論理セルの自動生成および高位合成のスケジューリングに関する研究

1995年

山田晃久

論理セルの自動生成および高位合成のスケジューリングに関する研究

1995年

山　田　晃　久

内容梗概

本論文は, 著者が昭和 62 年から平成 1 年にかけて大阪大学大学院工学研究科博士前期課程に在学中に行った, 大規模集積回路のレイアウト設計における CMOS 論理セルの自動生成に関する研究成果, ならびに平成 4 年から平成 6 年にかけて大阪大学大学院工学研究科博士後期課程に在学中に行った, 機能・論理設計における高位合成のスケジューリングに関する研究成果をまとめたものである.

近年の集積回路技術の急速な進展により, 大規模システムの VLSI 化技法が進展する一方で, 設計複雑度の急激な増大という深刻な問題が発生している. VLSI の設計工程は機能・論理設計と実装設計に大きく分けることができるが, 設計自動化という観点からは, 後者の実装設計について早くから計算機援用設計 (CAD: Computer-Aided Design) 技法の研究開発が進められてきた. すなわち, ゲートアレイ方式, スタンダードセル方式, あるいはマクロセル方式という設計概念が考案され, セル間の配置配線に関して多くの設計自動化技法が実用化され, 特定用途向け集積回路 (ASIC: Application Specific Integrated Circuits) の設計方式として広く普及している.

特に, スタンダードセル方式およびマクロセル方式においては, 多用されるセルはあらかじめ設計し, ライブラリに登録するのが普通であるが, その際セル自体の設計は依然として人手設計に頼っている. 近年のように製造プロセスが頻繁に変更され, 設計規則に変更が起る場合には, その都度これらすべてのセルを再設計しなければならず, それに要する時間と労力は極めて大きく, したがって, これらのセルの自動生成システムの構築が強く望まれている.

一方, 実装設計に比べて自動化が著しく遅れていた機能・論理設計においても, ASIC に搭載される回路規模が数十万, 数百万ゲートに達するようになった今日, 人手による設計は限界に達しており, 設計自動化への要求はますます大きくなろうとしている.

1990 年代に入り, レジスタ転送レベルの記述から論理回路を合成する論理合成技術が実用化されるようになったが, システムの性能やコストを決定するレジスタ転送レベルの設計は依然として人手で行われており, さらに高度な設計支援策が望まれている. その一つの有力な手法として, 動作記述からレジスタ転送レベルの回路を合成する高位合成技法が実用化されてはいる

が, 現状では比較的小規模かつ単純な動作しか扱うことができず, 今後の一層の研究開発が望まれるところである.

本研究では, まず, CMOS 論理セルのレイアウト設計に注目し, これを構成するトランジスタ配列中の各トランジスタの順序が与えられたとき, 指定された設計規則に従ってマスクパターンを自動的に生成する手法について考察する. ついで, 高位合成において, 各演算の実行順序を決定し, 制御ステップに割り当てるスケジューリング問題に対して, 動作記述中に複雑な条件分岐がある場合にも最適な解を与えるような手法について考察する.

本論文は, 全 7 章から構成される. 第 1 章に序論を述べ, 第 2 章では本研究で取り扱う CMOS 論理セルの自動生成問題を定式化し, 第 3 章では CMOS 論理セルの自動生成手法について述べ, 第 4 章では本研究で取り扱う高位合成スケジューリング問題を定式化し, 第 5 章では整数計画問題への定式化と二分決定グラフ上ですべての解を列挙する解法を, 第 6 章では分枝限定法を用いた解法を述べ, 第 7 章に以上の章の結論を述べる.

第 1 章では, VLSI 設計における計算機援用設計, 特に論理セルのレイアウト設計と高位合成におけるスケジューリングに関するこれまでの研究と課題について述べ, 本研究の背景, 目的を明らかにするとともに研究内容と研究成果について概要を述べる.

第 2 章では, CMOS 論理セルの自動生成問題について定式化を行う. まず, 本研究で対象とする CMOS 論理セルのレイアウトモデルについて述べ, 次に, 本研究で扱う問題の定式化を行う.

第 3 章では, CMOS 論理セルの自動生成システムを知識ベースシステムとして構築する手法について考察する. まず, 知識ベースシステム構築言語 OPS5 の概要について述べ, 次に, この OPS5 を用いてマスクデータを生成する際の規則群について考察する. 最後に, 本章で提案する手法を実現し, 実験を通じて提案する手法の有効性を示す.

第 4 章では, 高位合成におけるスケジューリング問題について考察する. まず, 高位合成に関わるスケジューリング問題について概説し, 次に, 動作記述に条件分岐がある場合を含めたスケジューリング問題を定式化する.

第 5 章では, 第 4 章で定義したスケジューリング問題を整数計画問題として定式化し, 二分決定グラフと呼ばれるデータ構造を用いて, すべての解を効率良く列挙する手法について考察する. まず, 条件分岐がある場合に各演算の実行条件を識別する方法について述べ, 次に, 入力の動作記述からスケジューリングに必要な情報を抽出して, 先行関係グラフと呼ばれるグラフの生成法について述べる. さらに, 二分決定グラフを用いた解法について考察し, 最後に, 提案した手法を実験により評価する.

第 6 章では, 第 5 章で示した解法をさらに改善し, より大きなデータが取り扱えるように分枝

限定法を適用する手法を考察する．まず，第 5 章で定式化した整数計画問題を変形する手続きについて述べ，次に，分枝限定法における探索順序と探索中に解のコストの下界を求める関数を導入し，探索操作の高速化を図る．最後に，実験を通じて提案する手法がさらに改善されたことを確認する．

第 7 章では，本研究で得られた成果を要約し，今後に残された課題について述べる．

関連発表論文

I. 学会論文誌発表論文

(1) 山田晃久, 築山修治, 白川功, 神戸尚志: "知識ベースシステムによる CMOS 論理セル自動生成," 電子情報通信学会論文誌 A, vol. J72-A, no. 12, pp. 1236–1242 (1988 年 12 月).

(2) T. Onoye, A. Yamada, I. Arungsrisangchai, M. Tanaka, and I. Shirakawa: "An automatic layout generator for bipolar analog modules," *IEICE Transaction Fundamentals*, vol. E75-A, no. 10, pp. 1306–1314 (Oct. 1992).

(3) A. Yamada, T. Yamazaki, N. Ishiura, I. Shirakawa, and T. Kambe, "Datapath scheduling for behavioral description with conditional branches," to appear in *IEICE Transaction Fundamentals of Electronics, Communications and Computer Sciences*, vol. E77-A, no. 12 (Dec. 1994).

II. 研究会等発表論文（査読付）

(1) A. Yamada, I. Shirakawa, S. Tsukiyama, and S. Shinoda: "An expert system for mask pattern generator of CMOS logic cells," in *Proc. Joint Technical Conference on Circuits/Systems, Computers and Communications*, pp. 299-304 (Nov. 1988).

(2) I. Shirakawa, S. Tsukiyama, S. Shinoda, A. Yamada, and T. Kambe: "An expert system for mask pattern generator of CMOS logic cells," in *Proc. 7th European Conference on Circuit Theory and Design*, pp. 324-328 (Sep. 1989).

(3) T. Okada, A. Yamada, T. Kambe, and I. Shirakawa: "An analog module generator," in *Proc. Third Int. Forum on ASIC and Transducer Technology*, pp. 94-100 (May 1990).

(4) 尾上孝雄, 山田晃久, I. Arungsrisangchai, 田中正和, 白川功: "アナログモジュールの自動レイアウト生成システム," 電子情報通信学会回路とシステム軽井沢ワークショップ, pp. 71–76 (1992 年 4 月).

(5) T. Onoye, A. Yamada, I. Arungsrisangchai, M. Tanaka, and I. Shirakawa: "An automatic layout generator for bipolar analog modules," in *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 2264–2267 (May 1992).

(6) A. Yamada, M. Nakatani, C. Yoshioka, S. Fujihara, and T. Kambe: "A high dense routing method for macro cell layout," in *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 434–443 (Apr. 1992).

(7) 山田晃久, 山崎年樹, 神戸尚志, 石浦菜岐佐, 白川功: "動作記述に条件分岐があるデータパスのスケジューリング手法," 電子情報通信学会回路とシステム軽井沢ワークショップ, pp. 49–54 (1994 年 4 月).

(8) A. Yamada, T. Yamazaki, N. Ishiura, I. Shirakawa, and T. Kambe: "Datapath scheduling based on integer nonlinear programming," in *Proc. Symp. on Nonlinear Theory and its Applications*, pp. 291–294 (Oct. 1994).

(9) A. Yamada, T. Yamazaki, N. Ishiura, I. Shirakawa, and T. Kambe: "Datapath scheduling for conditional resource sharing," in *Proc. IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 169–174 (Dec. 1994).

III. その他研究会等発表論文

(1) 山田晃久, 白川功, 築山修治: "CMOS 論理セルジェネレータの一手法," 電子情報通信学会技術研究報告, CAS87-114 (1987 年 8 月).

(2) 山田晃久, 築山修治, 白川功: "知識ベースシステムによる CMOS 論理セル自動生成," 電子情報通信学会春季大会, SA-7-5 (1989 年 3 月).

(3) 山田晃久, 岡田時仁, 神戸尚志: "アナログ機能ブロック内配置の一手法," 電子情報通信学会春季大会, SA-3-4 (1990 年 3 月).

(4) 吉岡智良, 山田晃久, 中谷美千世, 藤原紳一, 神戸尚志: "マクロセル方式高密度配線手法について," 情報処理学会研究報告, 91-DA-58-6 (1991 年 7 月)

(5) 山田晃久, 中谷美千世, 吉岡智良, 藤原紳一, 神戸尚志: "マクロセル間配線の一手法について," 情報処理学会 DA シンポジウム'91, pp. 65–68 (1991 年 8 月).

(6) 山田晃久, 中谷美千世, 吉岡智良, 藤原紳一, 神戸尚志: “高密度マクロセル間配線の一手法,” 電子情報通信学会技術研究報告, VLD91-87 (1991 年 11 月).

(7) 尾上孝雄, 山田晃久, 神戸尚志, 田中正和, 白川功: “アナログモジュールの自動レイアウト生成システム,” 電子情報通信学会技術研究報告, VLD91-88 (1991 年 11 月).

(8) 尾上孝雄, 山田晃久, I. Arungsrisangchai, 田中正和, 白川功: “アナログモジュールの自動レイアウト生成手法,” 電子情報通信学会春季大会, A-122 (1992 年 3 月).

(9) 山崎年樹, 山田晃久, 石浦菜岐佐, 白川功: “論理関数処理による条件分岐制御のあるデータパスのスケジューリング手法,” 電子情報通信学会秋季大会, SA-3-3 (1993 年 9 月).

(10) 山田晃久, 山崎年樹, 神戸尚志, 石浦菜岐佐, 白川功: “条件分岐制御のあるデータパスのスケジューリング手法,” 電子情報通信学会技術研究報告, VLD93-52 (1993 年 10 月).

(11) 山田晃久, 山崎年樹, 神戸尚志, 石浦菜岐佐, 白川功: “条件分岐を含む動作記述に対するスケジューリング手法,” 情報処理学会 DA シンポジウム'94, pp. 21–26 (1994 年 8 月).

目 次

第 1 章

序論

1.1 背景

近年の集積回路技術の急速な進展により，ディジタル VLSI の集積度は年々急増しており，1 チップに搭載される回路規模も数十万から数百万ゲートにも達しようとしている．これにより，大規模システムの VLSI 化が可能になる一方で，設計複雑度の急激な増大という深刻な問題が生じている．また，生産体制も，従来のように様々なシステムに共通する部品を VLSI 化することにより VLSI を大量に生産するという少品種大量生産型から，システム仕様そのものを先に定め，それに必要な各種 VLSI を個別に生産するという多品種少量生産型へ移行しており，高性能 VLSI を短期間に低コストで設計するための計算機援用設計（CAD: Computer-Aided Design）技法は必要不可欠なものとなっている[1–3]．VLSI の設計工程は，製造技術とは独立に，システム仕様から論理回路を設計する機能・論理設計と，論理回路から製造技術に見合った製造情報を作成する実装設計に大きく分けることができるが，それぞれの設計の各工程で CAD が導入され，設計自動化が進められている[1–7]．

設計の下流にあたる実装設計では，高集積化にともなう処理データ量の急激な増大により，人手だけに頼る設計は不可能となり，早くから設計自動化が行われてきた．すなわち，ゲートアレイ方式，スタンダードセル方式，あるいはマクロセル方式と呼ばれる設計概念が考案され，セル間の配置・配線に関して多くの設計自動化技法が実用化され，現在では特定用途向け集積回路（ASIC: Application Specific Integrated Circuits）の設計方式として広く普及している[1, 3, 4]．

特に，スタンダードセル方式およびマクロセル方式においては，多用されるセルはあらかじめ設計し，ライブラリに登録するのが普通であるが，その際セル自体の設計は依然として人手設計に頼っている．近年のように製造プロセスが頻繁に変更され，設計規則に変更が起る場合には，その都度これらすべてのセルを再設計しなければならず，それに要する時間と労力は極めて大

きく, したがって, これらのセルの自動生成システムの構築が強く望まれている.

一方, 機能・論理設計の設計自動化に関しては, 機能レベル (レジスタ転送レベル) の記述から論理回路を合成する論理合成技術が1960年代から注目され, さまざまな研究が行われてきたが, 扱える問題規模が小さいこともあり実用化には至らなかった[1, 7]. しかし, 近年のように, 設計対象となるシステムが大規模化すると, 従来のような回路図エディタによる論理回路の入力, シミュレーションによる検証という設計スタイルでは効率的な設計は行えず, 変革的な設計自動化技法の開発が要請されるようになった. このような中, 1990年代に入り, 計算機の処理能力の飛躍的な向上にも助けられ, 機能・論理合成技術が導入され, 実用に供し始めている[7, 8].

論理合成を用いた設計方式では, 最終的なシステムの性能やコストはレジスタ転送レベルでの設計仕様に大きく左右されるため, 機能設計の時点で, さまざまな設計仕様を検討する必要がある. しかし, この機能設計は専ら人手で行われているために, 短期間に十分な検討を行うことができず, さらに高度な設計支援策が望まれている. その一つの有力な手法として, アルゴリズムレベルの動作記述から性能やコストに関する制約を与えて, それを満たすレジスタ転送レベルの回路を自動的に生成する高位合成技法が考案されているが, 現状では比較的単純な動作しか扱うことができず, 今後の一層の研究開発が期待されている.

以上に述べた課題を解決するために, 本研究では (I) 論理セルの自動生成と (II) 高位合成のスケジューリングの問題に焦点をあて, 種々の有用な手法を考察する.

(I) CMOS 論理セルの自動生成問題

スタンダードセル方式あるいはマクロセル方式設計において, よく用いられる CMOS 論理セルの自動生成問題を考える.

与えられた論理関数 f を CMOS 回路で構成する際に, f を単調減少論理関数[9]に分解し, それらの各々を CMOS 複合ゲートによって構成するという手法がよく用いられる[9–13]. 与えられた論理関数がこのような CMOS 複合ゲートに構成されるとき, その複合ゲートの全体を論理セルという.

このような論理セルの実現方法としてトランジスタを一次元配列状に配置し, それらの端子間を接続するという方法がよく用いられる. このような論理セルを設計する場合,

(1) セル内のトランジスタの最適な順序の決定

(2) セル内のトランジスタの最適な配置・配線

という手順で行われるのが一般的である. (1) については, これまでにも多くの手法が提案されている[9–13]. 一方, (2) に関しては, セル内において各層にまたがる複雑な設計規則が指定さ

れており, 効率的なアルゴリズムの構築が困難であることや, 一旦自動レイアウトシステムを構築したとしても, セルモデルが変わるなど設計制約に変更がある都度, それに合わせてシステムを変更しなければならないという問題があった. このような難しさからセル生成に対してはほとんど自動化は行われていなかった. そこで, 本研究では, 処理を if-then 型の規則で記述する知識ベースシステムとして構築することによりこれらの問題の解決を図る.

