MPU の性能に応じた共通鍵暗号の高速実装法

下山 武司 (Takeshi Shimoyama)
通信・放送機構 (TAO)

概要

共通鍵ブロック暗号 DES は 1977 年にアメリカ商務省標準局 (NBS) によって制定された暗号であり、現在に到るまで、共通鍵暗号の事実上の標準として、アメリカのみならず世界中で用いられている暗号アルゴリズムである。DES はもともとハードウエア実装に適するように設計されたと言われているが、長年にわたるソフトウエア実装法の改良によって、汎用マイクロプロセッサ (MPU) 上でも高速な実装が可能となったとされている。本発表では、比較的最近発表されたビットスライス法について触れ、一例として汎用 64 ビット MPU の一つである Ultra SPARC への実装について述べる。

1 はじめに

共通鍵ブロック暗号 DES は 1977 年にアメリカ商務省標準局 (NBS) による公募に対して、唯一 IBM が応募した暗号アルゴリズムを基にして、国家安全保証局 (NSA) による若干の手直しを経て、制定された暗号である [6]。暗号アルゴリズムが公開されているという点で画期的であったこともあり、現在に到るまで、高速暗号アルゴリズムの事実上の標準として、アメリカのみならず世界中で広く用いられているようになった。

DES は、64 ビットの入出力、56 ビットの鍵長を持ち、その暗号化部分は F 関数とよばれる撹拌操作を 16 回繰り返すというアルゴリズムで構成されている。F 関数はさらにビット転置と、S-box とよばれる 6 ビット入力 4 ビット出力の 8 種類の置換テーブルの組み合わせからなっている。

DES の安全性については 56 ビットという鍵の長さに関するもの、S-box の設計基準の非透明性によるもの等、制定当初から研究者の間で議論が交わされている。特に鍵ビット長に関して言えば、近年の計算機技術の発達によって実際に鍵空間の全数探索が可能であることが明らかになってきた。事実、1998 年 6 月には EFF (Electronic Frontier Foundation) が開発した専用鍵探索装置 DES Cracker を用いて 56 時間で鍵を求めてしまう事に成功しており、もはや十分安全とは言えなくなってきている。[5] また 1990 年台に入って急速に発展してきた差分攻撃法、線形攻撃法といった共通鍵ブロック暗号の解読理論の進歩も共通鍵ブロック暗号の世代交替に拍車をかけていると思われる [2, 8]。最近では、DES に続く新たな共通鍵ブロック暗号 AES を NIST (National Institute of Standard Technology) が募集し、現在 (1998 年 12 月) は、応募された 15 候補から 5 個へのしぼり込み選定作業中である。

さて、本論文では DES の安全性については触れず、DES の実装面について、とりわけソフトウエアによる実装法について述べる。DES の暗号化アルゴリズムは、ハードウエア (専用 IC チップ) への実装に向いた構造をしているといわれており、現在では暗号化速度がギガ bit/s を超えるという DES 暗号チップも研究レベルで存在すると聞かれる。逆にソフトウエアによる実装は、やや不利な点がある。例えば鍵スケジュールや IP 変換, FP 変換等で行われるビットの入れ換え操作の実装について、ハードウエアによる実装では配線の入れ換えだけですところ、ソフトウエアで単純に実装した場合には、各ビットの取り出し、並べ変え、書き込み等の処理が必要となり、演算数が増えるだけでなく、メモリアクセスも頻繁に発生し、パイプラインの流れが妨げられ、効率的な演算がしづらくなる。また、S-box による 4 ビット入力 6 ビット出力の変換に関して言えば、最近の大きなワード長を持つ CPU への実装を考えた場合、CPU レジスタビット中ほとんどのビットは用いられず、特に 64 ビット CPU への実装を考えた場合には CPU の持つ本来の性能を活かしきることが難しい。

一方で、ソフトウエアによる効率的な実装に関する研究も行われており、上記の問題に対する解決策についていくつか考えられてきた。ビットスライス法はその一つである。ビットスライス法は、1997 年にイスラエルの暗号研究者 Biham によって提案された実装法であり、ワード長が大きく、レジスタ数が多い CPU 上に DES のようにハードウエアを意識したアルゴリズムをソフトウエアで実装するのに適した方法である。この実装法は CBC モードの暗号化には使う事が出来ないという点と、ビットスライスフォーマットへの変換に手間がかかるという欠点があるものの、ソフトウエアによる DES の高速実装法として認められつつある。ちなみに RSA 社による DES の解読コンテストの優勝者である DESCHALL, Distributed Net らは、この実装法を用いた鍵探索アルゴリズムを用いている [3, 4]。

