システムＬＳＩ設計技術 106－９
（２００２．５．２４）

専用プロセッサ用命令セットの自動生成手法の提案と実装

小島 慶久† 黒羽 毅† 藤田 昌宏††

† 東京大学工学系研究科電子工学専攻 〒113-8656 文京区本郷 7-3-1 工学部 3 号館
E-mail: †{kojima,takeshi}@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

あらまし 本稿では、VLSI システムのハードウェア・ソフトウェア協調設計の一環として、Application Specific プロセッサ用命令セットを自動生成する手法の提案をし、その実装について述べる。アプリケーションに対する専用命令セット生成は、専用命令 (マッチ) による、データフローグラフへの DAG カバリング問題として定式化される。本研究では、制約条件とコスト関数を Multi Terminal Binary Decision Diagram で表現し、汎用 BDD パッケージの Cudd(Colorado University DD package) を用いて最適解を求めた。

キーワード 命令セット生成, MTBDD, ハードウェア・ソフトウェア協調設計, Cudd

Proposition and implementation of a technique for Instruction Set Generation for Application Specific Processors

Yoshihisa KOJIMA†, Takeshi KUROHA†, and Masahiro FUJITA††

† Erectronics Engineering, University of Tokyo Engineering Building No. 3, Hongo 7-3-1, Bunkyo-ku, Tokyo 113-8656, JAPAN
E-mail: †{kojima,takeshi}@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

Abstract In this paper we propose a technique for generating instruction set for application specific processors and show its implementation, as a part of HW/SW co-design of VLSI systems. Generating the instruction set for the target application can be formulated as a DAG covering problem for the data flow graph with the specialized instructions(matches). We built a Multi Terminal Binary Decision Diagram which representing the constraints and the costs, and got the optimal solution with Cudd(Colorado University DD package).

Key words Instruction Set Generation, MTBDD, HW/SW co-design, Cudd

表 1 実現方法 (ソフトウェア、ハードウェア) による特性の比較

	Software	Hardware	SW + HW
チップ面積	△	◎	○
処理速度	×	◎	○
消費電力	×	◎	○
設計期間	○	×	○
設計変更	◎	×	○

1. 背 景

特定用途のシステムの機能を実現するとき、汎用プロセッサを用いソフトウェアで実現する方法と、専用ハードウェア (ASIC) で実現する方法とがあるが、それぞれの特性には一長一短がある (表 1)。これらのそれぞれの長所を活かすために、汎用プロセッサ上のソフトウェアですべての処理を実行するのではなく、特にボトルネックになりそうな処理を複合命令として追加ハードウェアを用いて実装するソフトウェア・ハードウェア協調設計が注目を浴びている。 ここで、短期に設計したい、性能を確保したい、チップ面積を小さくしたい、あるいは消費電力を抑えたいといったさまざまな要求に対して用途に応じた最適化をすることになる。従来、最適化の過程でのハードウェアとソフトウェアの分担の切り分けは設計者の経験と勘に頼っていたが、本研究では、専用命令セットを自動生成 (図 2) することによって、汎用プロセッサと専用ハードウェアの処理の分担の切り分けの自動化を図り、生産性の向上を狙う。

2. システム LSI 設計の流れ

2.1 ターゲットアーキテクチャ

ボトルネックへの対応の仕方を、図 1 に示す。特定用途向けシステムのうち、特にボトルネックになる部分を処理するプロセッサを用意する。ここでは、汎用 RISC コアプロセッサに、ハードウェアを追加命令として実装することを考える。ボトル