(II) 高位合成におけるスケジューリング問題

高位合成に関わるスケジューリング問題は, 動作記述中の各演算の実行順序を決定し, 制御ステップに割り当てる問題である. 合成後の回路の性能やコストはスケジューリングによって決定されるので, スケジューリングは高位合成の中でも極めて重要な処理ということができる.

このスケジューリング問題に対して, これまでにもさまざまな手法が提案されているが, ディジタル信号処理のフィルタなどのような比較的単純な動作に対するものが多く, 実用的なシステムを構築するためには, 条件分岐やループといったより一般的な動作を扱うことが必要である[14,15]. 本研究では特に, 動作中に条件分岐があるスケジューリング問題について考察する.

条件分岐がある場合のスケジューリングでは, 条件分岐中の演算間で演算器をいかに共有するかがコストを最小化する上で重要な鍵となる. 条件分岐中の演算間で資源を共有できるかどうかは, これらの演算と分岐条件を判断する演算 (以後, **条件演算**と呼ぶ) の先行関係によって決まる. したがって, 条件分岐中の二つの演算が異なる実行条件で実行されるとき,

(i) 条件演算が既に終了していれば, 分岐条件が確定しているので一つの演算器を共有できる.

(ii) 条件演算がまだ終了していなければ, 分岐条件が確定していないので一つの演算器を共有できない.

という二つの事項を考慮に入れなければならない. しかし, これまでに提案された手法の多くは, 問題を簡単にするために, 条件演算は必ず分岐中の演算よりも先行するという前提を設け, 条件分岐中の異なる実行条件を持つ演算間で必ず演算器を共有できるものとして問題を解いていた[16-24]. しかし, このような前提を置いた場合には, 条件演算と分岐中の演算を並行して実行した方が良い解が得られる場合でも, 条件演算を先に実行することを強いられ, その結果, 処理に必要な制御ステップ数が増加することがある.

最近になって, (i), (ii) の両方を考慮した手法が提案されたが[25, 26], これらの手法では, 扱える条件分岐の構造に制限があったり, ヒューリスティックな手法を用いているために最適解が保証されるとは限らなかった.

本研究では, (i), (ii) の両方を考慮し, しかも複雑な条件分岐がある場合にも最適なスケジューリングを得るような手法について考察する.

1.2 本論文の概要

本論文では, 第 2 章および第 3 章で CMOS 論理セルの自動生成について, 第 4 章, 第 5 章, および第 6 章で高位合成のスケジューリングについて議論する.

第 2 章では, CMOS 論理セルの自動生成問題について考察を行う. まず, 本研究で対象とする CMOS 論理セルのレイアウトモデルについて述べ, 次に, 本研究で扱う問題の定式化を行う.

第 3 章では, CMOS 論理セルの自動生成手法[27-31]について考察する. 本手法の主な特徴は, マスクパターンの自動生成がプログラミング言語 OPS5 による知識ベースシステムを用いて実行されるという点である. まず, 知識ベースシステム構築言語 OPS5 の概要について述べ, 次に, この OPS5 を用いてマスクデータを生成する際の規則群について考察する. 最後に, 本章で提案する手法を実現し, 実験を行い, 提案する手法が実用上有効であることを示す.

第 4 章では, 高位合成におけるスケジューリング問題について考察する. まず, 高位合成に関わるスケジューリング問題について概説し, 次に, 動作記述に条件分岐がある場合を含めたスケジューリング問題を定式化する.

第 5 章では, 動作記述中に条件分岐がある場合のスケジューリング手法[32-38]について考察する. 本論文で提案する手法は, スケジューリング問題を整数非線形計画問題として定式化することにより最適解を得ようとするものである. まず, スケジューリングを行う際に必要となる (i) 各演算の実行条件, (ii) 演算間の先行関係という二つの情報を入力の動作記述から抽出する方法について述べる. (i) については, 各演算の実行条件を文献 [39] で提案されたタグを用いて, (ii) については, 先行関係グラフと呼ばれるグラフを用いて表すことにする. 次に, これらの情報をもとにスケジューリング問題を整数非線形計画問題として定式化し, さらに, その解法について考察する. 一般に整数非線形計画問題は効率良く解くことができないが, ここでは文献 [40] と同様, 制約式をブール式で表現し, 目的関数を線形で表して, 二分決定グラフと呼ばれるデータ構造を用いて充足問題を解くことにより, すべての解を効率良く列挙する. 最後に, 実験を通じて本手法の有効性を示す.

しかしながら, ここに提案した手法では, 問題が大きくなると変数の個数が急増し, その結果, 制約式を表すための二分決定グラフが大きくなり過ぎて, 処理速度が激減するという問題点がある.

これを解決するために, 第6章では, 変数の個数が少なくなるように整数非線形計画問題を変形し, 分枝限定法を適用する手法[34–38]について考察する. まず, 第5章で定式化した整数計画問題を変形する手続きについて述べ, 次に, 分枝限定法を用いた解法について考察する. 分枝限定法の効率は解のコストの下界の計算と変数の探索順序に大きく依存するので, 効率良く解を求めるための探索順序と解のコストの下界を求める関数を導入し, 探索操作の高速化を図っている. 最後に, 本手法の高性能性をいくつかの実験により示す.

第7章では, 本研究で得られた成果を要約し, 今後に残された課題について述べる.

第 2 章

CMOS 論理セルの自動生成問題

2.1 緒言

スタンダードセル方式あるいはマクロセル方式 VLSI においては, 頻繁に使用されるスタンダードセルあるいはマクロセルはそれぞれあらかじめ設計されライブラリに登録されるのが普通である. しかしながら, 製造プロセス等の変更により設計規則に変更がある場合には, 通常はこれらすべてのセルを再設計しなければならず, それに要する時間と労力は極めて大きい. 従って, 指定された設計規則を満たすようにセルを自動的に生成するシステムは実用上極めて有用となる.

本研究では, 特に CMOS 論理セルに注目して, その自動生成手法について考察する. CMOS 回路は, 1980 年頃から, 電卓や時計などの低電力用 LSI として民生分野で使われ始めた. CMOS 回路は NMOS 回路に比べて同じ機能を実現するために必要なトランジスタ数が多くなるという短所があるが, 設計の容易さという長所に加え, 微細加工技術の進展とともに集積度も高くなり, 今や LSI の主力テクノロジとして大きく発展している[41]. 本論文では, CMOS 論理セルを構成するトランジスタ配列中の各トランジスタの順序が与えられたとき, 指定された設計規則に従ってマスクパターンを自動的に生成する手法について考察する.

本章では, まず, 本研究で対象とする CMOS 論理セルのレイアウトモデルについて述べ, 次に, 本研究で扱う問題を定式化する.

2.2 論理セルのレイアウトモデル

本研究で取り扱う CMOS 論理セルは p-MOS 側と n-MOS 側のトランジスタの接続構造が互いに双対であるようなスタティック (static) CMOS 回路に限定する. 本章では, 図 2.1 の例が示

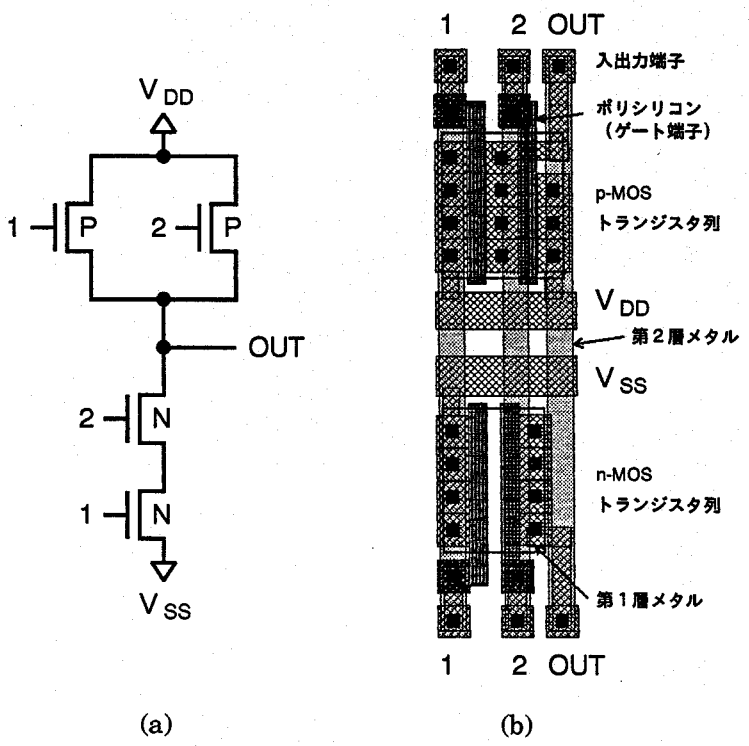


図 2.1: CMOS 論理セルのマスクパターン例.

すように, p-MOS, n-MOS トランジスタは電源線 (V_{DD}, V_{SS}) を境にそれぞれその上, 下側で 1 次元配列状に配置するものとし, かつ論理セル内においては, 第 1 層メタル, 第 2 層メタル, ポリシリコン層を以下のように用いる.

1. 上下端の対応する入出力端子 (第 1 層メタル一第 2 層メタル間のコンタクト) を接続する際の垂直配線には第 2 層メタルを用いる.
2. ゲート端子にはポリシリコン層を用いる.
3. 他のすべての配線 (電源線 (V_{DD}, V_{SS}), ゲート端子と入出力端子との間, ソース/ドレイン端子間, ソース/ドレイン端子と入出力端子との間, ソース/ドレイン端子と電源との間) には第 1 層メタルを用いる.

以下では, マスクパターンを描く際には, 主題となる第 1 層メタルにだけ注目し, 第 2 層メタルを省略する. 例えば, 図 2.1 (b) のマスクパターンの第 1 層メタルだけに注目すれば, 図 2.2 のように描かれる.

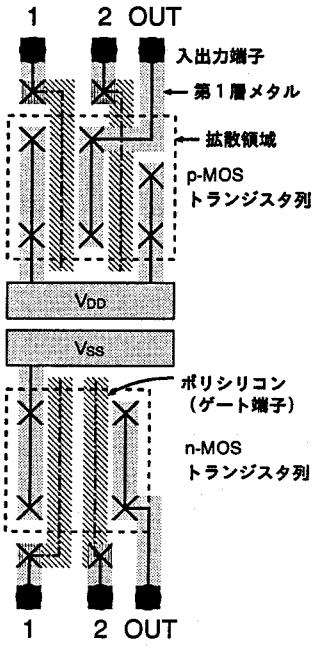


図 2.2: CMOS 論理セルの表現方法.

2.3 問題の定式化

図 2.3 (a) のトランジスタ回路を実現する CMOS 論理セルの基本的なレイアウトは図 2.3 (b) のように描かれる. 各トランジスタは独立して配置され, トランジスタを形成する拡散領域はそれぞれ分離されている. 隣接する拡散領域が同一ネットに属する場合は, 図 2.4 に示すように, 第 1 層メタルで接続する代わりに拡散領域を用いて接続することができるので, この操作を図 2.3 (b) のセルに対して行うと, 図 2.5 (a) のようなレイアウトが得られる. さらにトランジスタの順序の入れ換えおよびトランジスタの左右の拡散領域の入れ換えを行って, トランジスタの拡散領域による接続を最大限行うと, 図 2.5 (b) のように面積最小のレイアウトを得ることができる. ただし, 拡散領域による接続を行い, 面積最小のレイアウトを得るめには, 規則的にトランジスタを配置する図 2.3 (b) の場合に比べて, 各素子 (トランジスタや入出力端子) の配置・配線は複雑になる.

この例が示すように面積最小の CMOS 論理セルを設計しようとした場合,

1. 分離を最小とするようなトランジスタの順序の決定.
2. 面積を最小とするようなトランジスタ間の配置・配線.

をいかにして行うかという二つの問題を解く必要がある. 1. の問題は, 与えられた回路に対し

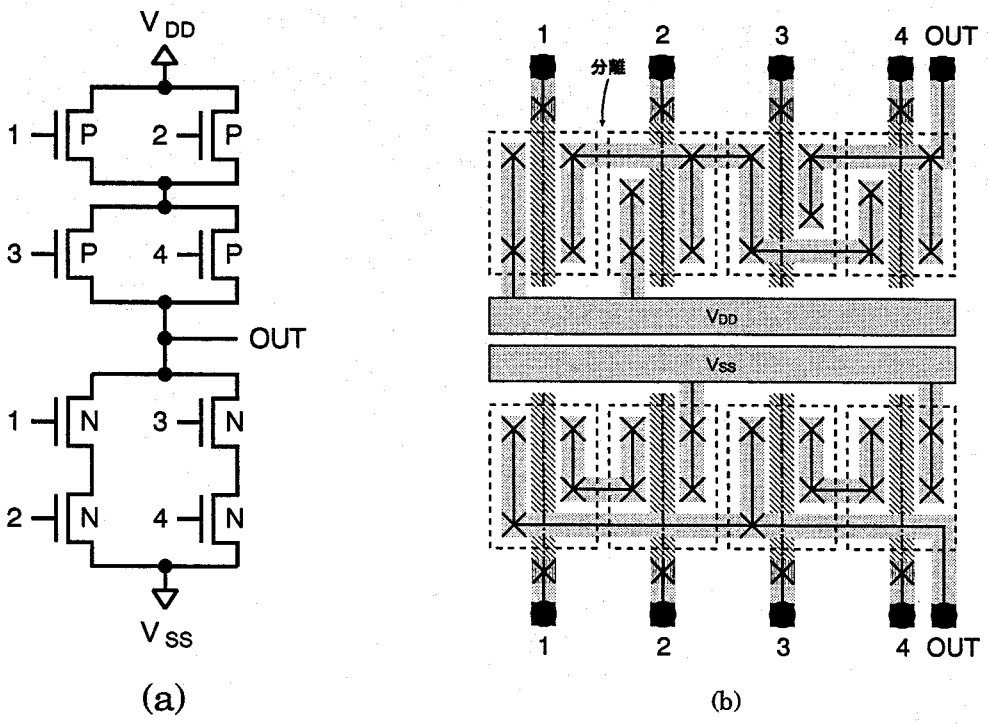


図 2.3: 論理セルの基本的なレイアウト: (a) トランジスタ回路, (b) レイアウト.

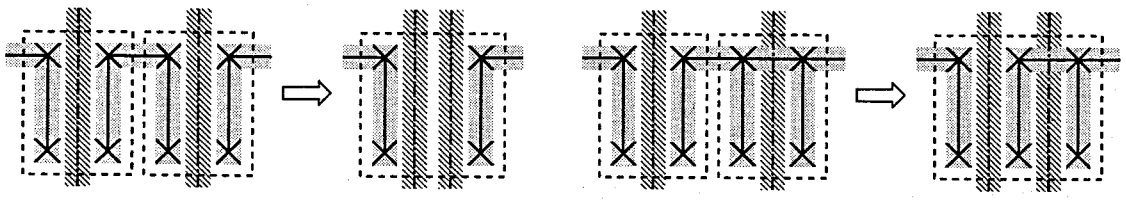


図 2.4: 拡散領域による接続.

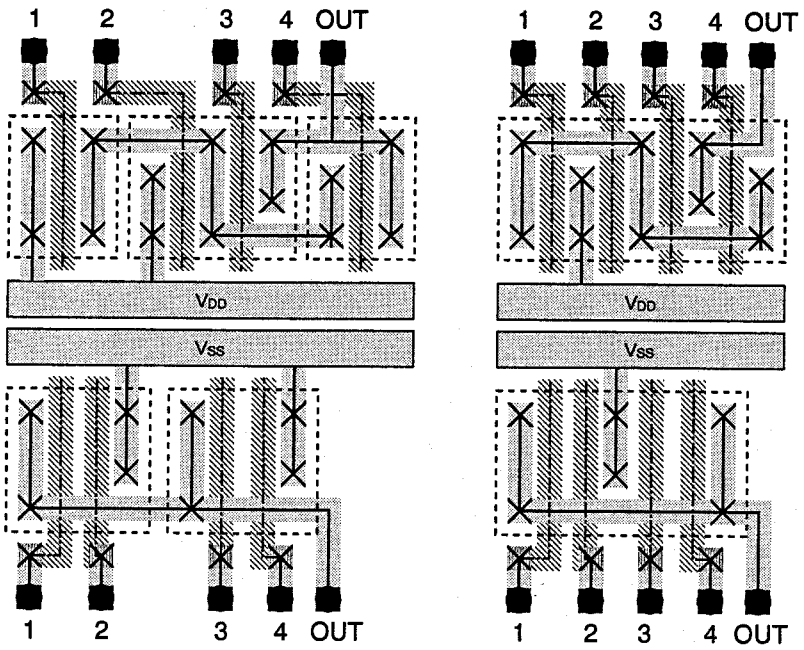


図 2.5: 論理セルの最適化: (a) 拡散領域による接続を行った場合, (b) トランジスタの順序の入れ換えとトランジスタの左右の拡散領域の入れ換えも行なった場合.