ビットスライス法は、ビット長の長いプロセッサに向いた実装法であるとはいえ、現実に用いる個々のプロセッサへの実装法に関して言えば、CPU が持つ命令の種類、レジスタ数、キャッシュサイズ、各命令の実行クロック数等の固有のパラメータに大きく影響されてしまう。またパイプラインやスーパースケーラ等 CPU が持つ性能を十分に引き出そうとしても、コンパイラの性能に左右されがちで、思い通りの速度が得られない事も考えられ、実際に実装してみなければ判らない点が多い。

本論文では、64 ビット CPU の一つである Ultra SPARC に、ビットスライス法を実装した際に発生した問題点と、解決のための効率化テクニックを示す。なお、同じ 64 ビット CPU である Alpha 21164a へのビットスライス法の効率的実装については [9] で詳しい記述がある。さて本論文で述べる効率化のポイントは以下の通りである。

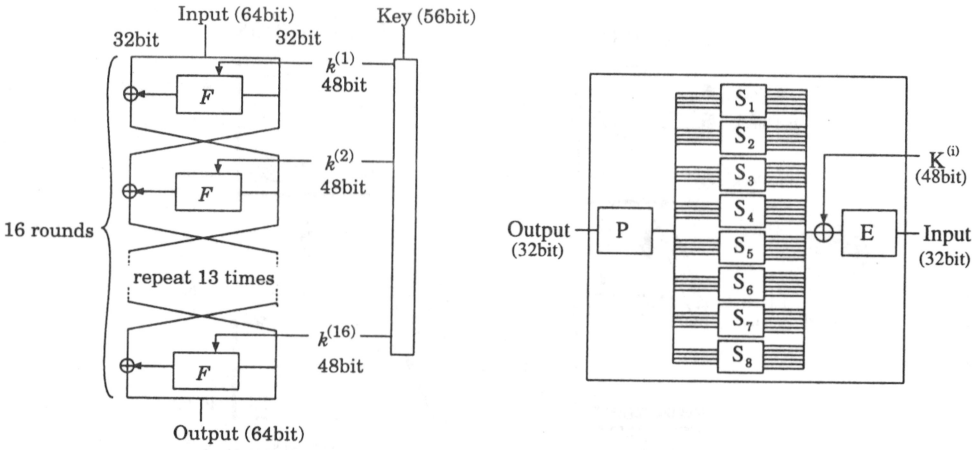
(1) Kwan による S-box 論理表現 [7] (各 S-box 平均 51 命令) を用いる。

(2) プログラムはアセンブリ言語で記述する。

(3) SPARC-V8, 32 ビット論理命令及び SPARC-V9 64 ビット論理命令を並列させる。

(4) 命令の並べ替えや中間変数の入れ換えを組み合わせ、レジスタを有効に使う事でスタックポインタへの不要なロード、ストア命令をほぼ完全に除去する。

図 1: DES の暗号化アルゴリズム



これらの手法を用い SUN Ultra I (Ultra SPARC 167MHz) へビットスライス法を実装した結果、暗号化速度 80.3 Mbit/s を実現し、従来法に比べ 2.35 倍の高速化が実現された事を報告する。

2 DES

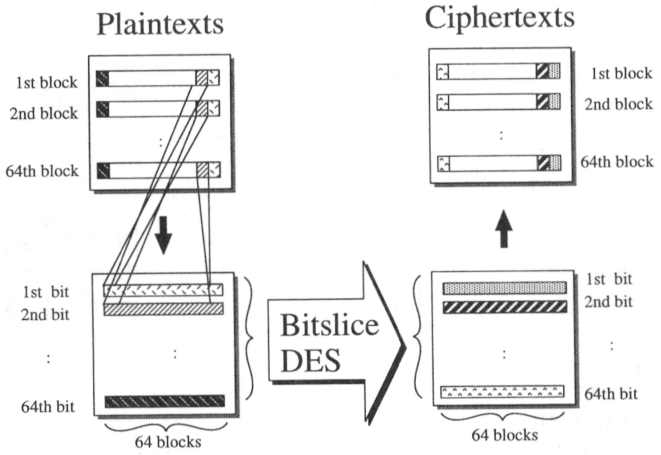
DES (Data Encryption Standard) は、1977 年にアメリカ商務省標準局 (NBS) によって制定された共通鍵ブロック暗号で、64 ビットの入出力、56 ビットの鍵長を持つ。DES の暗号化アルゴリズムは F 関数とよばれる撹拌操作を 16 回繰り返すものであり、F 関数はさらに拡大転置 E、ビット転置 P (線形関数) と、S-box とよばれる 6 ビット入力 4 ビット出力の 8 種類の置換テーブル (非線形関数) の組み合わせからなっている。図 1 は、DES 暗号化アルゴリズムの概念図である。

