	64	18	64	0	384	324	1088	924
メモリ量[Bytes]	52	39.5	24	14	40	53	108	68	24	8.5
最高動作周波数[MHz]	36	103	181	165	181	108	8	140	181	107
並列性	あり	—	なし	—	なし	—	なし	—	なし	—

表 11　本システムによる AES 暗号の全分割パターンの解析結果

パターン番号	KE	Add	Shift	Mix	Sub	回路規模 [slices]	実行サイクル数[cycles]	評価値		
								速度	回路規模	バランス
1	S	S	S	S	S	0	25269	0	1	0.500
2	S	S	S	S	H	1088	22549	0.116	0.530	0.327
3	S	S	S	H	S	384	17925	0.313	0.834	0.574
4	S	S	S	H	H	1472	15205	0.428	0.364	0.389
5	S	S	H	S	S	64	17269	0.341	0.972	0.660
6	S	S	H	S	H	1152	14549	0.456	0.502	0.475
7	S	S	H	H	S	448	9925	0.653	0.806	0.722
8	S	S	H	H	H	1536	7205	0.769	0.336	0.549
9	S	H	S	S	S	64	22629	0.112	0.972	0.548
10	S	H	S	S	H	1152	19909	0.228	0.502	0.363
11	S	H	S	H	S	448	15285	0.425	0.806	0.610
12	S	H	S	H	H	1536	12565	0.541	0.336	0.437
13	S	H	H	S	S	128	14629	0.453	0.945	0.696
14	S	H	H	S	H	1216	11909	0.569	0.475	0.523
15	S	H	H	H	S	512	7285	0.765	0.779	0.770
16	S	H	H	H	H	1600	4565	0.881	0.309	0.585
17	H	S	S	S	S	714	22479	0.119	0.691	0.415
18	H	S	S	S	H	1802	19759	0.235	0.221	0.230
19	H	S	S	H	S	1098	15135	0.431	0.525	0.477
20	H	S	S	H	H	2186	12415	0.547	0.055	0.304
21	H	S	H	S	S	778	14479	0.459	0.664	0.563
22	H	S	H	S	H	1866	11759	0.575	0.194	0.390
23	H	S	H	H	S	1162	7135	0.772	0.498	0.638
24	H	S	H	H	H	2250	4415	0.888	0.028	0.452
25	H	H	S	S	S	778	19839	0.231	0.664	0.451
26	H	H	S	S	H	1866	17119	0.347	0.194	0.278
27	H	H	S	H	S	1162	12495	0.544	0.498	0.525
28	H	H	S	H	H	2250	9775	0.659	0.028	0.340
29	H	H	H	S	S	842	11839	0.572	0.636	0.611
30	H	H	H	S	H	1930	9119	0.687	0.166	0.426
31	H	H	H	H	S	1226	4495	0.884	0.470	0.673
32	H	H	H	H	H	2314	1775	1	0	0.500

表 12　本システムによる AES 暗号の分割案

重視項目	KE	Add	Shift	Mix	Sub	回路規模 [Slices]		実行サイクル数[Cycles]		評価値		
						解析値	実測値	解析値	実測値	速度	回路規模	バランス
速度	H	H	H	H	H	2314	1600	1775	1397	1	0	0.5
回路規模	S	S	S	S	S	0	0	25269	31246	0	1	0.5
バランス	S	H	H	H	S	512	1266	7285	5618	0.765	0.779	0.770

表 12 に示す解析結果は、本システムの最終結果であり、表 11 の結果を元に解析されたものである。速度重視・回路重視ではそれぞれ、ハードウェアのみのパターンとソフトウェアのみのパターンを算出しており妥当である。バランス重視では、KeyExpansion と SubByte がソフトウェアとなっている。KeyExpansion は全ての関数・モジュールの中で一番ソフトウェア実行クロックサイクル数が大きいが、全体の処理の中では一度しか実行されず、9～11 回実行される他の関数・モジュールと比べ、負荷割合は低い。このため、KeyExpansion をソフトウェアで実行させることは妥当である。また、SubBytes は全処理中の負荷割合が低いだけでなく他の関数・モジュールと比べハードウェア化した場合の回路規模に対するクロック数の減少率が少ない。そのため、ハードウェア化するよりもソフトウェアで実行するほうが望ましい。これより、解析結果による AES 暗号の分割案は妥当であると考えられる。

4.3. MiBench への適用

4.3.1. MiBench

MiBench とは IEEE では発表された、ベンチマーク・ソフトウェアのセットである。35 種類の組み込みアプリケーション・ソフトウェアを車載機器・産業機器・民生機器・OA 機器・ネットワーク機器・セキュリティ機器・通信機器の 6 種類の応用分野に分類し用意されている。それぞれの用途毎に用意されているアプリケーションの内容を表 13 に示す。

MiBench より、SHA に適用し実験を行う。ベンチマーク・ソフトウェアそのままのソースコードでは関数・モジュール　による分割を考慮した記述がされてないことや、ハードウェアのリファレンスとなる記述になっていないことため、入力のソースコードは自作のものを使用する。

表 13　Mi-Bench アプリケーション内容

車載	民生機器	ネットワーク	OA	セキュリティ	通信機器
Basicmath	JPEG	Dikstra	Ghostscript	blowfish enc.	ADPCM enc.
bitcount	lame	patricia	ispell	blowfish dec.	ADPCM dec.
qsort	mad	(CRC32)	rsynth	pgp sign	FFT
susan(edge)	tiff2bw	(SHA)	sphinx	pgp verify	IFFT
susan(corner)	tiff2rgba	(blowfish)	stringsearch	rijndael enc.	CRC32
susan(smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			SHA	GSM dec.
	typeset				

4.3.2.　SHA への適用

SHA は NIST（米国国立標準技術研究所）によって採用された米国政府標準のハッシュ関数である。SHA-1、SHA-224、SHA-256、SHA-384、SHA-512 の 5 種類が存在しているが、本研究では SHA-1 を扱う。図 13 に SHA-1 によるハッシュの生成フローを示す。SHA-1 では、一般的にデータをパッキングし 512bit にまとめられたデータのことをブロックという。MessagePadding はメッセージデータのパッキングを行う。ブロック化されたデータから SequenceGenerate、CalculateHash によりハッシュデータを生成し、ブロックの個数だけ AddHash で更新することで 150bit のハッシュデータが生成される。ハッシュ生成における計算量はブロック個数分の処理で決まる。