て, 回路中の p-MOS 側, および n-MOS 側のトランジスタの接続関係を,

(i) すべてのトランジスタのソース/ドレインのネットを節点で表す.

(ii) 各トランジスタを, そのトランジスタのソースとドレインを表すネットに対応する節点を接続する枝で表す.

という定義で与えられるグラフ (それぞれ, **p-グラフ**, **n-グラフ**と呼ぶ) で表現すれば (図 2.6 (a)), この二つのグラフを最小個数の双対道 (dual trail[13]: 両方のグラフで同じ枝の系列を持つ道) で被覆する問題として定式化することができ[10] (図 2.6 (b)), これまでにも様々な手法が提案されている[9-13].

本研究では, 2. の問題に注目し, 次の問題を考えることにする.

[CMOS 論理セルの自動生成問題]

CMOS トランジスタ回路と, CMOS 論理セル内でのトランジスタの順序が与えられたとき, 指定された設計規則を満たし, かつ面積が最小となるようにトランジスタの

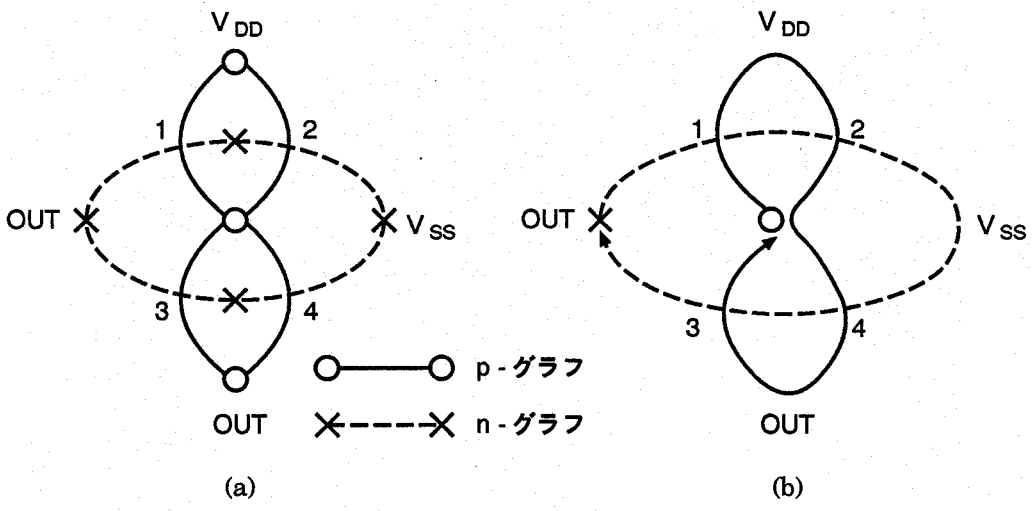


図 2.6: 図 2.1 (a) のトランジスタ回路のグラフモデル: (a) グラフモデル, (b) 図 2.2 (b) に対応する双対道.

端子や入出力端子を配置し, 端子間の配線を行う.

ただし, セル内の配置・配線に関して, 設計規則以外に次のような制約を設ける.

(1) トランジスタの各端子, 電源, 入出力端子の縦方向の位置はあらかじめ定められている.

(2) p-MOS, n-MOS 側の対応する入出力端子は, 第 2 層メタルを用いて垂直に接続されるので, 横方向に同じ位置にある.

(3) ゲート端子と入出力端子を接続する際には, 入出力端子から第 1 層メタルで垂直に取りだして, L 型に折り曲げたポリシリコン (ゲート端子) とコンタクトで接続する.

(4) ソース/ドレインを構成する拡散領域と第 1 層メタルを接続する際には, コンタクト抵抗が小さくなるように, できるだけ多くのコンタクトを設ける.

2.4 結言

本章では, 本研究で対象とする CMOS 論理セルのレイアウトモデルについて述べ, 本研究で取り扱う自動生成問題を定式化した. このレイアウト問題は二つの部分問題を含む. その一つは各種の設計規則を満たすようなレイアウト素子の位置決めをどのようにして行うかということであり, 他の一つは各セルの配線経路をどのようにして見い出すかということである.

第 3 章

知識ベースシステムを用いた CMOS 論理セルの自動生成手法

3.1 緒言

本章では, CMOS トランジスタ回路とそれを実現する論理セル上でのトランジスタの順序が与えられたとき, 指定された設計規則に従ってそのマスクパターンを自動的に生成する手法[27-31]について考察する.

CMOS 論理セル内においては各層にまたがる複雑な設計規則が指定されており, これらの規則すべてを満たすような自動生成アルゴリズムの構築は極めて困難であるため, 処理を if-then 型の規則で記述する知識ベースシステムとして構築する. 知識ベースシステムを用いたシステムとして NMOS セルの自動生成システムが提案されているが[42-44], これらは数個のトランジスタからなるブロック間の配置配線を行い, それに基づいて描かれるマスクパターンにコンパクションを施そうというものである. これに対して本手法は, CMOS 複合ゲート内のトランジスタのソース, ドレイン, およびゲートの位置決めおよびそれらの間の配線設計を設計規則を満たすように行って最終的なマスクパターンを生成しようとするものである.

本章では, まず, 知識ベースシステム構築用言語 OPS5[45]の概要を述べ, 次に, マスクパターンを作成する際の規則群について述べる. さらに提案する手法を実現し, 実験を通じて提案する手法の有効性を示す.

3.2 知識ベースシステム構築用言語 OPS5 の概要

規則 (rule) として表現された知識をもとに推論して問題を解決するシステムを知識ベースシステム, ルールベースシステム, あるいはプロダクションシステムといい, 多くのエキスパート

```
(p place_terminal_according_to_design_rule
  (place_transistor ^state active)
  ({ <terminal1> << poly md_contact >> }
    ^position <position> ^x <x1> ^type <type>)
  ({ <terminal2> << poly md_contact >> }
    ^position ( compute <position> - 1 ) ^x <x2> ^type <type>)
  (design_rule ^layer1 poly ^layer2 md_contact ^space <space>)
  (drc ^judge (check <x2> <x1> <space>))
-->
  (modify 2 ^x (compute <x2> + <space>)))
```

図 3.1: 規則の例.

```
(design_rule ^layer1 md_contact ^layer2 md_contact ^space 7000)
(design_rule ^layer1 poly       ^layer2 poly       ^space 5000)
(design_rule ^layer1 md_contact ^layer2 poly       ^space 4000)
(design_rule ^layer1 metal      ^layer2 metal      ^space 4500)
(md_contact ^name d1 ^position 1 ^x 0 ^type p ^net_name VDD ^routed nil)
(poly       ^name p1 ^position 2 ^x 0 ^type p ^net_name 1   ^routed nil)
(md_contact ^name d2 ^position 3 ^x 0 ^type p ^net_name 9   ^routed nil)
(poly       ^name p2 ^position 4 ^x 0 ^type p ^net_name 2   ^routed nil)
```

図 3.2: 作業記憶要素の例.

システムに用いられている. OPS5 は, この知識ベースシステムを構築するための一つの記述言語であり, プロダクション記憶 (production memory), 作業記憶 (working memory), および推論部 (interpreter) からなる. プロダクション記憶には「if <条件> then <動作>」型の規則を集めた知識ベースが格納され, 作業記憶には規則によって参照されかつ更新されるデータが蓄えられ, 推論部はプロダクション記憶と作業記憶とを照合し, プロダクション記憶中から実行可能な規則を選択し, 作業記憶中の対応する要素に適用し, その内容を更新する[46–48].

本手法で用いる OPS5 の規則は, 例えば図 3.1 のように記述される. この規則は, 設計規則を満足しない二つのトランジスタの端子が存在した場合に, 一方の端子を設計規則を満足する位置に移動するというものである. また, 作業記憶には図 3.2 のような設計規則 (1–4 行目) や各トランジスタの端子についての情報 (5–8 行目) など設計に必要なデータが蓄えられる.

知識ベースシステムには,

- パターンマッチングを用いた処理を簡潔に記述できる.
- 規則を追加, 修正することにより, 比較的容易に機能の変更が行える.

という特長がある. 論理セルの自動生成においては, 上述のような設計規則検証や後述の配線予測など, パターンマッチングが必要となるような処理が多く, 知識ベースシステムは極めて効果的である. また, 設計規則だけでなく設計制約の変更に対しても, C や Pascal のような手続き型言語でプログラム中にコーディングする場合よりも, 規則の追加, 修正によって容易に対応できる.

3.3 マスクデータの作成

本節では, マスクデータを作成する際の生成規則について考察する.
マスクデータを作成する際の処理操作を大別すると次のようになる.

1°: トランジスタの端子の配置：p-MOS, n-MOS トランジスタのソース/ドレイン端子とゲート端子の配置.

2°: 入出力端子の配置：入出力端子としての第 1 層メタル−第 2 層メタル間のコンタクトの配置.

3°: 内部配線：ソース/ドレイン端子と入出力端子の間の配線.

1°, 2°の配置に関しては, 2.3 節で述べたように, 入出力端子, ゲート端子, ソース/ドレイン端子等のセル上での縦方向の位置はあらかじめ定められているものとし, 横方向の位置だけを決定することとする. ただし, 簡単のために設計規則は中心間の距離に関してのみ定められているものとし, したがってトランジスタの各端子, メタル, ポリシリコンは幅のない点や線として取り扱うものとする.

以下で, これらの処理操作を用いてマスクデータを作成する手順を示す.

3.3.1 トランジスタの端子の配置

トランジスタの端子を配置する際の制約は, 端子間の間隔に関するいくつかの設計規則に基づいて設定される. いま, 横方向を x 方向とし, 右向きを正方向とする. $x = 0$ に左端の端子を置き, 順次端子を配置することにすると, p-MOS, n-MOS トランジスタの各端子の位置は, 図 3.3 の配置規則によって決定される. ただし, すべての端子には, 左から昇順に番号を付けてあり, その位置は初期化により $x = 0$ となっている.

トランジスタの端子についての設計規則検証の要請が出され, $i-1$ 番目の端子と i 番目の端子との間に設計規則違反が生じているとき, 規則 1 により i 番目の端子が $i-1$ 番目の端子との

[規則1]
if （トランジスタの端子についての設計規則検証を行っている）
and （$i-1$番目の端子が$x = x_{i-1}$に置かれている）
and （$i-1$番目とi番目の端子間の設計規則αを満足する最小距離はD_αである）
and （i番目の端子が$x = x_i < x_{i-1} + D_\alpha$に置かれている）
then （i番目の端子を$x = x_{i-1} + D_\alpha$に置く）

[規則2]
if （トランジスタの端子についての設計規則検証を行っている）
then （処理を入出力端子の配置に移す）

図 3.3: トランジスタの端子の配置規則.

設計規則を満足する位置に置かれる. 規則1が実行されるとき, 規則2の条件部も満たされ実行可能となっているが, 規則1の条件がより厳しいため, まず規則1が実行される. その後すべての端子間の設計規則が満足されれば規則2によりトランジスタの配置が終了し, 入出力端子の配置に移る.

3.3.2 入出力端子の配置

入出力端子を配置する際の制約は, 次の三つである.

(1) 入出力端子間の設計規則（図3.4 (a)).

(2) 入出力端子とゲート端子とを接続する際に必要な第1層メタルーポリシリコン間のコンタクトと隣接するゲート端子（ポリシリコン）との設計規則（図3.4 (b)).

(3) p-MOS, n-MOS側の対応する入出力端子は, 第2層メタルを用いて垂直に接続されるので, 横方向に同じ位置にあること.

これらの制約を満足するように, 左から順にp-MOS, n-MOS側の対応する入出力端子を, 同時に同じ位置（横方向）に配置する. ただし, 制約2に関して, 入出力端子は以下のように配置される：まずその左側の入出力端子とゲート端子にのみ注目して位置を決定し, その後右側のゲート端子との設計規則検証を行い, 規則違反が生じている場合には, 規則1, 2によってそのゲート端子およびそれより右の端子を設計規則を満足する位置に移動する. 以上の制約に基づいて入出力端子の配置規則は, 図3.5のように記述される.

規則3によってまず左端の入出力端子が$x = 0$に配置される. それより右の入出力端子は規則4によって制約1–3を満足する位置に配置される. 規則5は論理セルの出力端子を配置する

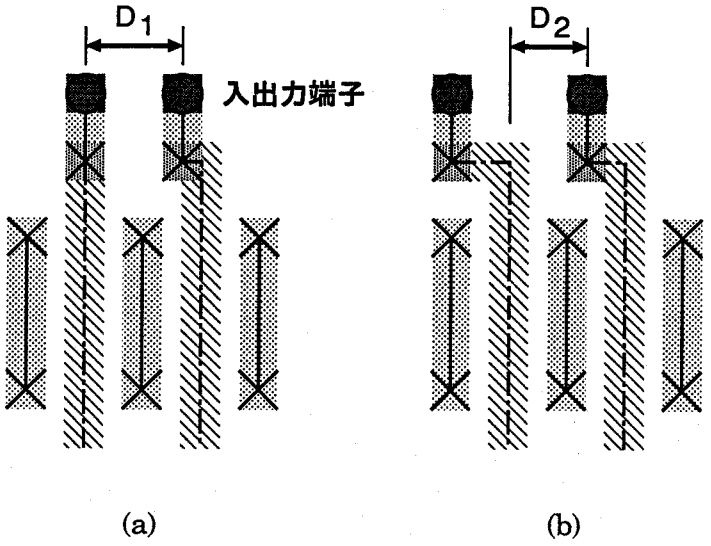


図 3.4: 入出力端子を配置する際の設計規則による制約: (a) 入出力端子間の最小間隔, (b) ゲート端子と第 1 層メタルーポリシリコン間のコンタクトとの間の最小間隔.

ための規則である. この出力端子は, ゲート端子とではなくソース/ドレイン端子と接続されるので, 制約 2 は関係なく, 左隣の入出力端子との設計規則を満足する位置に置かれる. 入出力端子とソース/ドレイン端子との配線はこの段階では行われず, 後述の配線の段階で行われる. 規則 7 は, 規則 6 の条件部が満たされないとき, すなわち設計規則違反が生じていないときに設計規則検証を終えるための規則である.

3.3.3 内部配線

ここでは, 第 1 層メタルを用いてソース/ドレイン端子や入出力端子の間の配線を行う. ソース/ドレイン端子と電源の間も配線する必要があるが, この配線は図 2.1 に示したように, ソース/ドレイン端子から電源へ垂直に第 1 層メタルを伸ばすことにより実現できるので, 規則には記述せず後処理で行う.

図 3.6 に示すように, 配線領域としては, p-MOS および n-MOS トランジスタ列の入出力端子側に近い配線領域 (以後, **外側**と呼ぶ) と電源側に近い配線領域 (以後, **内側**と呼ぶ) を使う. ソース/ドレイン端子と入出力端子を接続するときには必ず外側の配線領域を使い, ソース/ドレイン端子と電源を接続するときには必ず内側の配線領域を使う. また, ソース/ドレイン端子同士を接続するときには, その状況に応じて, 内, 外側の配線領域を使い分ける.