3 ビットスライス法

ビットスライス法は 1997 年に開催された第 4 回高速ソフトウェア暗号ワークショップ (FSE'97) においてイスラエルの暗号研究者 Biham によって提案された共通鍵ブロック暗号の実装法であり、64 ビットという比較的長いビット長を持つ CPU 上でのソフトウエアによる実装に適することが示されている。

この実装法は 64 ビット長のプロセッサを、1 ビットの演算を 64 個同時に行う SIMD (Single Instruction Multiple Datastream) 型並列計算機としてみて処理を実行するところに大きな特徴がある。64 個の 64 ビットブロックは、各ブロックの第 1 ビットだけを集めたブロック、第 2 ビットだけを集めたブロックとして、以降第 64 ビットまで入力ブロッ

図 2: ビットスライス実装法の全体図



クを再構成し、その後、DES の暗号化関数の論理展開式を用いて、まるでハードウエア上での演算を行うかのごとく各ビット単位の処理に分割し、プロセッサの語長の長さである 64 ブロックを並列に演算を行うことができる。(図 2参照。) 従来の実装法では、ビット転置やテーブル参照等を用いた処理に時間が浪費され、CPU のビット長が長くなるに従って、CPU の本来持つ性能を活かすことが難しくなっていたが、ビットスライス実装法では、ビット巾の長いプロセッサ程暗号化ブロックの並列度を高くすることができるため、より高速化が期待できる。

この実装法では、転置及び拡大転置に関しては、単なるビットの入れ換えであるから、ビット毎の転置の代わりに各ビット列のアドレス変更のみで処理が可能であるが、S-box については非線形写像であるため、論理式展開が必要で、従来の実装法より多くの命令を必要とする。しかし、計算の並列性が効果的に作用するため全体として命令数が減少し、その結果効率化を図ることができる。

DES 暗号化アルゴリズムは S-box を除けは全て線形変換であるから、ビットスライス法による実装の本質は、S-box の論理式表現、さらにその論理式ゲートの個数を如何に少なくするかにあるといっていい。ハードウエアに実装する場合には、論理式計算の並列度を、理論上可能な限り高くしていくことで、実際にかかる演算時間を減らす事ができる一方、ビットスライス実装は、あくまでもソフトウエアによる実装であるから、演算並列度は用いる CPU が持つパイプラインの本数に依存して決まり、それを超えた演算並列度は実現できない。ビットスライス法とハードウエア実装の最も大きな違いは、この点にあると言っていい。

Biham は [1] において 2 入力の `XOR,AND,OR,NOT` を用いた論理回路表現を使用することで S-box を表現できる事を示し、ソフトウェア実装上必要とされる論理回路のゲート数 (=命令数) の上限値を表している。Biham による S-box の論理式表現を用いた場合、一つの S-box につき最大 132 命令、平均 100 命令必要とすると述べている。その後、下

表 1: S-box の論理式展開による演算の命令数

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	平均
Biham	個々の値は示されていない								100
下山	95	84	89	78	95	87	86	88	87.75
Kwan	63	56	57	42	62	57	57	54	62
Kwan'	56	50	53	39	56	53	51	50	55.4

(ただし Kwan' は `AND-NOT`, `OR-NOT`, `NXOR` を用いた場合)

山 [10], Kwan [7] らの改良により、より少ない命令数で S-box を計算する論理式展開が示されている。また、一部の CPU が持つ以下の命令を用いれば、更に命令数を減らす事ができる。

`AND-NOT` ($= A$ `AND` (`NOT` B)), `OR-NOT` ($= A$ `OR` (`NOT` B)), `NXOR` ($= A$ `XOR` (`NOT` B))