SHA-1 に本システムに適用する。SHA-1 中の CulculateHash を GetF、GetK、及び Rotate の 3 つの関数・モジュールに分割し、MessagePadding はソフトウェアのみで動差させるものとして、分割をする。よって、関数・モジュールは、SequenceGenerate (SG)、AddHash（Add)、GetF、GetK、Rotate の 5 つに分割し、本システムに適用した。尚、実験結果はブロック個数 1 個分の計算量とし実験したものである。

表 14、表 15、及び表 16 に SHA-1 における本システムの解析結果を示す。表 14 は本システムによる SHA-1 暗号の各関数・モジュールの性能についての解析値と、図 3 のシステムで計測した実測値を示す。回路規模、動作周波数などのハードウェアに関する性能は、MicroBlaze を使用せず、MicroBlaze と協調関係のある ISE9.2i による測定とそれを元に算出したものである。表 15 は本システムによる SHA-1 の全分割パターンの性能、及び評価値である。表 16 は SHA-1 に対して本システムが出力した最適な分割案のパターンと、そのときの SHA-1 システムの処理性能、及び評価値を示している。表 15 と表 16 の H と S はハードウェアとソフトウェアを表す。

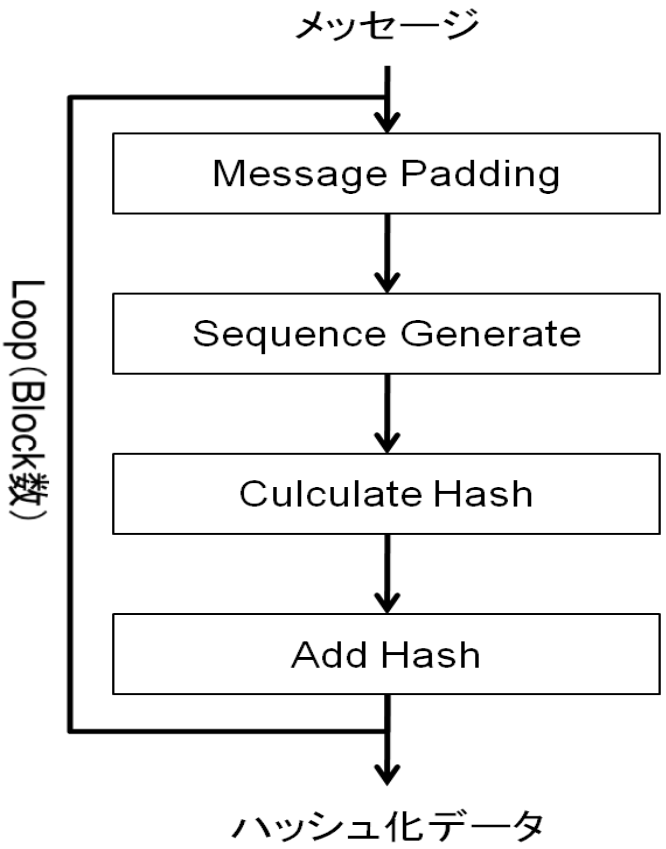

図 13　SHA-1 のハッシュ生成フロー

表 14　本システムによる SHA-1 の各モジュール性能

	SG		Add		GetF		GetK		Rotate	
	解析値	実測値	解析値	実測値	解析値	実測値	解析値	実測値	解析値	実測値
SW 実行サイクル数[Cycles]	4160	4287	80	88	14	34	8	27	68	142
SW 負荷割合[%]	18	13	1	1	5	8	3	6	47	34
HW 実行サイクル数[Cycles]	80	80	5	5	1	1	1	1	1	1
回路規模[Slices]	264	461	16	38	16	199	16	11	16	2
メモリ量[Bytes]	12	72	12	28	16	32	4	16	8	12
最高動作周波数[MHz]	181	156	181	164	181	94	181	123	181	164
並列性	なし	—	なし	—	なし	—	なし	—	なし	—

表 15　本システムによる SHA-1 の全分割パターンの解析結果

パターン番号	SG	Add	GetF	GetK	Rotate	回路規模[slices]	実行サイクル数[cycles]	評価値		
								速度	回路	バランス
1	S	S	S	S	S	0	35801	0	1	0.5
2	S	S	S	S	H	16	25081	0.651	0.951	0.8006
3	S	S	S	H	S	16	35241	0.034	0.951	0.49702
4	S	S	S	H	H	32	24521	0.685	0.902	0.79762
5	S	S	H	S	S	16	34761	0.063	0.951	0.51488
6	S	S	H	S	H	32	24041	0.714	0.902	0.81548
7	S	S	H	H	S	32	34201	0.097	0.902	0.49405
8	S	S	H	H	H	48	23481	0.748	0.854	0.8125
9	S	H	S	S	S	16	35726	0.005	0.951	0.47917
10	S	H	S	S	H	32	25006	0.655	0.902	0.77976
11	S	H	S	H	S	32	35166	0.039	0.902	0.47619
12	S	H	S	H	H	48	24446	0.689	0.854	0.77679
13	S	H	H	S	S	32	34686	0.068	0.902	0.49405
14	S	H	H	S	H	48	23966	0.718	0.854	0.79464
15	S	H	H	H	S	48	34126	0.102	0.854	0.49107
16	S	H	H	H	H	64	23406	0.752	0.805	0.79167
17	H	S	S	S	S	264	31721	0.248	0.195	0.20833
18	H	S	S	S	H	280	21001	0.898	0.146	0.50893
19	H	S	S	H	S	280	31161	0.282	0.146	0.20536
20	H	S	S	H	H	296	20441	0.932	0.098	0.50595
21	H	S	H	S	S	280	30681	0.311	0.146	0.22321
22	H	S	H	S	H	296	19961	0.961	0.098	0.52381
23	H	S	H	H	S	296	30121	0.345	0.098	0.22024
24	H	S	H	H	H	312	19401	0.995	0.049	0.52083
25	H	H	S	S	S	280	31646	0.252	0.146	0.1875
26	H	H	S	S	H	296	20926	0.903	0.098	0.50595
27	H	H	S	H	S	296	31086	0.286	0.098	0.18452
28	H	H	S	H	H	312	20366	0.937	0.049	0.48512
29	H	H	H	S	S	296	30606	0.315	0.098	0.20238
30	H	H	H	S	H	312	19886	0.966	0.049	0.50298
31	H	H	H	H	S	312	30046	0.349	0.049	0.1994
32	H	H	H	H	H	328	19326	1	0	0.5