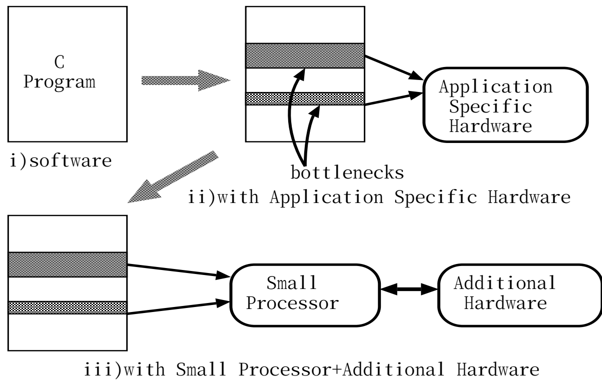

図 1　ソフトウェアとハードウェアの切り分け

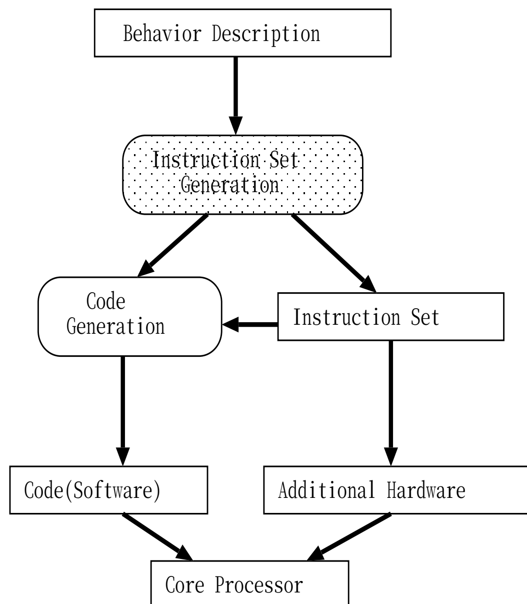

図 2　開発フローと本研究の位置づけ

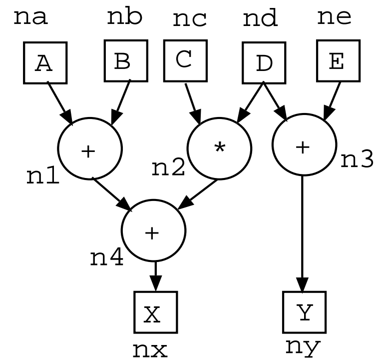

図 3　Data Flow Graph の例 (1)

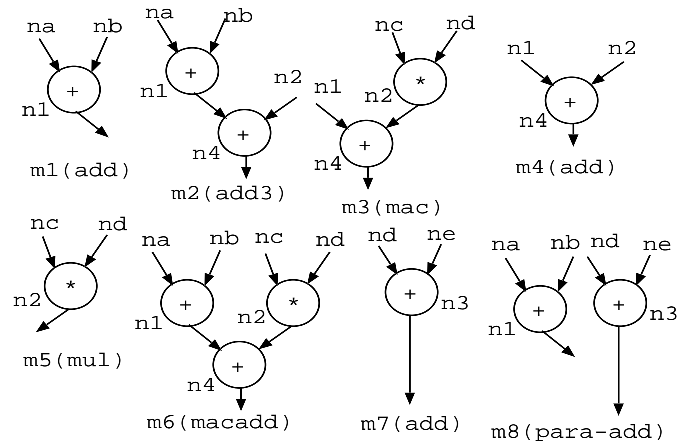

図 4　例 (1) に対して生成されたマッチ

ネック部分を解析することにより、追加すべき命令を探り、コア命令と追加命令を合わせた専用命令セットを生成する。コンパイラはこの専用命令セットを利用してコードを生成する。ボトルネックごとに専用ハードウェアを用意するよりも簡単で、ボトルネックごとのハードウェア資源の共有が可能であり、コード変更で済むレベルならば仕様変更にも対応が可能になる、というメリットがある。当然、追加命令がないコア命令のみで処理するよりも高速であることが期待できる。

3.　問題定義および定式化

ターゲットアプリケーションのボトルネック部分は Data Flow Graph(図 3) で表現されているとし、これを最小コストでカバーする専用命令セットを生成する。

3.1　命令モデル

専用命令は、DFG 上の一つもしくは複数の演算ノードから成るものとし、マッチとして、DFG 上のノードの組み合わせ (図 4) を考える。専用命令セットは選択されたマッチから構成され、専用命令セット生成は、DFG 全体を最小コストでカバーするマッチを選択することとして定式化される。

マッチは、プライマリ・インプットとプライマリ・アウトプットを持ち、専用命令として選択するためには以下に示す制約条件がある。

それぞれのマッチ m_i について、マッチが専用命令として選

ばれたときに 1 となり、そうでないときには 0 となるようなブール変数 m_i (マッチ変数と呼ぶ) を導入する。同様に、それぞれの DFG ノード n_k について、ノードがいずれかのマッチによりプライマリ・アウトプットとしてカバーされたときに 1 となり、そうでないときには 0 となるようなブール変数 n_k (ノード変数と呼ぶ) を導入する。ここでは、マッチの名前や DFG ノードの名前と、対応するブール変数の名前とを特に区別しない。