[規則 3]
if （入出力端子の配置を行っている）
and （入出力端子が置かれていない）
then （1番目の入出力端子を $x = 0$ に置く）
（接続すべきゲート端子と接続する）

[規則 4]
if （入出力端子の配置を行っている）
and （入出力端子と接続されていないゲート端子がある）
and （配置が済んだ入出力端子のうち, 最も右にある入出力端子の位置は $x = x_1$ である）
and （入出力端子に接続されているゲート端子のうち, 最も右にあるゲート端子の位置は $x = x_2$ である）
and （入出力端子間の設計規則を満足する最小距離は D_1 である）
and （ゲート端子と第1層メタルーポリシリコン間のコンタクトとの設計規則を満足する最小距離は D_2 である）
and （設計規則検証の要請がない）
then （入出力端子を $x = \max(x_1 + D_1, x_2 + D_2)$ に置く）
（接続すべきゲート端子と接続する）
（設計規則検証を要請する）

[規則 5]
if （入出力端子の配置を行っている）
and （入出力端子と接続されていないゲート端子がない）
and （配置が済んだ入出力端子のうち, 最も右にある入出力端子の位置は $x = x_1$ である）
and （入出力端子間の設計規則を満足する最小距離は D_1 である）
and （設計規則検証の要請がない）
then （入出力端子を $x = x_1 + D_1$ に置く）
（処理を配線に移す）

[規則 6]
if （入出力端子の配置を行っている）
and （設計規則検証の要請がある）
and （最も右にある入出力端子が $x = x_1$ にある）
and （ゲート端子と第1層メタルーポリシリコン間のコンタクトとの設計規則を満足する最小距離は D_2 である）
and （入出力端子と接続されていないゲート端子が $x = x_2 < x_1 + D_2$ にある）
then （このゲート端子を $x = x_1 + D_2$ に置く）
（処理をトランジスタの端子についての設計規則検証に移す）

[規則 7]
if （入出力端子の配置を行っている）
and （設計規則検証の要請がある）
then （設計規則検証の要請を取り除く）

図 3.5: 入出力端子の配置規則.

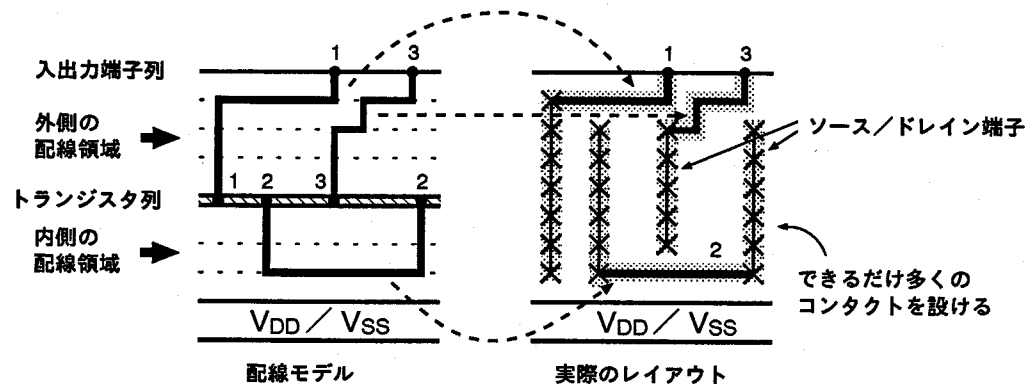


図 3.6: 配線モデルと実際のレイアウト.

2.3節で述べたように, ソース/ドレイン端子となる拡散ー第1層メタル間のコンタクトの抵抗を小さくするために, できるだけ多くのコンタクトを置くことが望ましいので, p-MOS および n-MOS トランジスタ列内では共に第1層メタルの水平方向の配線を出来るだけ上下の境界に詰め, ソース/ ドレイン端子の垂直距離を出来るだけ長くする.

配線の前にまず各ネットについて, それに属するソース/ドレイン端子の数, 入出力端子の数, 全端子の数, 最も左にあるソース/ドレイン端子の名前と位置, 最も左にある入出力端子の位置に関する情報を収集し, それをもとに以下の手順で配線を行う.

1°: 配線する2端子を選び出す.

2°: 配線領域 (p (n)-MOS トランジスタ列の内側か外側) を選定する.

3°: 配線経路を決定する.

これらの操作がすべての端子が配線されるまで繰り返される. 以下に **1°–3°**の各操作の内容を述べる.

3.3.3.1 配線する2端子の選出

この処理では, 配線される2端子 (ソース/ドレイン端子, もしくは入出力端子) が選ばれるが, その選出の規則は図3.7のようになる.

図3.7の規則によると, 各ネットにおいては, ソース/ドレイン端子がすべて配線された後に, 最も右のソース/ドレイン端子と入出力端子が配線される. 単純に左の端子から配線しない理由は, 図3.8 (a) に示すような配線要求に対して, 左の端子から順に配線を行っていくと配線不可

[規則8]
if （配線を行っている）
and （配線される2端子が選ばれていない）
and （最も左にあるソース/ドレイン端子を含むネットに二つ以上のソース/ドレイン端子がある）
then （このネットから, 左から順に二つのソース/ドレイン端子を選ぶ）
（最も左のソース/ドレイン端子をネットから取り除く）

[規則9]
if （配線を行っている）
and （配線される2端子が選ばれていない）
and （最も左にあるソース/ドレイン端子を含むネットにただ一つのソース/ドレイン端子と一つ以上の入出力端子がある）
then （このネットから, ソース/ドレイン端子と最も左にある入出力端子を選ぶ）
（この入出力端子をネットから取り除く）

[規則10]
if （配線を行っている）
and （配線される2端子が選ばれていない）
and （端子を一つしか含まないネットがある）
then （このネットを取り除く）

[規則11]
if （配線を行っている）
and （配線される2端子が選ばれていない）
and （ネットがない）
then （配線処理を終わる）

図 3.7: 2端子の選出規則.

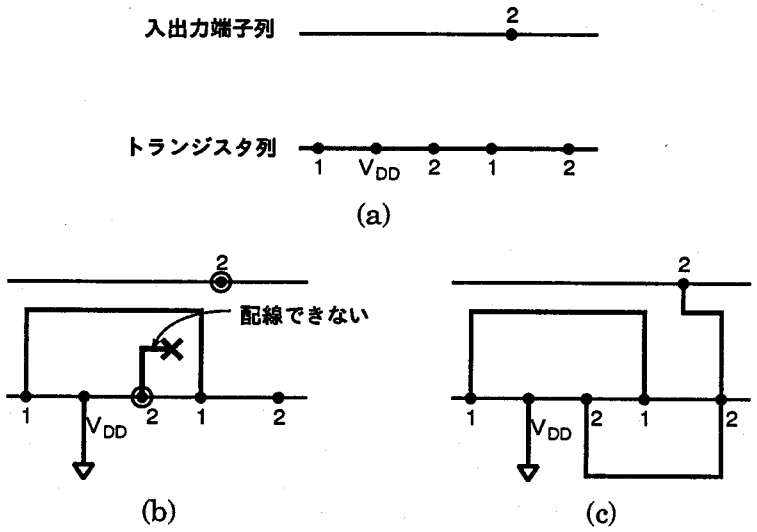


図 3.8: 2 端子の選出規則の違いによって生じる配線の変化: (a) 配線要求, (b) 左の端子から順に 2 端子を選んだ場合, (c) 規則 8–9 を用いた場合.

能な状況に陥るが (図 3.8 (b)), 上記の規則に従えば, このような場合にもすべての端子を配線することができるからである (図 3.8 (c)).

3.3.3.2 配線領域の選定

配線領域を選定する規則を図 3.9 に示し, これらの規則が適用される状況を図 3.10 に示す.

規則 12–15 の条件部を満たすそれぞれの状況において (図 3.10 (a) – (e)), もしその動作部の記述に反した配線領域を選択するとすれば, 以後の段階において必ず配線不可能な状況に陥る. これに対し, 規則 16 および 17 の条件部を満たすそれぞれの状況においては (図 3.10 (f), (g)), その動作部の記述に反した配線領域を選択しても以後に配線不能となるとは限らず, 逆の配線領域を選択することも可能である. そこで, これらの配線領域の選定に関してバックトラッキングを導入している. これは, 規則 16 または 17 により配線領域が選定された後, ある段階で配線が不可能になったならば, その時点で最も新しく選定された (規則 16 または 17 による) 配線領域を変更し, 再び配線を行うものである. これにより, より多くの配線要求を満たすことができる. このバックトラッキングに関する規則を図 3.11 に示し, これらの規則が適用される例を図 3.12 に示す. 図 3.12 (a) では, ネット 1 が規則 17 によって選定された外側の配線領域で配線された後に, 規則 18 によりネット 3 が配線不可能になった状況が検出され, 規則 20 によりネット

[規則 12]
if （配線を行っている）
and （選出された2端子が共にソース/ドレイン端子である）
and （この2端子の間にソース/ドレイン端子があり, それが属するネット中には他にソース/ドレイン端子はないが, 入出力端子がある）
then （内側の配線領域を用いる）

[規則 13]
if （配線を行っている）
and （選出された2端子が共にソース/ドレイン端子である）
and （この2端子の間に, 電源と接続されるソース/ドレイン端子がある）
then （外側の配線領域を用いる）

[規則 14]
if （配線を行っている）
and （選出された2端子のうち, 一方がソース/ドレイン端子で, 他方が入出力端子である）
then （外側の配線領域を用いる）

[規則 15]
if （配線を行っている）
and （選出された2端子が共にソース/ドレイン端子である）
and （この2端子の間に, 既に外側（内側）の配線領域で配線が行われたソース/ドレイン端子がある）
then （内側（外側）の配線領域を用いる）

[規則 16]
if （配線を行っている）
and （選出された2端子が共にソース/ドレイン端子である）
and （この2端子の間にソース/ドレイン端子があり, それが属するネット中には入出力端子がある）
then （内側の配線領域を用いる）

[規則 17]
if （配線を行っている）
and （選出された2端子がある）
then （外側の配線領域を用いる）

図 3.9: 配線領域の選定規則.

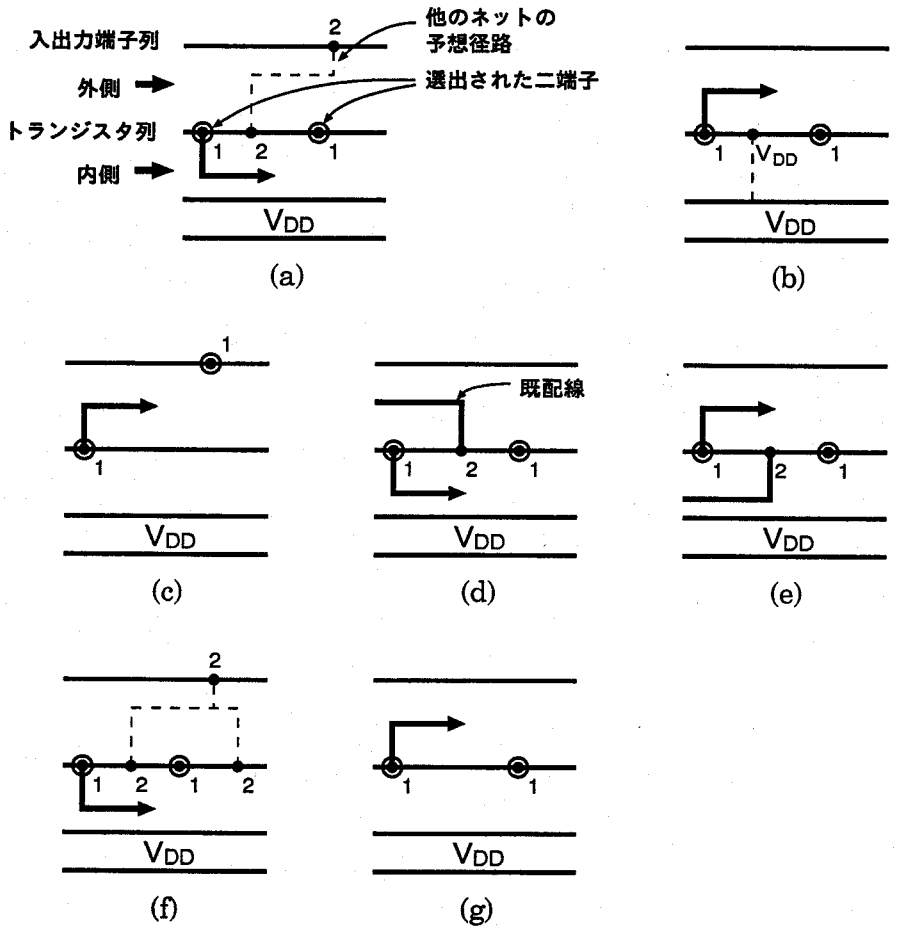


図 3.10: 規則 12–17 が適用される状況: (a) 規則 12 が適用される状況, (b) 規則 13 が適用される状況, (c) 規則 14 が適用される状況, (d) 規則 15 が適用される状況 (その 1), (e) 規則 15 が適用される状況 (その 2), (f) 規則 16 が適用される状況, (g) 規則 17 が適用される状況.

[規則18]
if （配線を行っている）
and （選出された2端子が共にソース/ドレイン端子である）
and （この2端子の間にソース/ドレイン端子があり，それが属するネット中には他にソース/ドレイン端子はないが，入出力端子がある）
and （この2端子の間に，既に内側で配線が行われたソース/ドレイン端子がある）
then （配線をやり直すよう要請する）

[規則19]
if （配線を行っている）
and （選出された2端子が共にソース/ドレイン端子である）
and （この2端子の間に電源と接続されるソース/ドレイン端子がある）
and （この2端子の間に，既に外側で配線が行われたソース/ドレイン端子がある）
then （配線をやり直すよう要請する）

[規則20]
if （配線を行っている）
and （配線をやり直す要請がある）
and （規則16または17によって選定された配線領域で行われた配線がある）
then （この配線以降に行われた配線を白紙に戻して，規則16または17において選定された配線領域（内側または外側）を逆にして配線をやり直す）

図 3.11: バックトラッキングに関する規則.

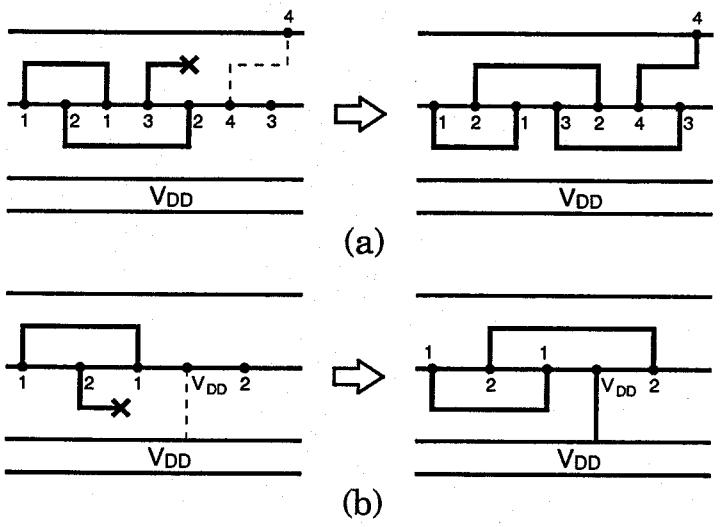


図 3.12: 配線中にバックトラッキングが行われる例: (a) 規則18と20によるバックトラッキングが行われる例, (b) 規則19と20によるバックトラッキングが行われる例.

[規則 21]
if （配線を行っている）
and （選出された 2 端子が共にソース/ドレイン端子である）
and （この 2 端子の間にソース/ドレイン端子があり, それが属するネット中には他にソース/ドレイン端子はないが, 入出力端子がある）
and （この 2 端子の間に電源と接続されるソース/ドレイン端子がある）
then （配線不可能のメッセージを出力する）
（処理を中止する）

[規則 22]
if （配線を行っている）
and （選出された 2 端子のうち, 一方がソース/ドレイン端子で, 他方が入出力端子である）
and （この入出力端子より左に未配線の入出力端子がある）
then （配線不可能のメッセージを出力する）
（処理を中止する）

[規則 23]
if （配線を行っている）
and （配線をやり直す要請がある）
and （規則 16 または 17 によって選定された配線領域で行われた配線がない）
then （配線不可能のメッセージを出力する）
（処理を中止する）

図 3.13: 配線不可能な状況を検出する規則.

1 の配線以降に行われたすべての配線を引き剥して, ネット 1 の配線領域を逆にして再び配線が行われる. その結果, すべてのネットの配線が完了する. また, 同図 (b) では, ネット 2 が配線不可能になった状況が規則 19 によって検出され, 規則 20 によって配線をやり直した結果, すべてのネットの配線が完了する.