本論文の実装におけるターゲットである Ultra SPARC は、上記の命令を持っている事に注目し、Kwan' による S-box の論理式表現 (表 1 参照) を用いることにする。

4 Ultra SPARC プロセッサ

SPARC は SUN Microsystems 社により 1985 年に企画化された CPU インストラクションセットアーキテクチャであり RISC の系統から派生したものである。アドレス空間及び命令セットはすべて 32 bit 幅で統一されており、メモリアクセスはロードとストア命令だけで行われ、全ての演算命令はレジスタ間で行われる。SPARC アーキテクチャの最大の特徴は、各アプリケーション毎にローカルに用いられる整数演算レジスタが、レジスタウィンドウとよばれる空間に分けられ、隣接するレジスタウィンドウ同士ではそれぞれの入力レジスタと出力レジスタが重なり合うようになっている点にある。この構造によって、関数呼出し及び復帰の際に生じるロードとストアの回数が押えられるしくみになっている。

Ultra SPARC プロセッサはこの SPARC アーキテクチャに則って設計された汎用マイクロプロセッサの一つであり、32 ビット命令である SPARC-V8 インストラクションセット及び 64 ビット命令である SPARC-V9 インストラクションセットを持つ。チップ上の一次キャッシュは、データキャッシュ、プログラムキャッシュ各々 16K バイトで合計 32 K バイトである。また、今回用いたプロセッサのクロック数は 167 MHz である。SPARC-V8,V9 が持つデータ型は 32, 64 ビット 整数, 及び 32 ビット, 64 ビット, 128 ビット IEEE 754 標準の浮動小数である。SPARC-V8 命令セット中の 32 ビット整数演算命令は、`%r00` から `%r31` までの 32 本の整数レジスタを用いて計算され、同じく浮動小数演算は `%f00` から `%f31` までの 32 本の浮動小数レジスタを用いて計算される。このように整数演算部と浮動小数点演算部は、各々独立したレジスタを持つユニットとして構成されているため、

表 2: 実験に用いた計算機 (Sun Ultra 1) の仕様

CPU	Ultra SPARC 167 MHz
1 次キャッシュ (on chip)	32K バイト
2 次キャッシュ (on boad)	512K バイト
主記憶	512M バイト
OS	Solaris 2.1.1
コンパイラ	SPARC Compilar version 4.0

メモリアクセス操作を除き各々並列に命令を実行する事ができる。後にこの性質を用いて、演算の並列度をあげることができることを示す。なお本論文では、浮動小数点演算は行わないため、浮動小数に関する機能についてはこれ以上解説しない。浮動小数演算に関する構造を知りたい場合は [11] を参照のこと。

Ultra SPARC プロセッサでは、先程も述べたように、32 ビット整数演算 64 ビット整数演算ともにレジスタ間のみで実行されるが、SPARC-V9 の 64 ビット命令は SPARC-V8 の 32 ビット命令を単純に二つ重ねたものではなく、全く独立した演算装置として位置付けられている。特に、使用するレジスタが異っている事に特徴が現われている。より詳しく言えば、SPARC-V9 の 64 ビット整数演算命令で用いられるレジスタは整数レジスタではなく、32 本の 32 ビット浮動小数演算用レジスタ `%f00`,...,`%f31` を用いて実行されており、実際に 64 ビットレジスタとして用いる場合は、(`%f00`,`%f01`) 等の隣り合うレジスタ組み、合計 16 本を対として用い、またアクセスする際には偶数レジスタ`%f00`, `%f02`, . . . ,`%f30` がその値を代表する。これは、倍精度浮動小数の扱いと同じであると考えてよい。当然の事ながらこれ以外のレジスタは、SPARC-V9 論理命令では参照できない。

5 SPARC-V8,V9 命令とサイクル数

今回使用したワークステーション Sun Ultra 1 の仕様は表 2 の通りである。

本論文で実装のターゲットとしている Ultra SPARC プロセッサは 64 ビットバス、64 ビット命令セットを持つマイクロプロセッサーとして知られているが、現在手元にある SPARC C コンパイラ (version 4.0, Solaris 2.x 上) を用いて実装してみた限りでは、C のソースコードから生成される実行形式としては 64 ビット SPARC-V9 命令は出力されない。よって、本論文の主題であるビットスライス法を実装するにあたって、アセンブリ言語を使用することとする。