表 16　本システムによる SHA-1 の分割案

重視項目	SG	Add	GetF	GetK	Rotate	回路規模 [Slices]		実行サイクル数[Cycles]		評価値		
						解析値	実測値	解析値	実測値	速度	回路規模	バランス
速度	H	H	H	H	H	328	711	19326	20252	1	0	0.5
回路規模	S	S	S	S	S	0	0	35801	32976	0	1	0.5
バランス	S	S	H	S	H	32	201	24041	26787	0.714	0.902	0.815

表 14 に示す各関数・モジュールにおける性能・コストの解析値と実測値の比較を行うと、実測値を 1 とした場合の解析値の比率の平均は 1.1 である。しかし、GetF の回路規模が大きく誤差を出しており、後の解析結果にも影響が出ている。また、メモリ量の誤差が全体的に大きく、システムを改良する余地がある。これは一部ポインタ演算を C ソースコードに取り込んだためである。ポインタ演算が使用されると全体的に精度は落ちるが、回路規模とメモリ量の精度が特に落ちる。並列性の解析では全モジュールに並列性がないと判断されている。SHA-1 は逐次処理において、先に計算した結果を次の計算に即座に使うため、データの依存性が高い。そのため、並列性がないと判断されることは妥当であるといえる。

表 15 に示す解析結果は、表 14 の結果を元に解析したものである。速度重視では SW 負荷割合が高く、ハードウェア化による速度向上率の高い Rotate、SequenceGenerate がハードウェアのパターンの評価値が高い。回路規模重視では、しばしば同じ値のパターンの評価値が見られる。表 14 より、回路規模の解析値が同じ関数・モジュールが 4 つあるためである。このため、SequenceGenerate 以外の関数・モジュールに対しては回路規模のみを見ただけではハードウェア化の優先度を決定できない。バランス重視では、全体的に Rotate がハードウェアのパターンの評価値が高い。これらの結果から、Rotate をハードウェア化することが望ましいと考えられる。

表 16 に示す解析結果は、表 15 の結果を元に解析したものである。速度重視・回路重視ではそれぞれ、ハードウェアのみのパターンとソフトウェアのみのパターンを算出しており妥当な結果である。バランス重視では、GetF と Rotate がハードウェアのパターンとなっている。SHA-1 では、MISTY1 暗号や AES 暗号とは違い、SW 実行サイクル数、回路規模、及び二つの性能の比率が解析値と実測値で大きく違う箇所があり、簡単には判断がつきにくい。そのため、表 17 に実測値でバランス重視での評価値を算出したもので優れた評価値のもの 6 パターンを評価値が高い順に示す。

表 17 より、ハードウェア化の優先度が高い関数・モジュールを順番に並べると Rotate、GetK、AddHash、GetF、SequenceGenerate となる。これは、負荷が高く、ハードウェア化の効果が高いモジュールが順に選出されているため妥当であると考えられる。本システ

ムの解析では、評価値が高いパターン番号は順に、6、8、2、4、14、16 となり、ハードウェア化の優先度が高い関数・モジュールを順番に並べると、Rotate、GetF、GetK、AddHash、SequenceGenerate となる。最適な分割パターンは実測値からの結果と解析結果とは違うものとなるが、上位の 6 パターンを見るとほぼ同じパターンが選出されており整合性は高いといえる。また、関数・モジュールのハードウェア化の優先順位も大きく外れているわけではない。これより、SHA-1 においては解析結果の評価値が高いものからサンプリングしていけば有効であるといえる。

表 17　SHA-1 における実測値による評価値が高いパターン

パターン番号	パターン					評価値
	SG	Add	GetF	GetK	Rotate	（バランス）
4	S	S	S	H	H	0.89286
12	S	H	S	H	H	0.85714
2	S	S	S	S	H	0.85714
10	S	H	S	S	H	0.83929
8	S	S	H	H	H	0.78571
6	S	S	H	S	H	0.75

4.4. ハード／ソフト最適分割システムの評価

本システムを Misty1 暗号、AES 暗号、及び SHA-1 の 3 つのアプリケーションに適用した。結果、Misty1 暗号、AES では、クロック数と回路規模についての見積もり精度が高く、パターンの評価も妥当であり、システムが有効であることがわかった。SHA-1 では、一部の回路規模の見積もりの精度が低かったが、優れた評価値のパターンを複数サンプリングすることで、ある程度の精度の低さを補い、パターン評価が行えることを確認した。精度の低さの原因としては、C ソースコードの仕様と実際のハードウェアの構成が違ったことがあげられる。これより、本システムの仕様通りの C ソースコード記述、及び C ソースコードの仕様通りのハードウェア設計をする限りであれば、本システムは有効であることがわかった。また、仕様通りのハードウェア設計が出来なくとも、本システムで評価の高いパターンを複数サンプリングすることで本システムが有効になることがわかった。

全体的な解析の精度としては、著しく低いところもあり、現段階では、ユーザ要求による足きり値の使用は有効ではない。各関数・モジュール性能の見積もり結果と実測値の比率の平均を性能毎に表したグラフを図 14 に示す。