ノード n_k は ノード n_k をプライマリ・アウトプットとして持つマッチが選択されていればカバーされる。よって、

$n_k = m_a + m_b + m_c ...$ となる。ここで $m_a, m_b, m_c, ...$ は、ノード n_k をプライマリ・アウトプットとしてカバーするマッチの集合である。

マッチが有効な命令となるためには、マッチのプライマリ・インプットのノードがすべてカバーされている必要がある。これは、$n_p * n_q * n_r * ... = 1$ として表される。ここで、$n_p, n_q, n_r ...$ はマッチのプライマリ・インプットである。

図 4 において、マッチ m_2 では n_a、 n_b 、 n_2 がプライマリ・インプットであり、 n_4 がプライマリ・アウトプットである。DFG ノード n_4 はマッチ m_2、m_3、m_4、m_6 のいずれか一つ以上が選択されたときにカバーされる。

最初 DFG のすべての入力ノード (図 3 の $n_a, n_b ... n_e$) はカバーされているとして、最終的に DFG のすべての出力ノード (図 3 の n_x, n_y) がカバーされるように、適切なマッチを選択していく。

専用命令としては、以下の 3 つの種類のものを考える。

- 単一演算命令 (通常の命令。図 4 の m_1, m_4, m_5, m_7)
- 依存性がある複数の演算を処理する直列命令 (MAC 命令など。m_2, m_3, m_6)
- 独立した複数の演算を処理する並列命令 (SIMD 命令など。m_8)

命令を極端に専用化すると、再利用性を損ない、その結果としてコストの増大を招くため、実用的には、マッチは 1 から 4 程度の演算ノードから構成する。

3.2 コストモデル

専用命令の選択においては、以下のコスト関数を用いる。全体のステップ数は、

$$C_{step} = \sum C_{s,i} \cdot m_i$$

と表される。ここで、 $C_{s,i}$ は m_i が必要とするステップ数である。

全体のチップ面積は、

$$C_{area} = \sum C_{a,p} \cdot op_p$$

と表される。ここで、$C_{a,p}$ は演算 op_p が必要とするチップ面積である。ブール変数 op_p は演算 op_p が用いられたときに 1 となり、そうでないときに 0 となる。op_p は $op_p = m_i + m_j + m_k ...$ と表される。ここで、マッチ $m_i, m_j, m_k ...$ は同じ演算 op_p を用いるとする。図 4 においては、$op_{add} = m_1 + m_4 + m_7$ である。

これら二つのコストのトレードオフは

$$C_{total} = v \cdot C_{step} + w \cdot C_{area}$$

と表される。ここで、 v, w はそれぞれステップ数コストと面積コストの重みである。

4. アルゴリズム

ターゲットの Data Flow Graph が図 5 のように与えられているとする。ノードをカバーするマッチを列挙する (図 6)。ここでは、簡単な例として、隣接する 2 つのノードをカバーするものに限定した。これらのマッチの中から、プライマリ・アウトプット n_8, n_9 を共に 1 にする組み合わせを求める。ただし、プライマリ・インプット $n_1, n_2, n_3 = 1$ である。

n_8 は n_6 の出力, n_9 は n_7 の出力なので

$$n_6 * n_7 = 1$$

n_6 を出力とするマッチは m_1, m_2, m_3 であり、一方 n_7 を出力とするマッチは m_4, m_5 である。m_1 を選ぶためには入力 $n_4, n_5 = 1$ が必要、と m_1, m_2, m_3, m_4, m_5 について条件を考慮していくと

$$(m_1 * (n_4 * n_5) + m_2 * (n_1 * n_2 * n_5) + m_3 * (n_2 * n_3 * n_4))$$
$$(m_4 * (n_3 * n_5) + m_5 * (n_2 * n_3)) = 1$$