Ultra SPARC が持つ 64 bit 論理演算命令は全部で 16 種類あり、各々の命令は浮動小数レジスタを介して実行される。表 3 は、SPARC-V9 (64 ビット) 論理演算命令セットと、その機能に対応する SPARC-V8 (32 ビット) 命令の表である。

ここで、C コンパイラが出力した V8 命令を単純に全て V9 命令に置き換えた場合について考えてみる。実験によって、SPARC-V9 64 ビット論理演算インストラクションの

表 3: SPARC-V8,V9 の論理演算命令セット

SPARC-V8	SPARC-V9	機能
%g0	fzero	zero fill
–	fone	one fill
mov	fsrc1	copy
mov	fsrc2	copy
not	fnot1	negate src
not	fnot2	negate src
or	for	logical OR
–	fnor	logical NOR
and	fand	logical AND
–	fnand	logical NAND
xor	fxor	logical XOR
xnor	fxnor	logical XNOR
–	fornot1	nageted src1 OR src2
orn	fornot2	src1 or nagated OR src2
–	fandnot1	nageted src1 AND src2
andn	fandnot2	src1 AND nageted src2

実行サイクル数は一演算につきすべて 2 サイクルかかることが判った。これは 32 bit 論理演算を 2 回実行する時間と同じである。このことから、C 言語で記述したプログラムのコンパイル時に-S オプションをつけて出力させたアセンブリコード (SPARC-V8 インストラクション) の論理命令を単に SPARC-V9 に書き直した場合、ビットスライスのブロックサイズは 32 ブロックから 64 ブロックへ 2 倍に増やす事ができるものの実行時間も 2 倍かかってしまうため、32 ビット演算をすべて 64 ビット演算に組み変えたとしても結果的に暗号化にかかる時間に差が現われず、これだけでは効果がないことが判った。

6 命令の並列度の向上と冗長なメモリアクセスの除去

6.1 32 ビット命令、64 ビット 命令の並列実行

前章で、SPARC-V9 64 ビット命令単独では効果が上がらない事が判ったため、V8 32 ビット論理演算と V9 64 ビット論理演算とを並列実行させることを考える。Ultra SPARC は SPARC-V8,V9 命令共に 2 重のパイプライン構造を持ち、依存関係のない論理演算を 2 命令同時に実行する事ができ、また、V8, V9 命令は各々全く異る演算部を用いているため、独立に命令を発行させる事ができるために、V8, V9 を並列実行させれば、1 サイクルの間にあわせて、理論的には最大 4 命令の論理演算を並列に実行させる事ができる

表 4: 命令削減手順の一例

オリジナル

```
std    %f10,[%sp+1080]
std    %f20,[%sp+1056]
fxor   %f08,%f18,%f16
ldd    [%sp+1032],%f08
fxor   %f18,%f08,%f20
ldd    [%sp+1096],%f18
ldd    [%sp+1056],%f08
fxor   %f18,%f08,%f12
std    %f20,[%r26+128]
std    %f12,[%r26+176]
```

操作 1

```
std    %f10,[%sp+1080]
std    %f20,[%sp+1056]
fxor   %f08,%f18,%f16
ldd    [%sp+1032],%f08
fxor   %f18,%f08,%f20
ldd    [%sp+1096],%f18
ldd    [%sp+1056],%f12
fxor   %f18,%f12,%f12
std    %f20,[%r26+128]
std    %f12,[%r26+176]
```

%f08→%f12

操作 2

```
std    %f10,[%sp+1080]
std    %f20,[%sp+1056]
fxor   %f08,%f18,%f16
ldd    [%sp+1032],%f08
fxor   %f18,%f08,%f12
ldd    [%sp+1096],%f18
ldd    [%sp+1056],%f20
fxor   %f18,%f20,%f20
std    %f12,[%r26+128]
std    %f20,[%r26+176]
```

%f12↔%f20

操作 3

```
std    %f10,[%sp+1080]
--     --
fxor   %f08,%f18,%f16
ldd    [%sp+1032],%f08
fxor   %f18,%f08,%f12
ldd    [%sp+1096],%f18
--     --
fxor   %f18,%f12,%f12
std    %f12,[%r26+128]
std    %f20,[%r26+176]
```