図 3.13 の規則 20–23 は配線不可能な状況で適用される規則である.

3.3.3.3 配線経路の決定

配線経路は, 水平線分を可能な限り上下の境界に詰めて割り付けるという方法で決定される. この配線経路の決定は, 図 3.14 の規則 24, 25 の二つの規則によって行われる. ただし, 各配線領域では, 配線のためのトラックが定義されており, 入出力端子列あるいは電源に近いものから昇順に番号が付けられているものとする. また, 配線開始時の始点とは 2 端子のうち, ソース/ ド

[規則24]

if （配線を行っている）
and （選出された2端子があり，配線領域が選定済みである．その始点のx座標はx_0であり，終点のx座標は$x_1 \neq x_0$である）
and （第1層メタル間の設計規則を満足する最小距離はDである）
and （x座標が$x_2 > x_0 - D$で終端している他のネットの配線がトラックiにある）
and （トラック$j > i$にはx座標が$x_3 > x_0 - D$でで終端している他のネットの配線がない）
then （始点からトラックiの一つ内側のトラック$i+1$まで垂直に行き，そこから水平方向に$\min(x_2 + D, x_1)$まで行く配線を行う）
（始点をこの点に移す）

[規則25]

if （配線を行っている）
and （選出された2端子があり，配線領域が選定済みである．その始点のx座標はx_0であり，終点のx座標もx_0である）
then （始点と終点を結線する）
（選出された2端子を取り除く）

図 3.14: 配線経路の決定規則.

レイン端子同士の配線の場合は左側のソース/ドレイン端子，ソース/ドレイン端子と入出力端子との配線の場合はソース/ドレイン端子のことである．

図3.15に各規則が適用される状況を示す．規則24は始点と終点のx座標が異なるときに適用される．まず，始点sから垂直方向に他のネットにぶつかる手前まで，すなわち，他のネットがトラックiにあったとすると，トラック$i+1$まで行く配線を行う．次に，その点から水平方向に

(i) トラックiにあるネットを避けることのできる位置（同図(a)），あるいは

(ii) 終点と同じx座標となる位置（同図(b)）

まで行く配線を行い，始点をその位置に移す．規則24が，始点と終点のx座標が同じになるまで繰り返し適用され，同じになった時点で規則25が適用されて始点と終点が垂直配線により結ばれる（同図(c)）．実際に配線が進む様子を図3.16に示す．

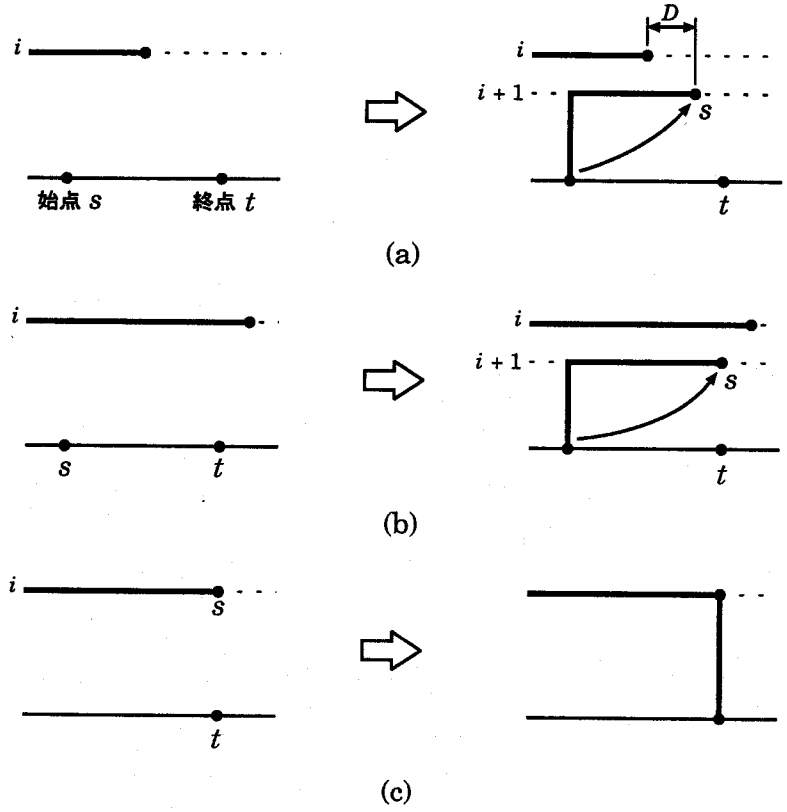


図 3.15: 規則 24, 25 が適用される様子: (a) 規則 24 で, 他のネットが終点 t よりも左側で終端している場合, (b) 規則 24 で, 他のネットが終点 t よりも右側で終端している場合, (c) 規則 25 が適用される様子.

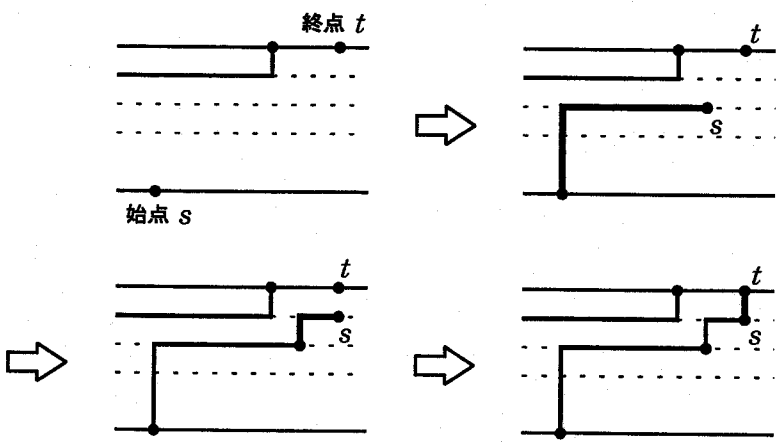


図 3.16: 配線が進む様子.

3.4 実験結果

本章で提案した手法を, MicroVAX II 上で実現し, いくつかの実験データに対して適用した.

マスクパターンの生成において, 適用順序の前後関係がはっきりしている規則に関しては, 作業記憶内にこれらの適用順序を制御するための要素(フラグ)を用意し, これを用いて順序付けを行っている. 例えば, 端子の配置は内部配線に先行すべき処理であるので, 内部配線に関する規則は, 端子の配置が終了し, フラグが配線可能状態に変更されるまでは実行されないようにしている. 規則間の競合解消は, OPS5 のもつ競合解消仕様に従って行われるため, 3.3.1 項の終りに述べたように, 規則の条件部をそれに沿うように工夫している. なお用いた規則の総数は 120 個である.

表 3.1 に本システムを適用した回路 1–4 内のトランジスタの個数, 本システムの処理時間および実行された規則の回数を示す. 適用したすべての実験データに対して面積最小のセルが得られた.

図 3.17 (b) は回路 2 (同図 (a)) に対してトランジスタの配置の順序を (2, 1, 3, 4, 5) と与えて本システムを適用した結果である. この例のように本システムは複数の複合ゲートを含む論理セルも取り扱うことができる.

図 3.18 (b) は, 回路 4 (同図 (a)) に対して, セル上のトランジスタの配置の順序を (1, 2, 6, 3, 7, 4, 8, 5, 9) と与えて本システムを適用した結果である. これは, p-MOS 側, n-MOS 側ともに, 配線が一度で決定されず, バックトラッキングを行って決定された例である.

次に, 設計制約の変更があった場合に, 規則の追加・修正で容易に対応できる例を示す. 図 3.20 (b) は回路 3 (図 3.20 (a)) に対してトランジスタの配置の順序を (1, 2, 3, 5, 4, 6, 7, 8]) と与えて本システムを適用した結果である. 本システムでは 3.3.3 項で述べたように, 内部配線については, できるだけ上下の境界に詰めるように経路を決定しているが, その必要がないとき, 規則 24 を図 3.19 の規則 24′ に変更することにより, 図 3.20 (c) のように 2 回曲がりの経路を得る

表 3.1: 実験結果.

データ名	トランジスタ数	計算時間 (秒)	実行された規則の回数
回路 1	6	17.4	223
回路 2	10	24.3	275
回路 3	16	59.7	510
回路 4	18	67.9	541

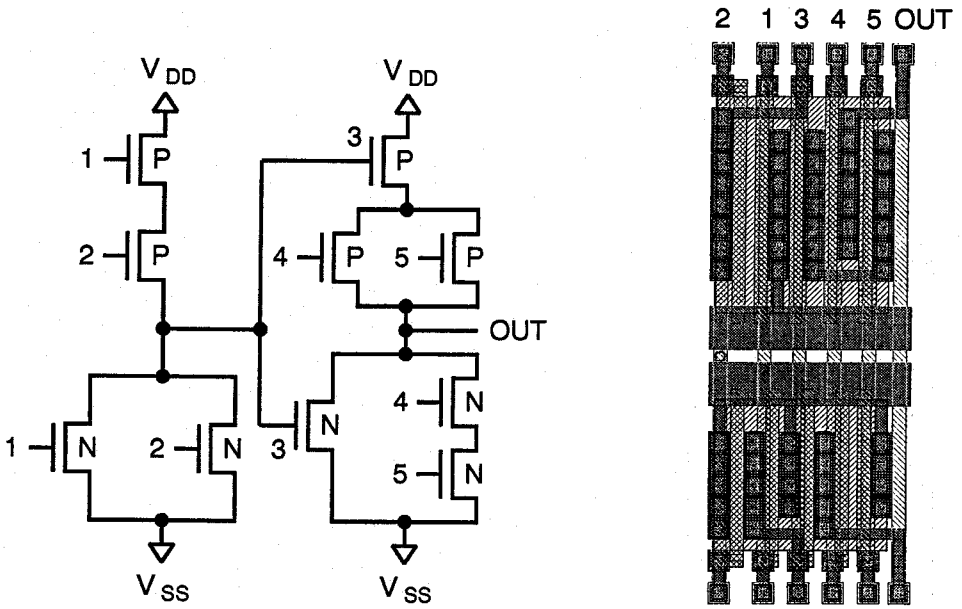
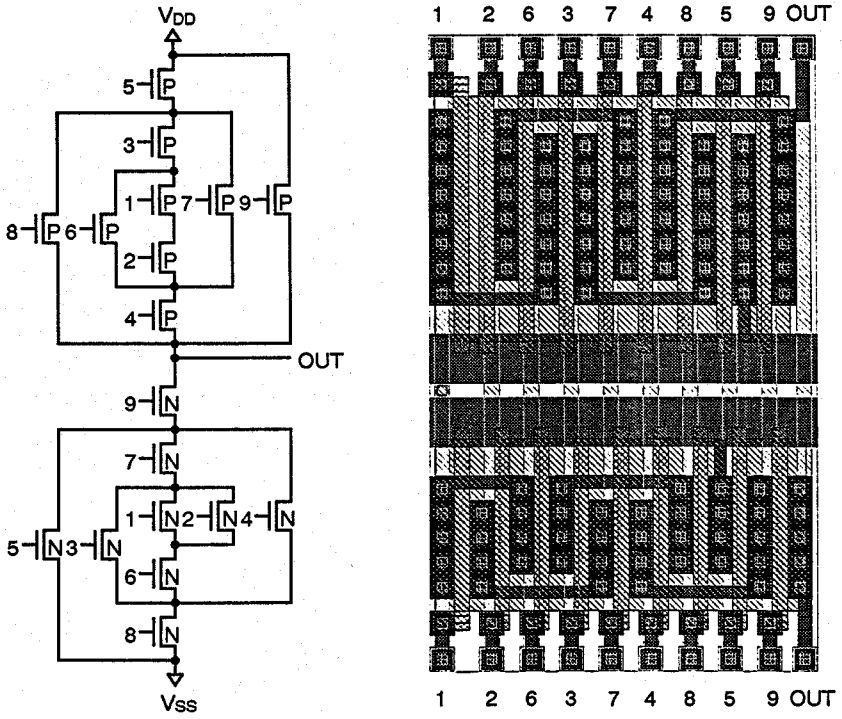


図 3.17: 回路 2 に対して生成されたマスクパターン: (a) 回路図, (b) マスクパターン.



(a) (b)

図 3.18: 回路 4 に対して生成されたマスクパターン: (a) 回路図, (b) マスクパターン.

[規則 24']
if （配線を行っている）
and （選出された 2 端子があり, 配線領域が選定済みである. その始点の x 座標は x_0 であり, 終点の x 座標は $x_1 \neq x_0$ である）
and （第 1 層メタル間の設計規則を満足する最小距離は D である）
and （x 座標が $x_2 > x_0 - D$ で終端している他のネットの配線がトラック i にある）
and （トラック $j > i$ には x 座標が $x_3 > x_0 - D$ で終端している他のネットの配線がない）
then （始点からトラック i の一つ内側のトラック $i+1$ まで垂直に行き, そこから水平方向に x_1 まで行く配線を行う）
（始点をこの点に移す）

図 3.19: 2 回曲がりの配線径路を生成するための規則.

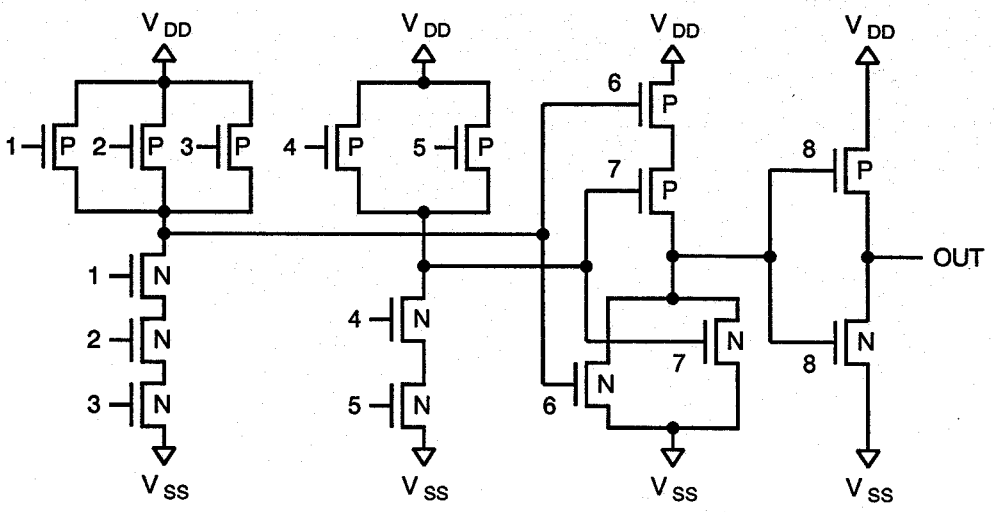
ことができた.

3.5 結言

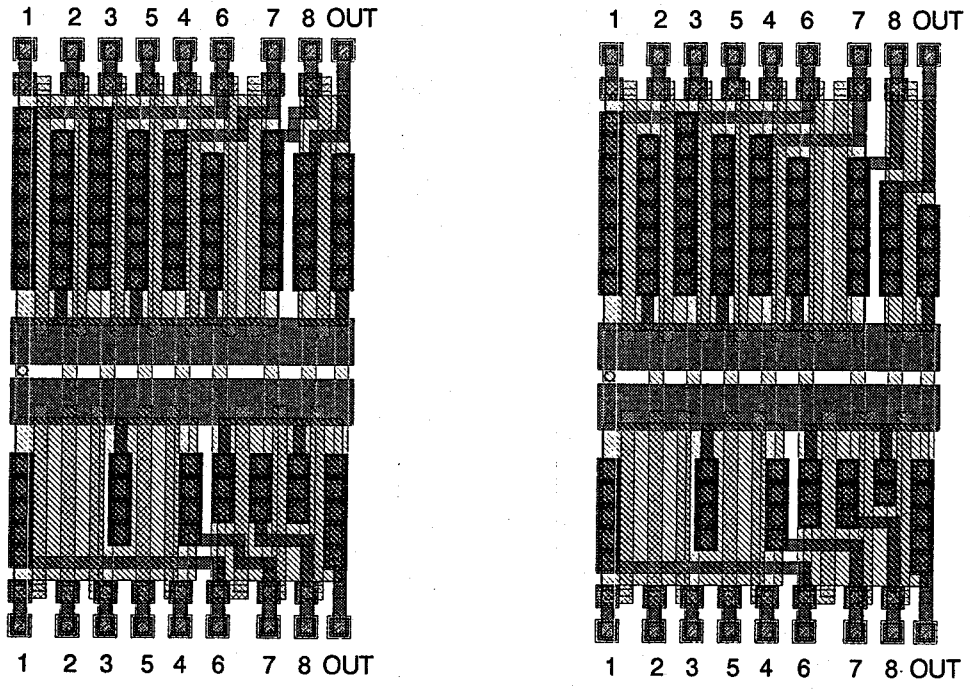
本章では, CMOS トランジスタ回路とそれを実現する論理セル上でのトランジスタの順序が与えられたとき, 指定された設計規則に従ってそのマスクパターンを自動的に生成する手法を提案した. さらに, プログラミング言語 OPS5 を用いてシステムを構築し, 実験を通じて本手法が実用上有効であることを確認した. さらに, 設計制約の変更に対して, 規則を修正することにより容易に対応できることも確認した.