となる。さらに n_4, n_5 についてマッチの制約条件を導き、プライマリ・インプット $n_1, n_2, n_3 = 1$ を考慮すると、

$$(m_1 * m_6 * m_7 + m_2 * m_7 + m_3 * m_6)$$
$$(m_4 * m_7 + m_5) = 1$$

となる。よって

$$m_1 * m_4 * m_6 * m_7 + m_2 * m_4 * m_7 + m_3 * m_4 * m_6 * m_7$$
$$+ m_1 * m_5 * m_6 * m_7 + m_2 * m_5 * m_7 + m_3 * m_5 * m_6 = 1$$

となる。これらの積項ひとつひとつが、制約条件を満たすマッチの組み合わせである (図 7)。

一般には、これらの制約条件を BDD の形式で表現し、同時に Multi Terminal Binary Decision Diagram を用いて評価関数を記述すると、制約を満たす最適な解を求めることができる。

制約条件を満たすものについてはコストはそのまま、制約条件を満たさないものはコストが∞になるように構築した MTBDD から、コストが一定以下のものを含む BDD を生成する。この「一定以下」の基準は MTBDD の最小値 (最適解のコスト) を用いて決定する。

具体的には、コスト MTBDD に findMin をかけて最適解のコスト min を取り出し、bddInterval によりコストが min と $min + \epsilon$ の間にあるものだけが 1 となるような BDD を生成する。この BDD に PickOneCube をかけて各マッチ変数の値を

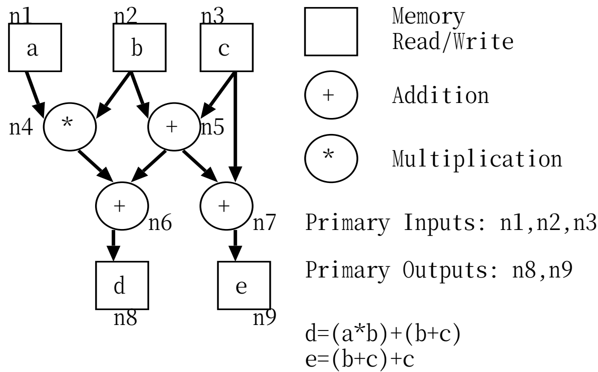

図 5 Data Flow Graph の例 (2)

取り出し、1 になっているものを選択されたマッチとして解釈する。

これは網羅的なアプローチであり、列挙したマッチの範囲内での最適性が保証される。

本研究では、汎用 BDD パッケージである CUDD [1] を用いた。

5. 実験結果

5.1 例題

ここでは、Advanced Encryption Standard として採用された暗号化アルゴリズム Rijndael [2] を例題として利用した。Rijndael で特に使用頻度が高い関数は KeyAddition,MixColumn,ShiftRow,ByteSubstitution であり、この部分の処理の高速化をする。ACE 社の CoSy を用いて関数に含まれる BasicBlock をデータフローグラフ (CDFG) としてダンプし、各 BasicBlock(一つあたりの演算ノード数は 10-30 程度) を独立に専用命令セット生成ツールにかけた。なるべく多くの BasicBlock をまとめて処理する方がよりよい結果を得られるはずであるが、単純に、一つの大きな MTBDD を作ってしまうと、BDD が爆発するため、計算が完了しないという問題がある。

5.2 命令セット生成

生成された命令セットに含まれる命令の数を表 2 に示す。また、生成された命令の一部を、図 8 に示す。

- 直列命令のみを考慮
- マッチに含まれる演算ノード数は 3 まで
- 全体のステップ数を最適化 (面積は考慮しない)
- 各マッチのコストはすべて 1

という条件で計算を行った。

なお、複数の箇所に現れる同一の命令は一つの命令として数えるため、表の数字は単純な算術的加算の合計ではなく、集合的な加算をした合計である。

5.3 コード生成

Code Generation Finite State Machine [3] ベースの Code Generator による最適コード生成を行った結果のステップ数を表 3 に示す。表中、sw はソフトウェアのみ (MIPS R1000 を想定)、our は専用命令セットを使用したプロセッサ、vliw は専用