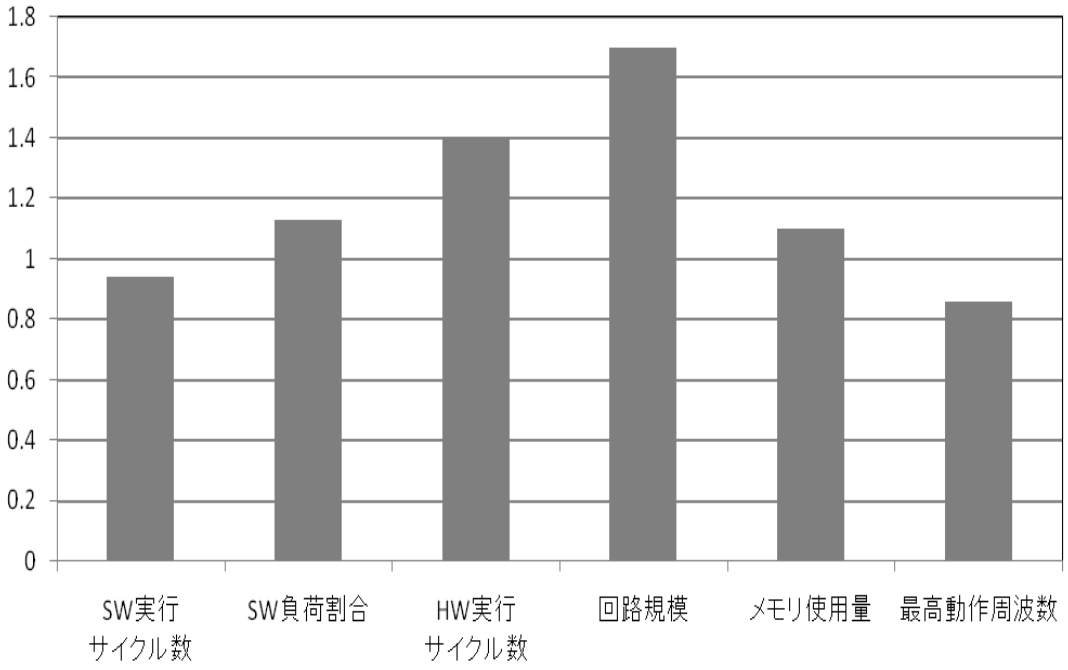

図 14　各性能における解析の実測値との比率平均

図 14 において縦軸は、実測値を 1 としたときの解析値の誤差の平均値となる。SW 実行サイクル数･SW 負荷割合･メモリ使用量などのソフトウェアに関する性能は精度が高いが、HW 実行サイクル数・回路規模・最高動作周波数などのハードウェアに関する性能・コストの精度が低い。これは、ハードウェア設計が、設計者の設計能力・手法に依存することや、C ソースコードだけでは得る情報が限られていることが原因である。現在、システムによる各性能・コストの解析値は、比率が概ね正しいため、システムの最終的な分割案は妥当な結果となっている。そのため、足きり値などのユーザ要求を考慮しない場合は性能・コストの精度の低さは問題ではなく、本システムは有効であると考えられる。

実装環境を考慮した解析を可能にするためにはより詳細な解析が必要となると考えられる。特に現システムにはハードウェアに関する性能・コストの精度向上が必要である。これは、システムを使用する前に使用するハードウェア設計のガイドラインを設定することや、高位合成によるハードウェア生成を対象とし見積もりを行うことなどで解決が可能であると考えられる。

5. おわりに

本論文では、ハードウェア／ソフトウェア協調設計における設計空間領域の探索をするハード／ソフト最適分割システムを提案し、その設計・構築をした。システムのリファレンスとなるプロトタイプの C ソースコードを解析し、関数・モジュール毎に性能・コストを算出し、その結果を利用し、分割案の評価を行い、設計者に分割案を提案するシステムを考案し、作成した。また、現時点での本システムの有効性を検証するために、検証対象として MISTY1 暗号・AES 暗号・MiBench のアプリケーションから SHA-1 を利用し、解析結果から考察を行った。

Misty1 暗号、AES 暗号については本システムによる性能・コストの解析精度が高く、最終的な分割案の解析も妥当なものであることを示した。また SHA-1 では C ソースコードのハードウェア仕様と実際のハードウェアの仕様が違ったため、性能・コスト解析について一部の解析精度が低いものとなったが、本システムで評価される分割案の評価の高いものを複数サンプリングすることで本システムが有効になるということを示した。これらの結果より本システムが有効であることを示した。

今後の課題として、本システムの完成と精度の向上があげられる。現時点では、性能・コスト解析において消費電量の見積もりができていないため見積もり手法の確立と実装が必要である。精度の向上では、SW クロックサイクル数・SW 負荷割合などは精度が高いが、他の項目にやや問題がある。特に、ハードウェアの性能・コストに関わる項目である。設計者の設計能力・手法に依存することから、システムと設計者の間で誤差が生じてしまうことが原因である。これらの差異を埋めることが今後の大きな課題となるであろうと考える。また、FPGA ボード上にマルチコア環境を構築し、並列実験を行うことも課題として挙げられる。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導を頂きました山崎勝弘教授、小柳滋教授に深く感謝いたします。また、ハード／ソフト最適分割システムを立ち上げ、事あるごとに相談に乗って頂き、貴重な助言を頂いた梅原直人氏に深く感謝いたします。

最後に、本研究に関して多くの助言を頂いた高性能計算研究室の皆様に深く感謝いたします。

参考文献

[1] 梅原直人：設計仕様解析によるハード／ソフト最適分割システムの実現と評価，立命館大学理工学研究科修士論文，2007.
[2] 船附誠弘：ソフトマクロ CPU 上でのハード／ソフト協調設計と並列処理環境の実現，立命館大学理工学研究科修士論文，2007.
[3] 古川達久：FPGA 上でのソフト・マクロ CPU によるハードウェア／ソフトウェア分割手法の研究，立命館大学理工学研究科修士論文，2005.
[4] 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卒業論文，2005.
[5] 的場督永：ハード／ソフト最適分割を考慮した JPEG エンコーダの協調設計，立命館大学理工学部卒業論文，2005.
[6] 梅原直人，和田智行，山崎勝弘：設計仕様解析によるハード／ソフト最適分割システムの構築と評価，FIT2007，C-001，2007.
[7] 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
[8] 山崎勝弘：電子デザイン実験Ⅲ「教育用ボードコンピュータ」テキスト，立命館大学理工学部電子情報デザイン学科，2007.
[9] 志水建太：ハード／ソフト協調学習システム上でプロセッサの設計とプロセッサデバッガによる検証，立命館大学理工学部卒業論文，2007.
[10] 三菱電機株式会社：暗号技術仕様書　Misty1，2001.
[11] 松井充：ブロック暗号アルゴリズム MISTY，信学技報 ISEC96-11，1996.
[12] 石渡俊一：SoC 時代のカスタマイズ可能なマイクロプロセッサコア，電子情報通信学会誌，Vol.84，No.7，pp.491-493，2001。
[13] 島田健市，福士将，堀口進：SoPC をベースとした組込みシステムのハードウェア／ソフトウェア分割手法，信学技報，Vol.107，No.255，RECONF2007-24，pp.47-52，2007.
[14] 西村啓成，石浦菜岐佐，石守祥之，神原弘之，冨山宏之：高位合成システム CCAP におけるハードウェアからのソフトウェア関数の呼び出し，情報処理学会研究報告 Vol.2007，No.4，pp.13-18，2007.1.
[15] 中村秀一，安浦寛人：ハードウェア／ソフトウェア同時協調設計のための Soft－Core Processor，情報処理学会研究報告 Vol.1993，No.111，pp.167-174，1993.12.
[16] 遠藤聡，清尾克彦，三井浩康，小泉寿男，神戸英利：SystemC を用いたハードウェア／ソフトウェア協調設計方式とその評価，情報処理学会研究報告，Vol.2007，No.121，pp.39-44，2007.4.
[17] 木下修平，並木滋，近藤信行，中島隆二，清水尚彦：ハードウェア・ソフトウェア協調エミュレーション・シミュレーション手法，情報処理学会研究報告，Vol.2007，No.29，pp.25-32，2007.3.