消去

事になる。この方法によって、CPU の持つ並列度特性を最大限に活かす事ができると考えられる。

ここで、ビットスライス法の本質である S-box の実装について考える。V8, V9 演算は使用するレジスタセットが異るため、メモリーを仲介する以外に値をやり取りすることができない。よって、一つの S-box の論理式演算を V8,V9 命令の両方を用いて演算する事は余分なメモリアクセスが発生し得策とは考えられない。一方で各 S-box の計算は独立に行えるのでそれぞれの S-box を V8 あるいは V9 に均等に割り振ることで、並列性を高める方法をとる。各 S-box の命令数 (56, 50, 53, 39, 56, 53, 51, 50) を比較した結果、S_1, S_2, S_3, S_4 を V8 命令を用いて、S_5, S_6, S_7, S_8 を V9 命令を用いて演算することにし、(S_1, S_5), (S_2, S_7), (S_3, S_6), (S_4, S_8) をそれぞれ組みにする。

6.2 V9 命令を用いた S-box 計算の効率化

今回の実装で用いる Kwan による S-box 論理式表現そのものは、中間変数が論理式の数だけ必要であり、この個数は利用できるレジスタの個数を遥かに超えてしまっている。そのために、このまま実装した場合には頻繁にメモリーアクセスが発生することになる。

Ultra SPARC の場合、メモリーアクセスはロードとストア命令のみに限るよう設計されているため、メモリーアクセスの増加は演算とは関係ない命令を増やす事になってし

まう。しかもそればかりでなく、処理の遅いメモリーへのアクセスが入ることで、使用するデータが供給されるまでの間に CPU の待ち状態ができ、パイプラインを遅延させる要因となる。レジスタ間演算であれば通常最短のサイクル数で実行され、相互依存のない順番にレジスタ間のみで処理ができれば、パイプラインストールも発生しない。よって、S-box の実装を効率的に実装するには、出来る限り中間変数を減らし、レジスタを有効に利用する事によって、メモリーアクセスをなくすようにしなければならない。V9 命令が利用できるレジスタ数は 16 個であるから、中間変数の個数をこれ以下に抑えられれば、無駄なメモリーアクセスが避けられる。

本論文では、C 言語で記述したコードからコンパイラに生成させたアセンブリコードをもとにして、命令の並べ替え、変数名の入れ換え等の手段を用いて手計算により最適化を行った。命令削減手順の一例として表 4 で、オリジナルコードにおけるスタックメモリ [%sp+1056] へのストアとロード (下線) を消去する例を示す。

6.3 V8 命令を用いた S-box 計算の効率化

続いて V8 命令を用いた S-box の論理式計算について述べる。全ての浮動小数レジスタはユーザーが自由に使う事が出来たが、整数レジスタに関しては、%g0 はハードワイヤーゼロ、%o6 はスタックポインタ、%i6 はフレームポインタにそれぞれ割り当てられており、勝手には使えない。そのため多倍長演算のために使える整数演算レジスタは 16 個中 13 個となってしまう。とはいえ、S-box の中間変数の個数をこれ以上減らす事は難しいと思われるため、別の方法を考える。多倍長演算による論理演算は、単精度の論理演算のちょうど 2 倍であることから、入力 64 ブロック中始めの 32 ブロックと後の 32 ブロックの S-box ビットスライス演算を直列に実行する事にした。これにより、それぞれの計算はあくまでも単精度の S-box の論理式展開演算ですむことから、使う事が出来るレジスタ数を 29 に増やすことができる。この方法の利点は、多倍長演算を行った場合と比較しても計算時間に差が現われないことである。ちなみに V9 命令にも 32 ビット演算を行う命令 (fands, fxors 等) があるが、この命令は実験により 64 ビット演算と同じく 2 クロック必要とすることが確かめられるため、32 ビット V9 演算を用いると演算時間が 2 倍になってしまう。V9 命令では V8 と同じような効率化は図れない。

6.4 連続する命令の相互依存性の除去と実装結果

さて、効率化の最終段階として、上記によって得られた S-box 演算において、CPU 演算の並列度を出来る限り高めるには、V9 命令 1 に対し V8 命令 2 を並べ、さらに連続する命令の依存関係をなくすことが必要である。また、最適化をはかるためにはメモリアクセスについても考慮に入れなければならない。Ultra SPARC の 1 次キャッシュから整数レジスタへのロード (1 ワード (32 ビット)) 命令については実験によって、各々 2 サイクルで実行されるデュアルポートを持っていることが確かめられたため、前後に相互依存がなければ 2 サイクルで 2 回のアクセスが可能である。浮動小数レジスタへのロードの場合も同様であるが、2 ワードのロードも 1 ワード同様 2 サイクルで行える所に特色が