本システムでは, 各段階において設計規則検証を行うための規則を陽に記述しているため, 終了時には必ず設計規則違反がないマスクパターンが得られる.

本システムは, スタンダードセル方式設計やマクロセル方式設計において, ライブラリ中のセルの再設計時に有用となる. さらに, 論理合成ツールと組み合わせた論理セルの自動生成システムを構築することにより, より効果的な設計支援が可能となるが, これについては今後の課題である.



(a)



(b) (c)

図 3.20: 規則の変更による配線径路の変化: (a) 回路図, (b) 規則 24 を用いた場合に生成されたマスクパターン, (c) 規則 24′ を用いた場合に生成されたマスクパターン.

第 4 章

高位合成におけるスケジューリング問題

4.1 緒言

高位合成は, 動作レベルの記述からレジスタ転送レベルの回路を合成する技法である. 高位合成におけるスケジューリング問題は, 動作記述中の演算の実行順序を決定し, 制御ステップに割り当てる問題である. これまで提案されたスケジューリング手法の多くは, ディジタル信号処理で用いられるフィルターのように, 比較的単純な動作を対象とするものが多かった[14, 15]. しかし, より一般的なシステムに適用するためには, ループや条件分岐といった複雑な動作を扱う必要がある. 本研究では, 条件分岐がある場合の最も一般的なスケジューリング問題について考察する.

条件分岐がある場合のスケジューリングでは, 条件分岐中の演算間で資源をいかにして共有するかということがコストを最小化する上で重要な鍵となる. 条件分岐中の演算間で資源を共有できるかどうかは, これらの演算と分岐条件を判断する演算 (条件演算) の先行関係によって決まる. したがって, 条件分岐中の二つの演算が異なる実行条件で実行されるとき,

(i) 条件演算が既に終了していれば, 分岐条件が確定しているので一つの演算器を共有できる.

(ii) 条件演算がまだ終了していなければ, 分岐条件が確定していないので一つの演算器を共有できない.

という二つの事項を考慮に入れなければならない. しかし, これまでに提案された多くの手法は, まず条件演算を行った後, その結果に従って分岐中の演算を実行するという前提のもとでスケジューリングを行っていたため[16–24], 必ずしも最適解が得られるとは限らなかった. さらに, 上記 (i), (ii) を考慮した手法[25, 26]が提案されたが, 扱える条件分岐の構造に制限があったり,

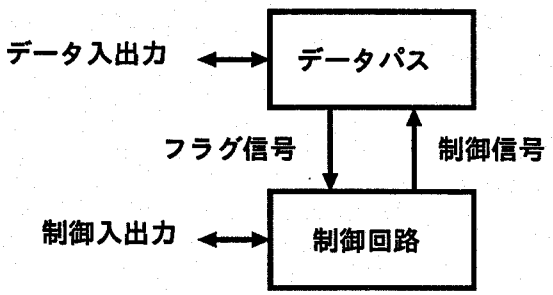


図 4.1: データ処理装置のアーキテクチャモデル.

ヒューリスティック手法を用いているために最適解が得られるとは限らなかった.

本論文で提案する手法[32-38]は, このような前提を置かずに, 最も一般的なスケジューリング問題に対処しようとするものである. 本手法の特徴は, スケジューリング問題を整数非線形計画問題として定式化し, これに対する効率的な算法によって最適解を得ようとすることにある.

本章では, まず, 高位合成に関わるスケジューリング問題について概説し, 次に動作記述に条件分岐がある場合を含めた最も一般的なスケジューリング問題を定式化する.

4.2 高位合成の概要

機能設計を自動化することを目的とする高位合成は, ハードウェアの動作記述から, 論理合成で扱うことができるレジスタ転送レベルの回路を自動的に合成するものでなければならない. 以下では, 設計対象としてデータパスと制御回路から構成されるデータ処理装置 (図 4.1) を考える. データパスは, 加算器や乗算器等の演算器, レジスタや RAM 等の記憶ユニット, およびマルチプレクサ, バスなどの接続ユニットからなり (以後, これらを総称して**資源**と呼ぶことにする), 主にデータの演算を行う. 制御回路は, 順序回路であり, 制御用の入力とデータパス中の状態を示すフラグ信号を入力として状態遷移を行い, データパス中の資源を制御する信号を出力する.

このようなデータ処理装置を合成する際の処理手順は概ね以下のようになる.

1. コンパイル
2. スケジューリング
3. アロケーション
4. 制御回路の合成

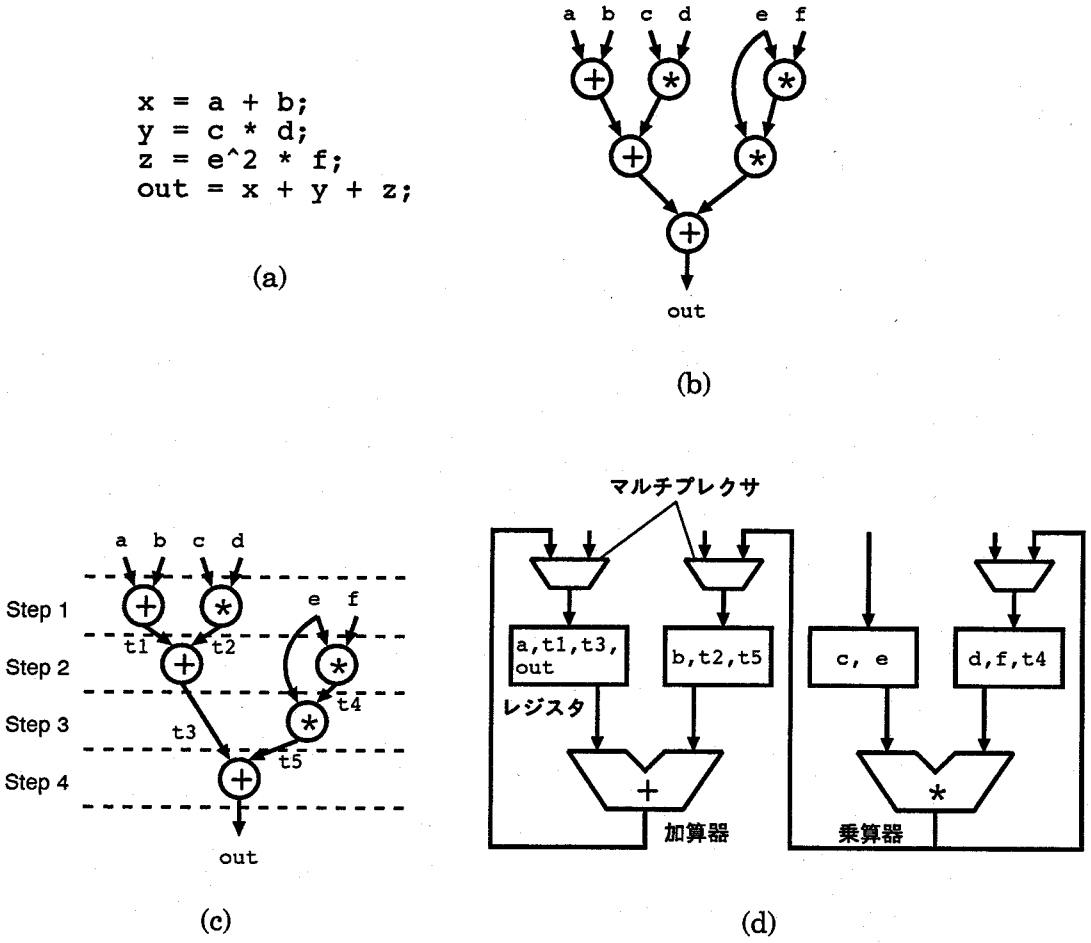
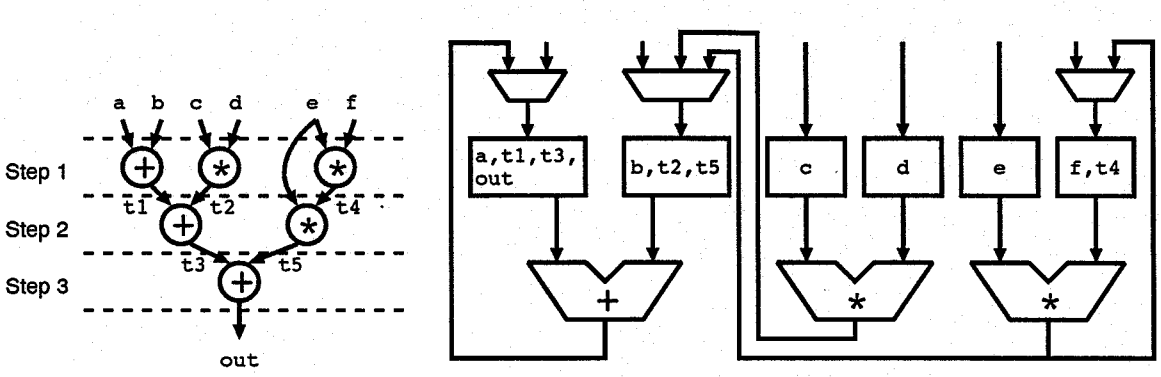


図 4.2: 高位合成の例: (a) 動作記述, (b) データフローグラフ, (c) スケジューリング例, (d) データパスの構成例.

これらの処理を図 4.2 を用いて簡単に説明する.

まず, 与えられた動作記述 (同図 (a)) を計算機の内部表現へと変換する (コンパイル). 内部表現としては, 同図 (b) のようなデータフローグラフがよく用いられる[14]. このグラフでは, 節点が動作記述中の演算を表し, 節点間の有向枝は対応する演算間のデータの授受を表している.

次に, 動作中の各演算を実行する制御ステップを決定する (スケジューリング). 1 制御ステップは制御回路の 1 状態に対応し, 単相クロックの場合は 1 クロックに相当する. 以後, 混乱の恐れがない限り, 制御ステップのことを単にステップともいうことにする. 正しい演算結果を得るためには, あるデータを使用する演算は, そのデータを生成する演算が終了した時点以降でなければ実行できない. スケジューリングを行う際にはこのような演算間の先行関係を壊さないように各演算の実行順序を決定する必要がある. 図 4.2 (c) は, 同図 (b) のデータフローグラフに



(a) (b)

図 4.3: 図 4-2 (a) の動作記述に対する別の実現例: (a) スケジューリング, (b) 構成されたデータパス.

対する一つのスケジューリングの例である.

このスケジューリングの結果, データパスを構成するために必要な演算器や記憶ユニットの個数が決るので, 接続コストが小さくなるように, 演算を演算器に, 記憶すべきデータを記憶ユニットに割り当て, これらの間を接続するのがアロケーションである. 図 4.2 (c) のスケジューリングの結果をもとに構成したデータパスを同図 (d) に示す.

最後に, スケジューリングの結果得られた順序で状態を遷移し, データパス中の資源を制御する信号を出力するような順序回路を合成することによって制御回路を得る.

このようにして, 動作記述からレジスタ転送レベルの回路が合成される.

4.3 スケジューリング問題

スケジューリングによって, 処理に必要なステップ数や, データパスのコストの大きな要因となる演算器や記憶ユニットの個数が決定されるので, スケジューリングは高位合成の中でも非常に重要な処理ということができる[14].

スケジューリングでは, ステップ数およびデータパスのコストを最小化することが望ましいが, ステップ数を少なくしようとすると, データパスのコストが増え, 逆にデータパスのコストを減らそうとすると, ステップ数が増えることがある. 例えば, 図 4.2 (a) の動作記述に対して, 3 ステップで処理を行うようなスケジューリングを得ようとすると, 図 4.3 (a) のようになり, この場合, 1 個の加算器と 2 個の乗算器が必要になり, またレジスタも 6 個必要になるため (同図 (b)), 図 4.2 (d) に比べてデータパスのコストは増加する.

このように, ステップ数とデータパスのコストの両方を同時に最小化できないので, 通常は, ステップ数とデータパスのコストのどちらか一方を制約として与えて, 他法を最小化するという問題として定式化する. ステップ数を制約として与えて, データパスのコストを最小化する問題を, **時間制約型スケジューリング**と呼び, データパスのコストを制約として与えて, ステップ数を最小化する問題を, **資源制約型スケジューリング**と呼ぶことにする. データパスのコストとしては, 通常, 資源の個数や面積等が用いられる.

4.4 条件分岐がある場合のスケジューリング問題

条件分岐を含むハードウェアの動作の表現法として, 4.2節で述べたデータフローグラフに条件分岐を表すための節点を加えた**コントロール・データフローグラフ**（Control Data-Flow Graph）[39]（以後 **CDFG** と呼ぶ）を考える. この CDFG は3種類の節点（**演算節点**, **分岐節点**, **マージ節点**）と 3 種類の有向枝（**S-枝**, **P-枝**, **W-枝**）からなり, それらは以下のように定義される.

[コントロール・データフローグラフの定義]

節点: **(n1)** 演算節点（図中の○で示された節点）は, 動作記述中の演算を表す.

(n2) 分岐節点（図中の △ で示された節点）は, 動作記述中の条件分岐の始まりを表す.

(n3) マージ節点（図中の ▽ で示された節点）は, 条件分岐内で生成された一つの値を表す.

枝: **(e1)** S-枝（図中の細線で示された枝）は, 二つの節点間の値の授受を表す. 演算節点間だけではなく演算節点と分岐節点あるいはマージ節点との間にも S-枝が設けられる.

(e2) P-枝（図中の太実線で示された枝）は, 分岐節点と対応するマージ節点の間に設けられる.

(e3) W-枝は（図中の破線で示された枝）は, 条件分岐中の演算が, 条件が真のときに実行されるパス（以後, **真のパス**と呼ぶ）中にあるか, 偽のときに実行されるパス（以後, **偽のパス**と呼ぶ）中にあるかを表す. W-枝にはその枝が真・偽どちらのパスを表すかを示すために, 真のパスを表すときは T, 偽のパスを表すときは F の属性をつける.