[18] 西原佑，松本剛史，小松聡，藤田昌宏：FSM への変換に基づく HW／SW 協調設計の形式敵検証手法に関する研究, 情報処理学会研究報告 Vol.2005, No.27, pp.37-42, 2005.3.
[19] 小川修，高木一義，伊藤康史，木村晋二，渡邉勝正：変数のビット幅の最適化に基づく C プログラムからのハードウェアの生成，電子情報通信学会研究報告 Vol.98，No.625，pp.33-40,1999.3.
[20] 山崎大輔，小原俊逸，戸川望，柳澤政生，大附辰夫：HW／SW 協調合成におけるアプリケーションプロセッサの面積／遅延見積もり手法，電子情報通信学会技術研究報告 Vol.106，No.111,113，pp.1-6，2006.6.
[21] 中田育男：コンパイラの構成と最適化，朝倉書店，1999.
[22 松瀬秀作，石田光成，田中正浩，安浦寛人：SoC 開発講座，丸善，2006.
[23] 長尾智晴：最適化アルゴリズム，昭晃堂，2000.
[24] 高橋征義，後藤裕蔵：たのしい Ruby，ソフトバンクパブリッシング，2002.
[25] オブジェクト指向スクリプト言語 Ruby 公式サイトチュートリアル，http://www.ruby-lang.org/ja/
[26] Announcing the DVANCED ENCRYPTION STANDARD（AES），http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[27] MiBench Version1.0，http://www.eecs.umich.edu/mibench/
[28] MiBench：A free, commercially representative embedded benchmark suite, http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf