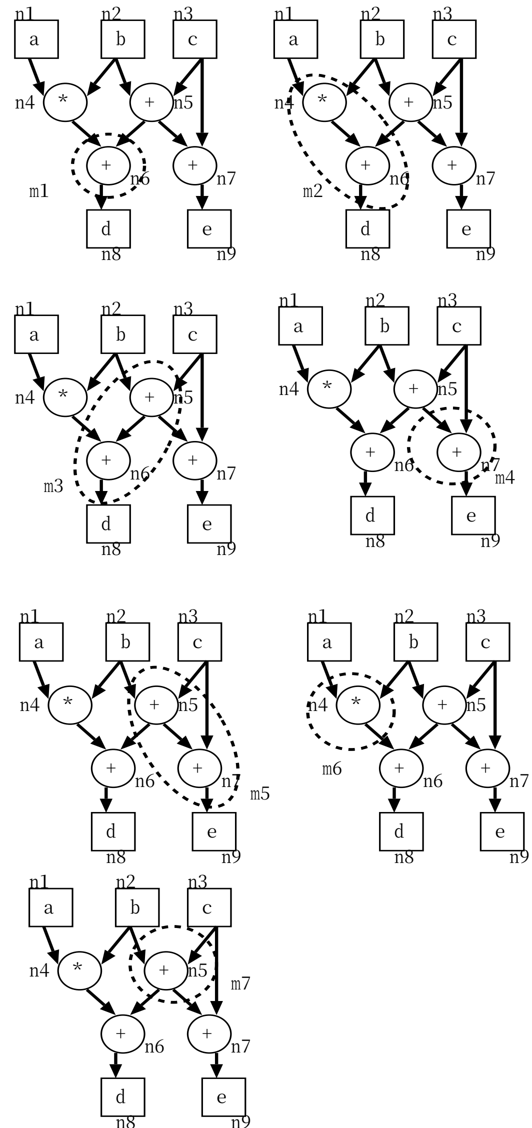

図 6 例 (2) に対するマッチの生成 (隣接する 2 ノードまでに限定)

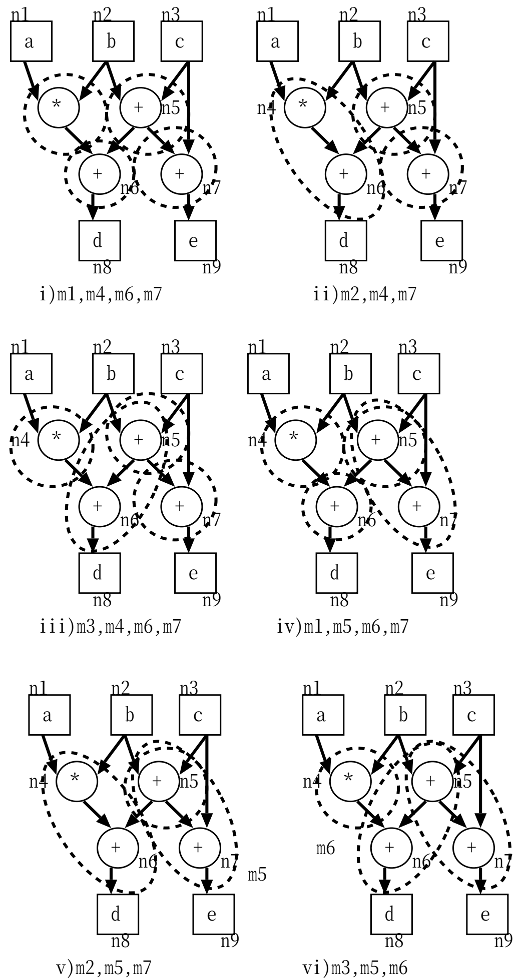

図 7 例 (2) に対する制約条件を満たすマッチの組み合わせ

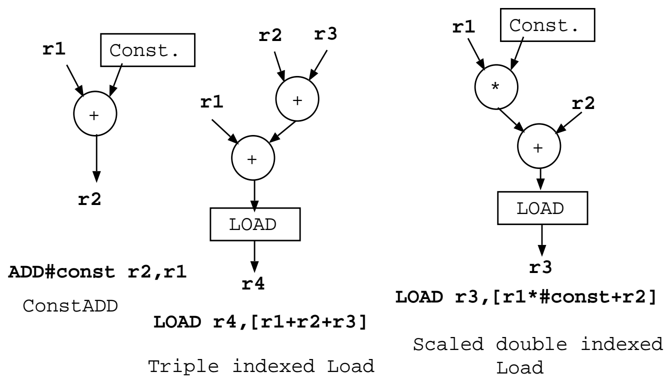

図 8 生成された命令の例 (一部)