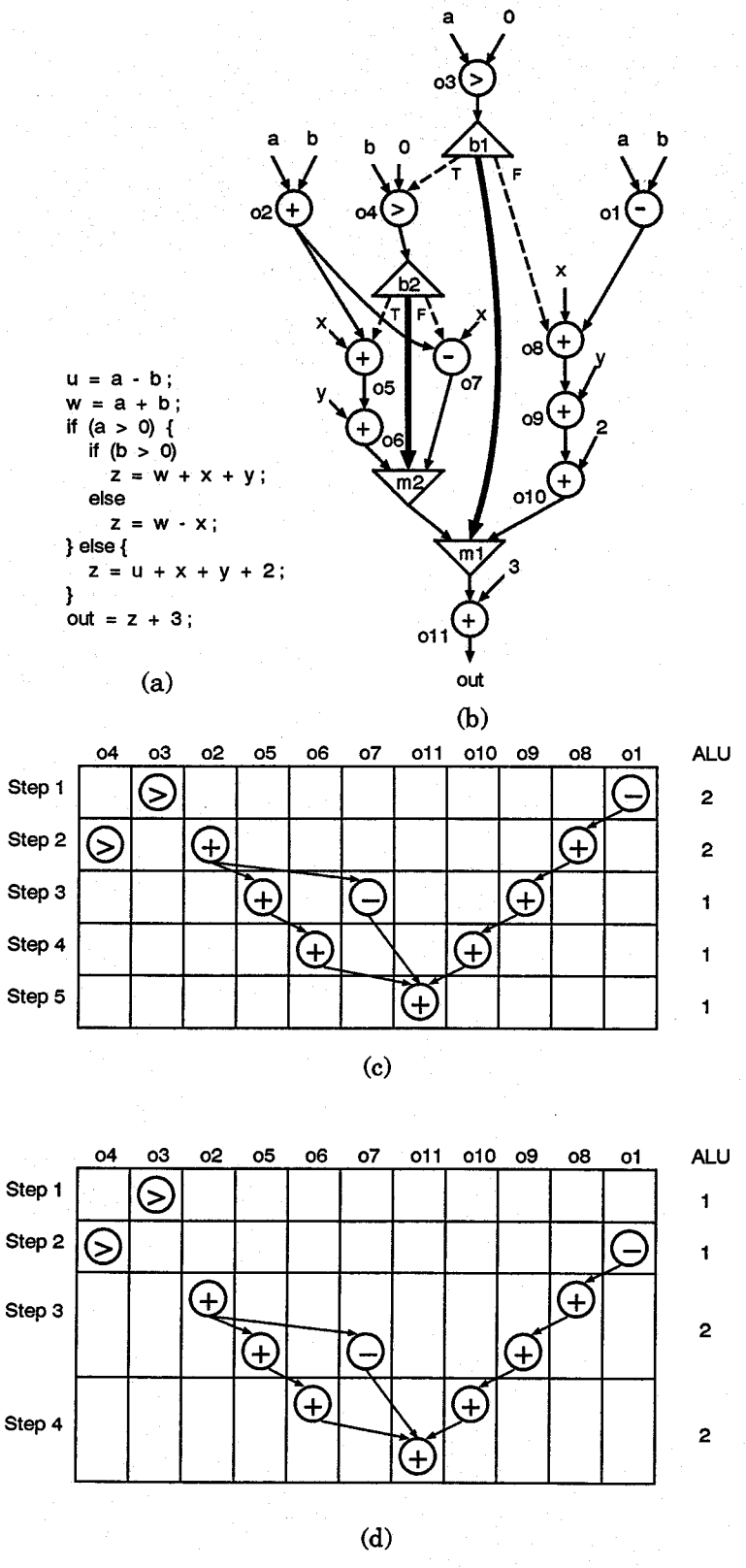


図 4.4: コントロール・データフローグラフ（CDFG）とスケジューリングの例: (a) 動作記述, (b) CDFG, (c) スケジューリング結果（制御ステップ数 5, ALU 数 2), (d) スケジューリング結果（演算のチェインを行う場合）.

例えば, 図 4.4 (a) の動作記述に対する CDFG は同図 (b) のようになる.

以下では, o_i は演算およびそれを表す演算節点の両方を表し, b_i は条件分岐およびそれを表す分岐節点の両方を表すことにする. また, 条件分岐 b_i の分岐条件を決定する演算のことを条件分岐 b_i の条件演算と呼ぶことにする.

図 4.4 (b) の CDFG に対して, すべての演算を ALU で実行することにして, スケジューリングを行って, 例えば図 4.4 (c) のような結果が得られたとする. ただし, 図 4.4 (c) の各列は CDFG 中の演算を表し, 各行がステップを表す. 図 4.4 (c) のステップ 4 に注目すると, このステップに割り当てられた演算は o_6 と o_{10} であるから, 一見 ALU は 2 個必要になるように見える. しかし, o_6, o_{10} はそれぞれ, 条件分岐 b_1 の真のパス, 偽のパス中にあり, しかも条件分岐 b_1 の条件演算 o_3 はステップ 1 で実行済みで, すでに分岐条件が確定しているため, 二つの演算を同時に実行する必要はなく, したがって 1 個の ALU で実行することができる.

二つ以上の演算が同時に実行されないとき, これらの演算は**互いに排他的**であるといい, これらの演算を**相互排他演算**と呼ぶ. 同じ種類の演算器で実行される相互排他演算は 1 個の演算器を共有することができ, 必要な演算器数を減らすことができる. 条件分岐がある場合には, 条件分岐中の演算間でいかに演算器を共有するかということが重要な問題となる.

これまでに提案された条件分岐を扱う多くの手法は, 問題を簡単にするために, 条件分岐が始まる前に必ず条件演算を実行するという前提を置いて条件分岐を扱ってきた[16–24]. このような前提のもとでは, 実行条件が異なる演算間で必ず演算器の共有ができ, 必要となる演算器の個数を少なくすることができるが, 条件演算の実行が終るまで分岐中の演算を行うことができず, ステップ数が増えてしまうことがある.

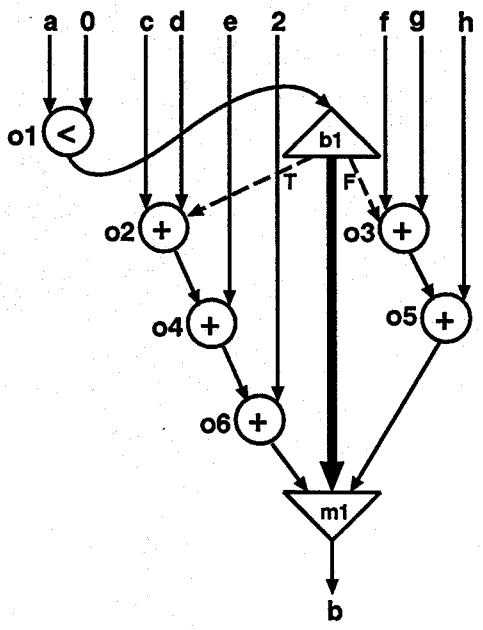
例えば, 図 4.5 (b) の CDFG に対するスケジューリングを考える. ここでは, '+' を加算器で, '<' を比較器で実行するものとする. 条件分岐中の演算よりも条件演算が先行した場合のスケジューリングは同図 (c) のようになる. この場合, o_2 と o_3, o_4 と o_5 はそれぞれ互いに排他的となり, 1 個の加算器を共有できる. しかし, 条件演算 o_1 を先に実行しておく必要があるため, 必要となるステップ数は 4 となる.

次に, 条件演算と条件分岐中の演算を並行して実行する場合 (同図 (d)) を考える. この場合, 同じ処理を 3 個のステップで実現できる. ただし, ステップ 1 では, 条件演算 o_1 がまだ終了しておらず, 条件分岐 b_1 の分岐条件を知ることができないために, o_2 と o_3 は加算器を共有することができず, 2 個の加算器が必要となる. この例に対する最適なスケジューリングは, o_3 と o_5 をそれぞれステップ 2 とステップ 3 に割り当てることにより得られる (同図 (e)).

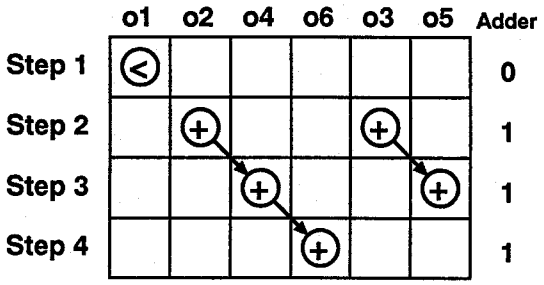
このように, 演算間で演算器を共有できるかどうかは条件演算と条件分岐中の演算の先行関

```
if (a < 0)
  b = c + d + e + 2;
else
  b = f + g + h;
```

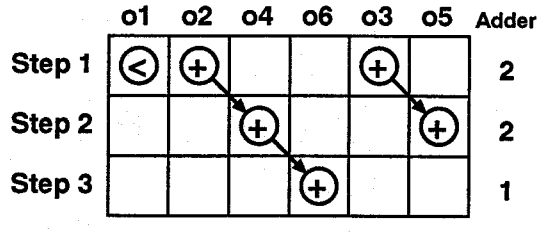
(a)



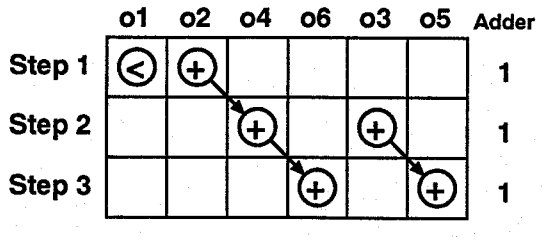
(b)



(c)



(d)



(e)

図 4.5: スケジューリング例: (a) 動作記述, (b) CDFG, (c) 条件演算を条件分岐よりも先行して実行する場合, (d) 条件演算と条件分岐中の演算を同時に実行する場合, (e) 最適なスケジューリング.

係に依存し, 次のことがいえる:

(i) 条件演算が終了しているステップでは, 異なる実行条件の演算間では演算器の共有が可能.

(ii) 条件演算が終了していないステップでは, 異なる実行条件の演算間でも演算器の共有は不可能.

したがって, 条件分岐がある場合に最適なスケジューリングを得ようとすれば, 演算間での演算器の共有を考えながら, 条件演算や条件分岐中の演算の実行ステップを決定する必要がある.

文献 [25] は, 条件ベクタ[24]という概念を導入して, 初めてこの問題に対する解法を試み, 文献 [26] は, 各実行パスを木で表現し, その木の上で各実行パスを最適化するという手法を提案した. これらの方法は従来の手法に比べて, 良い解を得ることができたが, ヒューリスティック手法であるために, 最適解が保証されるとは限らなかった. また, これらの二つの手法は資源制約型スケジューリングであり, 本研究が時間制約型スケジューリング問題を一般的に扱った最初の手法である.

本研究で考察する時間制約型スケジューリング問題は次のように定式化される.

[スケジューリング問題]

> 条件分岐を含む動作が CDFG で与えられ, ステップ数が制約として与えられたとき, 演算器のコストを最小にするスケジューリングを求めよ.

本論文では, 簡単のために, 以下のような前提を置く:

1. 各演算を実行する演算器の種類は既に決定されている.
2. 演算は 1 ステップ以内に終了する (複数のステップにまたがる演算やパイプライン演算は考えない).
3. 動作にループは含まれない.

ここで, 特に, 2. に関連して, 1 ステップ内で複数の演算を続けて実行する**演算のチェイン**は容易に扱えることを付言する. 例えば, 1 ステップの長さが 100ns であるとき, 実行 (遅延) 時間が 40ns の ALU を使えば, 1 ステップ内で最大二つの演算を実行でき, この場合にはチェイン数が 2 であるということができる. また, 図 4.4 (b) の CDFG に対してチェイン数を 2 としてスケジューリングを行えば, 図 4.4 (e) のようにステップ数を 4 とすることができる.

4.5 結言

本章では, 高位合成におけるスケジューリング問題について述べた. まず, 高位合成に関わるスケジューリング問題について概説した後, 動作記述に条件分岐がある場合を含めたスケジューリング問題を定式化した.

第 5 章

整数計画法に基づくスケジューリング手法

5.1 緒言

本章では, 前章で定義した時間制約型スケジューリング問題に対する最適スケジューリング手法[32-38] について考察する.

文献 [19] で提案された時間制約型スケジューリング手法は, スケジューリング問題を整数線形計画問題として定式化し, 条件分岐がない場合に最適解を得ることができるが, 複雑な条件分岐を扱うことができない. 文献 [40] は, この計画問題において演算の制御ステップへの割当を表す変数が 0, 1 の値をとることに注目して制約式をブール式で表し, 二分決定グラフを用いて効率良く解を求めているが, これも条件分岐を扱うことはできない.

本章では, 条件分岐がある場合のスケジューリング問題を整数非線形計画問題として定式化し, これに対する最適化手法を考察する.

動作に条件分岐が含まれる場合には, スケジューリングに際して,

(i) 演算間の相互排他性

(ii) 演算間の先行関係

という二つの性質を考慮する必要があるが, 与えられた CDFG からこれらの情報を直接見い出すことはできない. そこで (i) については, CDFG 中の各節点に実行条件を表す**タグ**を付け, 各演算の実行条件を明らかにし, (ii) については, CDFG から演算の先行関係だけを抽出して**先行関係グラフ**と呼ぶグラフを生成する. このタグ情報と先行関係グラフを用いて定式化される整数非線形計画問題を, 文献 [40] と同様, 制約式をブール式で表現し, 二分決定グラフを用いて充足問題として解くことにより, すべての解を効率良く列挙することができる.

本章では, まず, 条件分岐がある場合にタグを用いて各演算の実行条件を識別する方法につい

て述べ, 次に, 入力の CDFG から先行関係を抽出して, 先行関係グラフを生成する方法について述べる. 次に, 整数非線形計画問題への定式化とその解法について述べ, 最後に, 実験を通じて本手法の有効性を示す.

5.2 演算の実行条件の決定

図 4.4 (b) の演算 o_8 や o_9 のように条件分岐 b_1 の偽のパス中にある演算は, 条件演算 o_3 の結果が偽であるときに実行しなければならないことは明らかである. しかし, CDFG をよく観察すれば, データの依存関係により, 条件分岐中にはないが, ある実行条件でだけその演算結果が必要となる演算が存在する場合もある. 例えば, 同図の演算 o_1 の実行結果は o_8 でのみ使用されるので, 条件分岐 b_1 の真のパス中の演算で使用されることはない. したがって, もし演算 o_1 が条件分岐 b_1 中の演算 o_7 と同じステップに割り当てられたとしても, 条件分岐 b_1 の条件演算 o_3 の結果がそのステップまでに確定しているならば, 同じ ALU を共有することができる. 演算間の資源の共有を最大限行うためには, このような条件分岐の外の演算間についても, 実行条件の排他性を考慮する必要がある.

本手法では, 各演算の実行条件を明確にするために, 文献 [39] で提案されているように, CDFG 中の各節点 v に実行条件を表すタグ $tag(v)$ を付けることにする. CDFG 中の条件分岐の集合を $\{b_1, b_2, \ldots, b_q\}$ とすると, このタグ $tag(v)$ は $\{1, \overline{1}, 2, \overline{2}, \ldots, q, \overline{q}\}$ の部分集合として表される. CDFG 中の節点にタグを付ける手続き Tagging を図 5.1 に示す. ここでは, S-枝あるいは P-枝 (x, y) の始点 x (終点 y) を, 終点 y (始点 x) の親 (子) と呼んでいる.

図 5.2 に図 4.4 (b) の CDFG に対して手続き Tagging を適用した結果を示す. 図 5.2 (a) は 1°が終った時点での各節点の初期タグを示し, 図 5.2 (b) は各節点のタグを示す.

1°では分岐節点の真のパス, あるいは偽のパス中にある節点の実行条件が決定され, 節点 v の実行条件は初期タグ $tagin(v)$ によって表される. 例えば, 図 5.2 (a) の場合, $tagin(b_2) = \{1\}$, $tagin(o_5) = \{1, 2\}$ となる. o_1, o_2, o_3, o_{11} はどの条件分岐中にもないので, $tagin(o_1) = tagin(o_2) = tagin(o_3) = tagin(o_{11}) = \emptyset$ となる.

2°, 3°では, データ依存関係も考慮して, 各節点の最終的な実行条件が決定され, 節点 v の最終的な実行条件がタグ $tag(v)$ によって表される.

まず, 2°では, 子のない節点のタグが決定される. 子のない節点については, データ依存関係によって実行条件が変化しないので, そのタグは初期タグと同じである. ここで, 子のない節点は条件分岐中にもないので (条件分岐中にあれば少なくともマージ節点を子に持つ), 必ずその節

0°: $\{b_1, b_2, \ldots, b_q\}$ を CDFG 中の条件分岐の集合とする.

1°: 各節点 v に対して以下のように定義される初期タグ $tagin(v)$ を付ける.

1. $tagin(v)$ は $\{1, \overline{1}, 2, \overline{2}, \ldots, q, \overline{q}\}$ の部分集合である.
2. 節点 v が分岐節点 b_k に関する真(偽)のパスに含まれるとき, かつそのときに限り $k(\overline{k}) \in tagin(v)$ である.

2°: 節点 u の子がないとき, 節点 u に, $tag(u) \triangleq tagin(u) = \emptyset$ で定義されるタグ $tag(u)$ を付ける.

3°: タグが付けられていない各節点 u に対して次の処理を再帰的に適用する.