付録 A　[Misty1 暗号の C ソースコード例]

```
#include<stdio.h>

#define N 8
#define AMAX 8
int plain0, plain1;
int K[8], Kd[8];

unsigned char S7[128]={27, 50, 51, 90, 59, 16, 23, 84, 91, 26,114,115,107, 44,102, 73,
                  31, 36, 19,108, 55, 46, 63, 74, 93, 15, 64, 86, 37, 81, 28, 4,
                  11, 70, 32, 13,123, 53, 68, 66, 43, 30, 65, 20, 75,121, 21,111,
                  14, 85, 9, 54,116, 12,103, 83, 40, 10,126, 56, 2, 7, 96, 41,
                  25, 18,101, 47, 48, 57, 8,104, 95,120, 42, 76,100, 69,117, 61,
                  89, 72, 3, 87,124, 79, 98, 60, 29, 33, 94, 39,106,112, 77, 58,
                  1,109,110, 99, 24,119, 35, 5, 38,118, 0, 49, 45,122,127, 97,
                  80, 34, 17, 6, 71, 22, 82, 78,113, 62,105, 67, 52, 92, 88,125};

unsigned short S9[512]={451,203,339,415,483,233,251, 53,385,185,279,491,307, 9, 45,211,
                   199,330, 55,126,235,356,403,472,163,286, 85, 44, 29,418,355,280,
                   331,338,466, 15, 43, 48,314,229,273,312,398, 99,227,200,500, 27,
                   1,157,248,416,365,499, 28,326,125,209,130,490,387,301,244,414,
                   467,221,482,296,480,236, 89,145, 17,303, 38,220,176,396,271,503,
                   231,364,182,249,216,337,257,332,259,184,340,299,430, 23,113, 12,
                   71, 88,127,420,308,297,132,349,413,434,419, 72,124, 81,458, 35,
                   317,423,357, 59, 66,218,402,206,193,107,159,497,300,388,250,406,
                   481,361,381, 49,384,266,148,474,390,318,284, 96,373,463,103,281,
                   101,104,153,336, 8, 7,380,183, 36, 25,222,295,219,228,425, 82,
                   265,144,412,449, 40,435,309,362,374,223,485,392,197,366,478,433,
                   195,479, 54,238,494,240,147, 73,154,438,105,129,293, 11, 94,180,
                   329,455,372, 62,315,439,142,454,174, 16,149,495, 78,242,509,133,
                   253,246,160,367,131,138,342,155,316,263,359,152,464,489, 3,510,
                   189,290,137,210,399, 18, 51,106,322,237,368,283,226,335,344,305,
                   327, 93,275,461,121,353,421,377,158,436,204, 34,306, 26,232, 4,
                   391,493,407, 57,447,471, 39,395,198,156,208,334,108, 52,498,110,
                   202, 37,186,401,254, 19,262, 47,429,370,475,192,267,470,245,492,
                   269,118,276,427,117,268,484,345, 84,287, 75,196,446,247, 41,164,
                   14,496,119, 77,378,134,139,179,369,191,270,260,151,347,352,360,
                   215,187,102,462,252,146,453,111, 22, 74,161,313,175,241,400, 10,
                   426,323,379, 86,397,358,212,507,333,404,410,135,504,291,167,440,
                   321, 60,505,320, 42,341,282,417,408,213,294,431, 97,302,343,476,
                   114,394,170,150,277,239, 69,123,141,325, 83, 95,376,178, 46, 32,
                   469, 63,457,487,428, 68, 56, 20,177,363,171,181, 90,386,456,468,
                   24,375,100,207,109,256,409,304,346, 5,288,443,445,224, 79,214,
                   319,452,298, 21, 6,255,411,166, 67,136, 80,351,488,289,115,382,
                   188,194,201,371,393,501,116,460,486,424,405, 31, 65, 13,442, 50,
                   61,465,128,168, 87,441,354,328,217,261, 98,122, 33,511,274,264,
                   448,169,285,432,422,205,243, 92,258, 91,473,324,502,173,165, 58,
                   459,310,383, 70,225, 30,477,230,311,506,389,140,143, 64,437,190,
                   120, 0,172,272,350,292, 2,444,162,234,112,508,278,348, 76,450};


/***
FUNCTION : FI,
***/
int FI(int data_16,int KIij){
  signed int nine_side,seven_side;
  int truncated_nine2seven;
  int KIi1,KIi2;
```

```
  nine_side=(0x0000ff80&data_16)>>7;
  seven_side=(0x0000007f&data_16);
  KIi1=(0x0000fe00&KIij)>>9;
  KIi2=0x000001ff&KIij;
  nine_side=S9[nine_side];
  nine_side=nine_side^seven_side;
  //First crossing
  seven_side=S7[seven_side];
  truncated_nine2seven=0x0000007f&nine_side;
  seven_side=seven_side^truncated_nine2seven;
  seven_side=seven_side^KIi1;
  nine_side=nine_side^KIi2;
  //Second crossing
  nine_side=S9[nine_side];
  nine_side=nine_side^seven_side;
  //Third crossing

  return (seven_side<<9)|nine_side;
}

/***
FUNCTION : FO,
***/
int FO(int data_32,int KOi_left,int KOi_right,int KIi_left,int KIi_right){
  int left,right;
  int KIi1,KIi2,KIi3;
  int KOi1,KOi2,KOi3,KOi4;

  //Data masking
  left=(0xffff0000&data_32)>>16;
  right=0x0000ffff&data_32;
  //Key(KIij) masking
  KIi1=KIi_left;
  KIi2=(0xffff0000&KIi_right)>>16;
  KIi3=0x0000ffff&KIi_right;
  //Key(KOij) masking
  KOi1=(0xffff0000&KOi_left)>>16;
  KOi2=0x0000ffff&KOi_left;
  KOi3=(0xffff0000&KOi_right)>>16;
  KOi4=0x0000ffff&KOi_right;
  left=left^KOi1;
  left=FI(left,KIi1);
  left=left^right;
  //First crossing
  right=right^KOi2;
  right=FI(right,KIi2);
  right=right^left;
  //Second crossing
  left=left^KOi3;
  left=FI(left,KIi3);
  left=left^right;
  //Third crossing
  right=right^KOi4;

  return (right<<16)|left;
}

/***
FUNCTION : FL,
```

```
***/
int FL(int data,int KLij){
  int left,right;
  int KLi1,KLi2;

  KLi1=(0xffff0000&KLij)>>16;
  KLi2=0x0000ffff&KLij;
  left=(0xffff0000&data)>>16; right=0x0000ffff&data;
  right=(left&KLi1)^right;
  left=left^(KLi2|right);

  return (left<<16)|right;
}

/***
FUNCTION : KeyScheduling,
***/
void KeyScheduling(int *K128){
  int i;

  for(i=0;i<4;i++){
    K[i*2]=(0xffff0000&K128[i])>>16;
    K[i*2+1]=0x0000ffff&K128[i];
  }

  for(i=0;i<8;i++){
    if(i==7){
      Kd[i]=FI(K[i],K[0]);
      break;
    }
    Kd[i]=FI(K[i],K[i+1]);
  }
}

int main(void){
  //Key-width is 128bit
  int K128[4]={0x00112233,0x44556677,0x8899aabb,0xccddeeff};
  int i;
  int ciph_temp0,ciph_temp1;
  int KI_left,KI_right;
  int KO_left,KO_right;
  int KL;

  if(N%4!=0){
    exit(1);
  }

  KeyScheduling(K128);

  printf("¥nK[]: ");
  for(i=0;i<AMAX;i++){
    printf(" %04x",K[i]);
  }printf("¥nKd[]:");
  for(i=0;i<AMAX;i++){
    printf(" %04x",Kd[i]);
  }printf("¥n¥n");

  plain0=0x01234567; plain1=0x89abcdef;
  ciph_temp0=plain0; ciph_temp1=plain1;
```

```
  /*** Encryption phase ***/
  printf("plain0:%08x, plain1:%08x¥n",ciph_temp0,ciph_temp1);
  for(i=0;i<8;i=i+2){
    KL=(K[((i+2)/2-1)&0x7]<<16)|Kd[((i+2)/2+5)&0x00000007];
    ciph_temp0=FL(ciph_temp0,KL);

    KL=(Kd[((i+2)/2+1)&0x7]<<16)|K[((i+2)/2+3)&0x00000007];
    ciph_temp1=FL(ciph_temp1,KL);

    KO_left=(K[i]<<16)|K[(i+2)&0x00000007];
    KO_right=(K[(i+7)&0x7]<<16)|K[(i+4)&0x00000007];
    KI_left=Kd[(i+5)&0x00000007];
    KI_right=(Kd[(i+1)&0x7]<<16)|Kd[(i+3)&0x00000007];

    ciph_temp1=ciph_temp1^FO(ciph_temp0,KO_left,KO_right,KI_left,KI_right);
    //Crossing

    KO_left=(K[i+1]<<16)|K[(i+3)&0x00000007];
    KO_right=(K[(i)&0x7]<<16)|K[(i+5)&0x00000007];
    KI_left=Kd[(i+6)&0x00000007];
    KI_right=(Kd[(i+2)&0x7]<<16)|Kd[(i+4)&0x00000007];
    ciph_temp0=ciph_temp0^FO(ciph_temp1,KO_left,KO_right,KI_left,KI_right);
    //Crossing
  }
  KL=(K[((i+2)/2-1)&0x7]<<16)|Kd[((i+2)/2+5)&0x00000007];
  ciph_temp0=FL(ciph_temp0,KL);
  KL=(Kd[((i+2)/2+1)&0x7]<<16)|K[((i+2)/2+3)&0x00000007];
  ciph_temp1=FL(ciph_temp1,KL);

  //Final crossing
  printf("¥n%08x %08x¥n",ciph_temp1,ciph_temp0);
  /*** Encryption phase End ***/
  printf("8b1da5f5 6ab3d07c -- expected values¥n");
  return 0;
}
```