表 5: P, E 変換, ラウンド鍵との XOR 演算を含む S-box の命令数

命令数	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
Kwan の S-box 論理式表現	56	50	53	39	56	53	51	50
入出力, 鍵との XOR	30	30	30	30	30	30	30	30
(ld×18, st×4, xor×8)								
C コンパイラ出力†	165	190	168	127	199	155	149	165
最適化後	86	80	83	69	86	83	81	80

(† コンパイルオプション : cc -O4 -S)

表 6: ロードとストア命令並列実行時サイクル数

命令の組み合わせ	サイクル数
相互依存しない 32 ビット整数ロード	2
相互依存した 32 ビット整数ロード	4
相互依存のない 64 ビット整数ロード	4
相互依存のない 32 ビット浮動小数ロード	2
相互依存のない 64 ビット浮動小数ロード	2
相互依存しない 32 ビット整数ストア	4
相互依存した 32 ビット整数ストア	4
相互依存のない 64 ビット整数ストア	8
相互依存のない 32 ビット浮動小数ストア	4
相互依存のない 64 ビット浮動小数ストア	4

ある。なお、ストア命令はデュアルポートを持っていない。以上の事をすべて考慮に入れてコードのスケジューリングを行った結果、V8 演算並列度 1.8, V9 演算並列度 1.9、合計並列度 3.7 (ワード/サイクル) を実現できた。理論的並列度の最大値は、V8, V9 共に 2.0、合計 4.0 であるから、今回の実装で得られた並列度は、ほぼ満足できる数字であると思われる。この実装実験によって、80.3Mbit/s の暗号化速度を達成することができた。この演算速度は、従来法にくらべ 2.35 倍の高速化が実現されたことになる[1]。

7 まとめ

本論文では 64 ビット汎用 MPU である Ultra SPARC 上でのビットスライス法を用いた DES の高速実装について考察し命令並列度 3.7, 暗号化速度 80.3 Mbit/s を達成した。

[1]従来法による DES 実装としてアセンブリ言語を用いて記述されている DES 暗号ライブラリ descore (v 1.12) を用いた。同じ条件での暗号化速度は 34.2 Mbit/s である。

参考文献

[1] E. Biham, "A Fast New DES Implementation in Software," Proc. the Fourth Fast Software Encryption Workshop, LNCS 1267, pp.241-252, 1997.

[2] E. Biham, A. Shamir., "Differential Cryptanalysis of the full 16–round DES," CRYPTO'92, LNCS 740, pp. 487–496 (1992).

[3] DESCHALL homepage., `http://www.frii.com/~rcv/deschall.htm`

[4] Distributed Net homepage., `http://www.distributed.net/des`

[5] Electronic Frontier Foundation homepage., `http://www.eff.org/descracker.html`

[6] FIPS, "Data Encryption Standard," National Bureau of Standards, Federal Information Processing Standards Publications no.46, U.S. Department of Commerce, 1977.

[7] M. Kwan, "bitslice-des," private communication, 1997. (available at `http://www.cs.mu.oz.au/~mkwan/bitslice`)

[8] 松井充., "DES 暗号の線形解読法 (III)," SCIS 94 講演論文集, SCIS94 講演論文集, SCIS94-4A, (1994).

[9] 中嶋純子, 松井充, "MISTY のソフトウェアによる高速実装法について (I)," 信学技法, ISEC97-12, pp.121–131, 1997

[10] T. Shimoyama, S. Amada, S. Moriai., "Improved Fast Software Implementation of Block Ciphers," Proceedings of ICICS'97, LNCS 1334, pp. 269–273, (1997).

[11] SPARC International, Inc, (監訳 : 多田好克), "SPARC アーキテクチャマニュアル バージョン 8," トッパン, 1992.

[12] SunSoft, "SPARC Assembly Language Reference Manual (beta draft)," A Sun Microsystems Inc. Business, 1995. (Builted in SPARC Compiler version 4.0)