表 2 Rijndael 用に生成された命令数

Function	BasicBlock	# of Generated Matches	# of instructions in the solution
KeyAddition	bb1	10	3
	bb2	28	5
	bb3	9	2
	total	-	8
MixColumn	bb10bb11	69	9
	bb13	39	6
	bb1bb2	81	9
	bb4bb5	70	9
	bb7bb8	84	8
	total	-	19
ShiftRow	bb1	19	3
	bb2	28	5
	bb3	7	2
	bb4	18	4
	bb5	9	2
	total	-	8
Substitution	bb1	7	2
	bb2	24	4
	bb3	9	2
	total	-	6
total	-	-	29

表 3　コード生成結果 (ステップ数)

function	non-loop unrolling sw	our	vliw	loop unrolling swu	ouru	vliwu
AddRoundKey	79	60	64	48	20	28
ByteSub	76	44	64	50	20	32
ShiftRow	135	100	120	87	20	36
MixColumn	158	72	96	155	72	96
total	448	276	344	340	132	192

命令を含まない Very Long Instruction Word アーキテクチャのプロセッサをそれぞれターゲットとしている。swu,ouru,vliwu はそれぞれループ展開を施した場合である。ただしリソース制約は考慮していない。ステップ数を比較すると、sw だけの場合に比べ、vliw が 1.3 倍高速であるのに対し、our は 1.6 倍高速となっている。さらに、ループ展開を施した場合は、swu だけの場合に比べ、vliwu が 1.8 倍高速であるのに対し、ouru は 2.6 倍高速になり、改善の結果はさらに顕著になる。

6. 結論と今後の課題

提案した手法により自動生成した専用命令セットが、特定用途のアプリケーション実装の高速化において有利であることを示した。以下に、今後の課題を示す。

6.1 ヒューリスティックによる改善

DFG ノード数が増大すると、マッチ変数の数も増大する。また、マッチに含むノード数を増やす、直列命令のみならず並列命令を許す等、マッチに持たせる自由度を上げると、やはりマッチ変数の数が増大する。生成するマッチの数を増やすことで探索空間が増大し、よりよい解が得られる可能性があるが、一方で、MTBDD が爆発して計算できなくなるという問題がある。

直列命令のみ、マッチあたりノード 3 つまで、という条件で扱える DFG ノード数は 30 程度であり、関数全体をそのまま処理することはできないものの、小さな BasicBlock を扱うことは可能である。

DAG カバリングは NP-hard であり、網羅的なアプローチには限界があるため、より大きな範囲で最適化をかけるためには、明らかに不利に見える解は計算途中で捨てていく等のヒューリスティックが必要となる。

6.2 コード生成

今回の実験では、専用命令セット生成は単純にマッチングパターンを生成するためだけに用いて、コード生成は CGFSM に任せたが、実は、解を求めた時点で DAG カバリングが完了していることを利用すれば、追加的にスケジューリング・アロケーションを行うことで、比較的簡単にコード生成が行えるはずである。

7. 謝　　辞

実験をするにあたり、東大・南谷研の斎藤氏には、Rijndael のプロファイリングで、Pacific Design Inc. の瀬戸氏には、DFG の抽出、CGFSM によるコード生成で大変お世話になった。この場を借りて御礼申し上げる。

文　　献

[1] Fabio Somenzi, "CUDD: CU Decision Diagram Package Release 2.3.1", http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html

[2] J. Daemen, V. Rijmen: AES Proposal: Rijndael, Document Version 2, Sep. 1999.

[3] K. Seto, T. Kuroha, D. Nakatani, K. Asada, M. Fujita: Co-Design of Custom VLIW-DSP Type Data-path Architecture and its Parallel Program for Loops based on Formal Verification Technique, *5th Int. Workshop on Software and Compilers for Embedded Systems*, Mar. 2001.

[4] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: Instruction selection using binate covering for code size optimization, *Proc. Int'l Conf. on Computer-Aided Design*, pp.393–399, 1995. Guido Costa Souza de Araujo ,"Code Generation Algorithms for Digital Signal Processors", PhD Dissertation, Princeton University, Princeton, NJ, June 1997.