付録 B　[AES 暗号の C ソースコード例]

```
#include <stdio.h>

int plain0,plain1;
int K[8],Kd[8];
// Rcon matrix
unsigned char Rcon[11]={0x00,0x01,0x02,
                        0x04,0x08,0x10,
                        0x20,0x40,0x80,
                        0x1b,0x36};
// populate the Sbox matrix
unsigned char Sbox[16][16]={
 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

// Round Expansion Key
 unsigned char exkey[4][44];

/*------------- KeyExpansion fanction --------------*/
unsigned char Sword(unsigned char word){
  return (Sbox[word >> 4][word & 0x0f]);
}

void KeyExpansion(unsigned char key[16]){
  int i,j;
  int col;
  int round;
  for (col = 0; col < 4; ++col){
    exkey[0][col] = key[4*col];
    exkey[1][col] = key[4*col+1];
    exkey[2][col] = key[4*col+2];
    exkey[3][col] = key[4*col+3];
  }
  round = 1;

  for (col = 4; col < 44 ;col++){
    if (col%4 == 0){
      exkey[0][col] = Sword(exkey[1][col-1]) ^ exkey[0][col-4] ^ Rcon[round];
      exkey[1][col] = Sword(exkey[2][col-1]) ^ exkey[1][col-4];
      exkey[2][col] = Sword(exkey[3][col-1]) ^ exkey[2][col-4];
      exkey[3][col] = Sword(exkey[0][col-1]) ^ exkey[3][col-4];
      ++round;
    }
    else{
      exkey[0][col] = exkey[0][col-4] ^ exkey[0][col-1];
```

```
      exkey[1][col] = exkey[1][col-4] ^ exkey[1][col-1];
      exkey[2][col] = exkey[2][col-4] ^ exkey[2][col-1];
      exkey[3][col] = exkey[3][col-4] ^ exkey[3][col-1];
    }
  }
  return ;
}  // KeyExpansion


/*----------- AddRoundKey transformation -----------*/
void AddRoundKey(unsigned char state[4][4],unsigned char exkey[4][44], int round){
  int r, c;
  for (c = 0; c < 4; ++c){
    for (r = 0; r < 4; ++r){
      state[r][c] = state[r][c] ^ exkey[r][(round*4)+c];
    }
  }
  return ;
}  // AddRoundKey()


/*------------ ShiftRows transformation ------------*/
void ShiftRows(unsigned char state[4][4]){
  int r, c;
  unsigned char temp[4][4];

  /* copy to temp */
  for (c = 0; c < 4; ++c){
    for (r = 0; r < 4; ++r){
      temp[r][c] = state[r][c];
    }
  }
  for (r = 1; r < 4; ++r){
    for (c = 0; c < 4; ++c){
      state[r][c] = temp[r][ (c + r) % 4 ];
    }
  }
  return ;
}  // ShiftRows()


/*------------ MixColumns transformation -----------*/
unsigned char GFmult01(unsigned char byte){
  return (byte);
}

unsigned char GFmult02(unsigned char byte){
  if(byte < 0x80)
    return (byte << 1);
  else
    return ((byte << 1) ^ 0x1b);
}

unsigned char GFmult03(unsigned char byte){
  return (GFmult02(byte) ^ byte);
}

void MixColumns(unsigned char state[4][4]){
  int r, c;
```

```
  unsigned char temp[4][4];

  /* copy to temp */
  for (c = 0; c < 4; ++c){
    for (r = 0; r < 4; ++r){
      temp[r][c] = state[r][c];
    }
  }

  for (c = 0; c < 4; ++c)
    {
      state[0][c] = GFmult02(temp[0][c]) ^ GFmult03(temp[1][c]) ^
                                          GFmult01(temp[2][c]) ^ GFmult01(temp[3][c]) ;
      state[1][c] = GFmult01(temp[0][c]) ^ GFmult02(temp[1][c]) ^
                                          GFmult03(temp[2][c]) ^ GFmult01(temp[3][c]) ;
      state[2][c] = GFmult01(temp[0][c]) ^ GFmult01(temp[1][c]) ^
                                          GFmult02(temp[2][c]) ^ GFmult03(temp[3][c]) ;
      state[3][c] = GFmult03(temp[0][c]) ^ GFmult01(temp[1][c]) ^
                                          GFmult01(temp[2][c]) ^ GFmult02(temp[3][c]) ;
    }
  return ;
}  // MixColumns()


/*------------ SubBytes transformation -------------*/
void SubBytes(unsigned char state[4][4]){
  int r, c;
  for (c = 0; c < 4; ++c){
    for (r = 0; r < 4; ++r){
      state[r][c] = Sbox[ state[r][c] >> 4 ][ state[r][c] & 0x0f ];
    }
  }
  return ;
}  // SubBytes()


/*---------------- Main function -------------------*/
int main(){
  unsigned char key_c[16];
  unsigned char din_c[16];
  unsigned char dout_c[16];
  unsigned char state[4][4];
  int i,r;

  ((int *)key_c)[0] = 0x03020100;
  ((int *)key_c)[1] = 0x07060504;
  ((int *)key_c)[2] = 0x0b0a0908;
  ((int *)key_c)[3] = 0x0f0e0d0c;

  ((int *)din_c)[0] = 0x33221100;
  ((int *)din_c)[1] = 0x77665544;
  ((int *)din_c)[2] = 0xbbaa9988;
  ((int *)din_c)[3] = 0xffeeddcc;

  KeyExpansion(key_c);
  /* initialise state[4][4] */
  for (i = 0; i < 4; ++i){
    state[0][i] = din_c[4*i];
    state[1][i] = din_c[4*i+1];
```

```
    state[2][i] = din_c[4*i+2];
    state[3][i] = din_c[4*i+3];
  };

  int j;

  AddRoundKey(state, exkey, 0);
  for(r = 1; r < 10; r++){
    SubBytes(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(state, exkey, r);
  }
  SubBytes(state);
  ShiftRows(state);
  AddRoundKey(state, exkey, r);
  for (i = 0; i < 4; ++i){
    dout_c[4*i]   = state[0][i];
    dout_c[4*i+1] = state[1][i];
    dout_c[4*i+2] = state[2][i];
    dout_c[4*i+3] = state[3][i];
  };

  for (i=0; i<16; i++){
    printf("%2x", dout_c[i]);
  }
  printf("¥n-- expected value is ¥n");
  printf("69c4e0d86a7b 430d8cdb78070b4c55a  OK?¥n");

  return;
}
```

付録 C　[SHA-1 の C ソースコード例]

```
#include<stdio.h>

unsigned char MessageBitLength_Word[8]={0
};
unsigned int Sequence[80]={0
};
unsigned char BlockFromMem[64]={0
};

void    INIT(int    NumOfChar,int    *outBlockSize,int    *outLeftSizeOfLastBlock,unsigned    char
*outMessageBitLength_Word){
  int i;
  unsigned int MessageBitLength = NumOfChar << 3;
  int BlockSize_FullMessage = NumOfChar / 64;
  int time;
  *outLeftSizeOfLastBlock =  64 - (NumOfChar % 64);
  if(*outLeftSizeOfLastBlock > 8)
    *outBlockSize = BlockSize_FullMessage + 1;
  else
    *outBlockSize = BlockSize_FullMessage + 2;

  //Calc massage bit length
  outMessageBitLength_Word[0] = 0;
  outMessageBitLength_Word[1] = 0;
  outMessageBitLength_Word[2] = 0;
  outMessageBitLength_Word[3] = 0;
  outMessageBitLength_Word[4] = (MessageBitLength >> 24) & 0xFF;
  outMessageBitLength_Word[5] = (MessageBitLength >> 16) & 0xFF;
  outMessageBitLength_Word[6] = (MessageBitLength >> 8) & 0xFF;
  outMessageBitLength_Word[7] = MessageBitLength & 0xFF;
}

void     MESSAGE_PADDING(unsigned     char     INPUTDATA[],int     BlockIndex,int     BlockSize,int
LeftSizeOfLastBlock,unsigned char MessageBitLength_Word[],unsigned char *outBlockFromMem){
  int i;

  int time;
  if((BlockIndex < BlockSize - 1 && LeftSizeOfLastBlock > 8)||(BlockIndex < BlockSize - 2 &&
LeftSizeOfLastBlock <= 8)){
    for(i=0;i<64;i++){
      outBlockFromMem[i] = INPUTDATA[i];
    }
  }
  else if(BlockIndex < (BlockSize - 1) && LeftSizeOfLastBlock <= 8){
    for(i=0;i<64;i++){
      outBlockFromMem[i] = INPUTDATA[i];
    }
    outBlockFromMem[64 - LeftSizeOfLastBlock] = 0x80;
    for(i=0;i<32;i++){
      outBlockFromMem[i] = 0x00;
    }
  }
  else if(BlockIndex == BlockSize - 1 && LeftSizeOfLastBlock > 8){
    for(i=0;i<64;i++){
      outBlockFromMem[i] = INPUTDATA[i];
    }
    outBlockFromMem[64 - LeftSizeOfLastBlock] = 0x80;
    for(i=0;i<28;i++){
```

```
      outBlockFromMem[i] = 0x00;
    }
    for(i=56;i<64;i++){
      outBlockFromMem[i] = MessageBitLength_Word[i-56];
    }
  }
  else if(BlockIndex == BlockSize - 1 && LeftSizeOfLastBlock <= 8){
    for(i=0;i<56;i++){
      outBlockFromMem[i] = 0x00;
    }
    for(i=56;i<64;i++){
      outBlockFromMem[i] = MessageBitLength_Word[i-56];
    }
  }
}

void PREPARE_SEQUENCE(unsigned char BlockFromMem[],unsigned int *outSequence){
  int i;
  unsigned int tmp;

  int time;
  for(i=0;i<80;i++){
    if(i<16){
      outSequence[i] = ((int)BlockFromMem[4*i] << 24) + ((int)BlockFromMem[4*i+1] << 16) +
((int)BlockFromMem[4*i+2] << 8) + (int)BlockFromMem[4*i+3];
    }
    else{
      tmp = outSequence[i-3] ^ outSequence[i-8] ^ outSequence[i-14] ^ outSequence[i-16];
      outSequence[i] = (tmp << 1) | (tmp >> 31);
    }
  }
}

unsigned int GetF(unsigned int x,unsigned int y,unsigned int z,int i){
  if(i<20)
    return (x & y) ^ (~x & z);
  else if(i>19 && i<40)
    return x ^ y ^ z;
  else if(i>39 && i<60)
    return (x & y) ^ (x & z) ^ (y & z);
  else
    return x ^ y ^ z;
}


unsigned int GetK(int i){
  if(i<20)
    return 0x5a827999;
  else if(i>19 && i<40)
    return 0x6ed9eba1;
  else if(i>39 && i<60)
    return 0x8f1bbcdc;
  else
    return 0xca62c1d6;
}


unsigned int ROTATE(unsigned int x,int n){
  return (x << n) | (x >> 32 - n);
```

```
}


void CALC_TMP_DIGEST(unsigned int Sequence[],unsigned int HASHVALUE[],unsigned int outTMPHASH[]){
  int i;
  unsigned int a,b,c,d,e;
  unsigned int T;

  int time;

  a = HASHVALUE[0];
  b = HASHVALUE[1];
  c = HASHVALUE[2];
  d = HASHVALUE[3];
  e = HASHVALUE[4];
  for(i=0;i<80;i++){
    T = ROTATE(a,5) + GetF(b,c,d,i) + e + GetK(i) + Sequence[i];
    e = d;
    d = c;
    c = ROTATE(b,30);
    b = a;
    a = T;
  }

  outTMPHASH[0] = a;
  outTMPHASH[1] = b;
  outTMPHASH[2] = c;
  outTMPHASH[3] = d;
  outTMPHASH[4] = e;

}

void ADD_DIGEST(unsigned int TMPHASH[],unsigned int *outHASHVALUE){
  int i;
  int time;
  for(i = 0;i<5;i++){
      outHASHVALUE[i] = outHASHVALUE[i] + TMPHASH[i];
  }

}



int main (void){
  int i;
  int NumOfChar = 3;
  int BlockIndex = 0;
  int BlockSize;
  int LeftSizeOfLastBlock;

  unsigned int TMPHASH[5] = {0x67452301,0xefcdab89,0x98badcfe,0x10325476,0xc3d2e1f0};
  unsigned int HASHVALUE[5] = {0x67452301,0xefcdab89,0x98badcfe,0x10325476,0xc3d2e1f0};

  unsigned char INPUTDATA[64]={0x61,0x62,0x63,0x80,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
```

```
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0};

  //printf("--Entering SHA1 main--¥r¥n");

  INIT(NumOfChar,&BlockSize,&LeftSizeOfLastBlock,MessageBitLength_Word);

  for(BlockIndex = 0;BlockIndex < 1;BlockIndex++){

MESSAGE_PADDING(INPUTDATA,BlockIndex,BlockSize,LeftSizeOfLastBlock,MessageBitLength_Word,BlockFr
omMem);
    PREPARE_SEQUENCE(BlockFromMem,Sequence);
    CALC_TMP_DIGEST(Sequence,HASHVALUE,TMPHASH);
    ADD_DIGEST(TMPHASH,HASHVALUE);
  }
  printf("%x¥r¥n",HASHVALUE[i]);
  //printf("--Exitting SHA1 main--¥r¥n");//END
}
```