- 節点 u のすべての子 $v, w, \ldots, z$ のタグが付けられているならば, $tag(u) \triangleq tagin(u) \cup (tag(v) \cap tag(w) \cap \ldots \cap tag(z))$ で定義されるタグ $tag(u)$ を付ける.

図 5.1: 手続き Tagging.

点は空となることに注意しよう. 図 5.2 (b) の例では, 節点 o_{11} には子がないので, $tag(o_{11}) = \emptyset$ となる.

次に **3°**では, すべての子のタグが決定された節点のタグを決定するという操作を再帰的に適用することにより, すべての節点のタグを決定する. 例えば図 5.2 (b) で, o_5, o_7, o_8 のタグがそれぞれ $tag(o_5) = \{1, 2\}$, $tag(o_7) = \{1, \overline{2}\}$, $tag(o_8) = \{\overline{1}\}$ と決まったとすると, $tag(o_1) = tagin(o_1) \cup tag(o_8) = \emptyset \cup \{\overline{1}\} = \{\overline{1}\}$, $tag(o_2) = tagin(o_2) \cup (tag(o_5) \cap tag(o_7)) = \emptyset \cup (\{1, 2\} \cap \{1, \overline{2}\}) = \{1\}$ と決定される.

さて, CDFG に対する Tagging が終了した時点では, 演算節点 v について, $i\,(\overline{i}) \in tag(v)$ ならば,

(i) v に対応する演算が条件分岐 b_i の条件演算が終了した後のステップに割り当てられれば, その演算は条件分岐 b_i の分岐条件が真(偽)のときに実行されなければならず, それ以外のときは実行される必要がない.

(ii) v に対応する演算が条件分岐 b_i の条件演算が終了する前のステップに割り当てられれば, その演算は条件分岐 b_i の分岐条件に関わらず必ず実行されなければならない.

ということを意味する. したがって, 二つの演算節点 u, v のタグが $i \in tag(u)$, $\overline{i} \in tag(v)$ であ

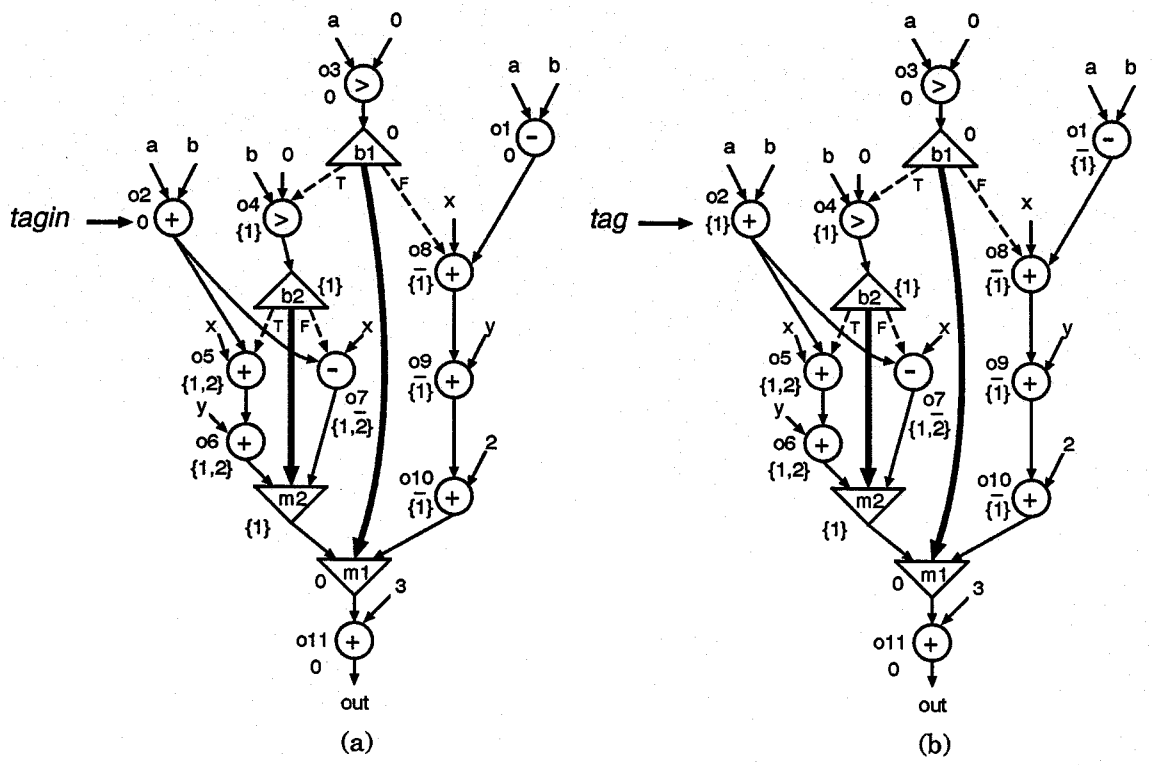


図 5.2: 図 4.3 の CDFG に対するタグ付け: (a) 各節点の初期タグ, (b) 各節点のタグ.

るとき, 条件分岐 b_i の条件演算が既に終了しているステップでは, u, v に対応する二つの演算は互いに排他的であることがわかる.

5.3 先行関係グラフの生成

スケジューリングでは演算間の先行関係を考慮しなければならないが, 条件分岐がある場合に CDFG から直接演算間の先行関係を見い出すことができない. そこで本手法では, CDFG から演算間の先行関係のみを抽出して, 以下で定義されるような先行関係グラフ $G = (V, E)$ を作成する.

[先行関係グラフの定義]

(I) 節点の集合 V は CDFG 中の演算節点の集合と同じである (CDFG の場合と同様, o_i は演算およびそれを表す節点の両方を表すことにする).

(II) 枝の集合 E は次のような互いに素な二つの部分集合 F と H からなり, それぞれ次のように定義する.

(a) 演算 o_u で生成された値が演算 o_v で使われるときかつそのときに限り, 有向枝 $(o_u, o_v) \in F$ を設け, $o_u \rightarrow o_v$ と書くことにする. 節点 o_u から F に属する枝だけを通って節点 o_v に到達可能であるとき, すなわち, $o_u \rightarrow o_x \rightarrow o_y \rightarrow \ldots \rightarrow o_v$ であるとき, o_v を o_u の子孫と呼ぶ.

(b) 以下のいずれかの条件が成り立つときかつそのときに限り, 節点 o_u と o_v の間に有向枝 $(o_u, o_v) \in H$ を設け, $o_u \Rightarrow o_v$ と書くことにする.

(i) 演算 o_v は, 演算 o_u と同じ制御ステップで実行できない子孫のうち, 最も近い子孫である. すなわち, 節点 o_v は, 節点 o_u を始点とし F に属する枝だけからなる有向道 $o_u \rightarrow o_x \rightarrow o_y \rightarrow \ldots \rightarrow o_v$ を考え, o_u, o_x, o_y, $\ldots$ という順に演算の実行時間を加算していったとき, その和が 1 ステップの時間を初めて越える子孫である.

(ii) 演算 o_v は, 演算 o_u を条件演算とする条件分岐で生成された値を使用する.

先行関係グラフにおいて, $o_u \rightarrow o_v$ であれば, o_v を o_u と同じステップ, または o_u より後のステップに割り当てなければならないことを表し, $o_u \Rightarrow o_v$ であれば, o_v が o_u より後のステップに割り当てられなければならない (同じステップに割り当ててはいけない) ことを表す.

先行関係グラフの作り方の例を, 図 5.3 に示す. 矢印の左側が与えられた CDFG で, 右側がチェイン数を 2 としたときの先行関係グラフである. 図では F に属する枝を細い実線で, H に属する枝をを太い実線で示している.

図 5.3 (c) で, 先行関係グラフの H に属する枝は定義 (b-i) に従って設けられた枝である. 図 5.3 (d) では, 演算 o_2 は, o_1 を条件演算とする条件分岐で生成された値を使用するため, 定義 (b-ii) に従って o_1 と o_2 の間に H に属する枝が設けられている.

図 5.4 は図 4.4 (b) の CDFG に対する先行関係グラフである. 図 5.4 (a), (b) はそれぞれ, 演算のチェインをしない場合, チェイン数を 2 とした場合を表している. 同図 (a) では, $o_u \rightarrow o_v$ かつ $o_u \Rightarrow o_v$ の場合は, $o_u \rightarrow o_v$ は省略している. 例えば $o_8 \rightarrow o_9$ かつ $o_8 \Rightarrow o_9$ なので, $o_8 \Rightarrow o_9$ だけを描いてある.

一つの条件分岐に対して二つの条件演算がある場合がある. 例えば, 図 5.5 (a) の場合, 条件分岐 b_2 の条件演算は o_1, o_2, o_3 の三つある. このような場合, 定義に従って作成すると先行関係

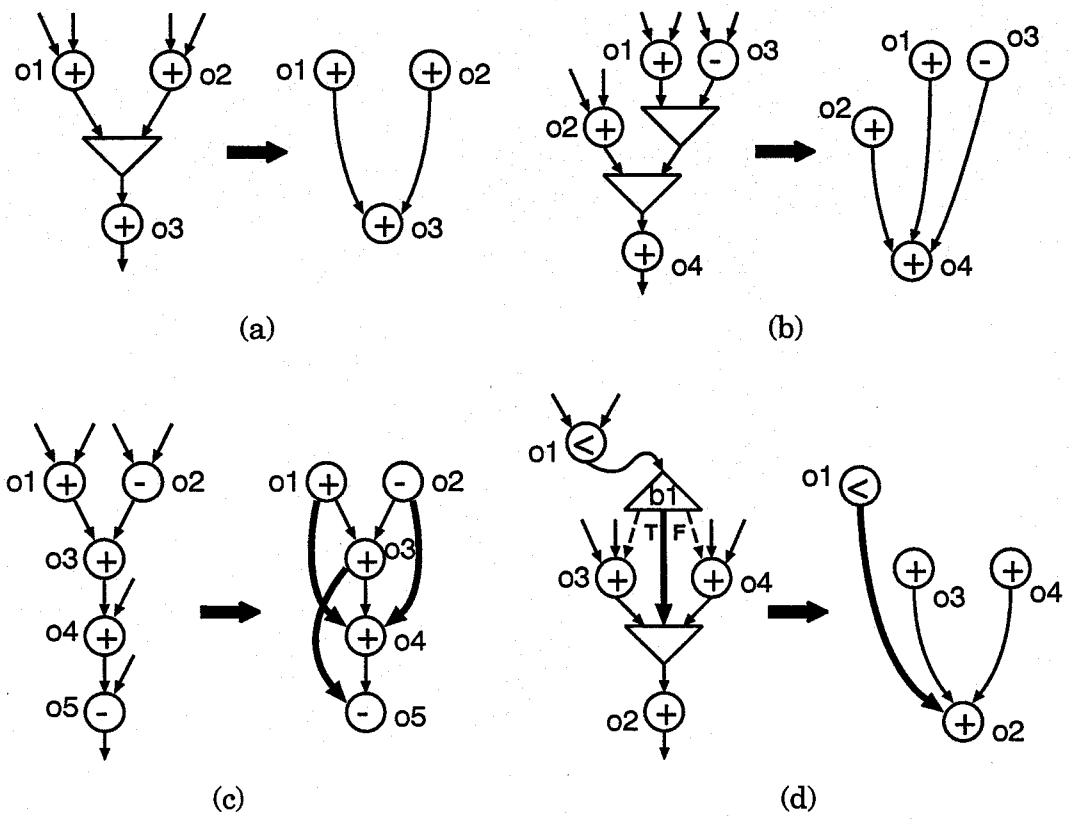


図 5.3: 先行関係グラフの例.

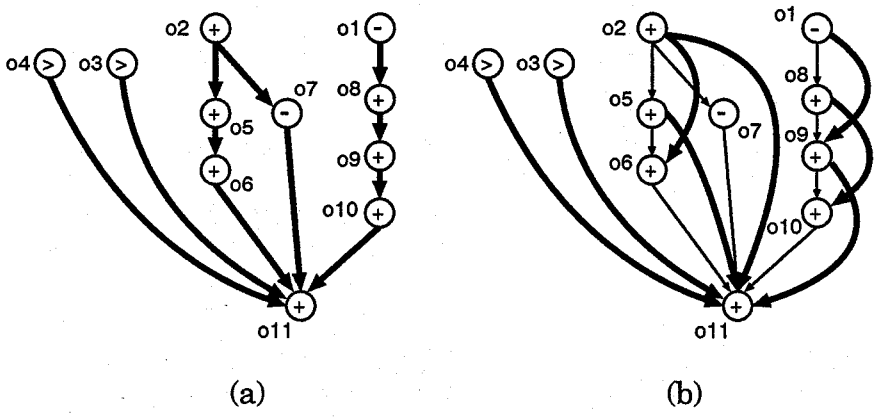


図 5.4: 図 4.5 (b) に対する先行関係グラフ: (a) チェインをしない場合, (b) チェイン数が 2 の場合.

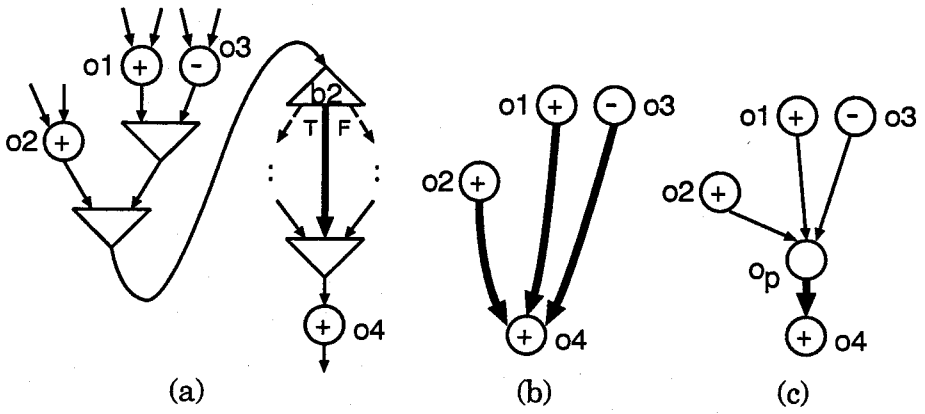


図 5.5: 複数の条件演算がある場合の先行関係グラフの変形: (a) CDFG, (b) 変形前の先行関係グラフ, (c) 変形後の先行関係グラフ.

グラフは同図 (b) のようになる. しかしながら, 後に述べる整数計画問題への定式化の中では, 問題を簡単にするために, 各条件分岐の条件演算はただ一つであることを前提としている. そこで, 条件演算が複数ある場合は, 図 5.5 (c) のように, 遅延時間が 0 である擬似演算節点 o_p を挿入し, この o_p を新たに条件演算とするように先行関係グラフを変形する. この o_p は o_1, o_2, o_3 よりも前のステップに割り当てられてはならないので, $o_1 \rightarrow o_p, o_2 \rightarrow o_p, o_3 \rightarrow o_p$ を設ける. また条件演算が o_1, o_2, o_3 の代りに o_p になったので, $o_1 \Rightarrow o_4, o_2 \Rightarrow o_4, o_3 \Rightarrow o_4$ を開放除去[49]し, 新たに $o_p \Rightarrow o_4$ を設ける. このようにして擬似演算節点を入れても, 演算間の先行関係は保たれ, しかも o_p はどの演算器でも実行されないのでスケジューリングの際に必要となる演算器の個数にも影響を与えない.

5.4 定式化

5.4.1 準備

本項では, 次項の定式化の中で必要な用語と記号の定義を行う.

CDFG 中の演算数を n とし, 演算の集合を $O = \{o_i | 1 \leq i \leq n\}$ とする. 各演算 o_i を実行する演算器の種類を $type(o_i)$ で表し, 演算器のすべての種類の集合を $T = \{t_k | 1 \leq k \leq m\}$ で表す. 種類 t_k の演算器で実行される演算の集合を O_k とすると, $O_k = \{o_i | o_i \in O, type(o_i) = t_k\}$ である. O_k に属する演算を種類 t_k の演算と呼ぶ.

制約として与えられた制御ステップ数を r で表し, 各制御ステップは j $(1 \leq j \leq r)$ で表すこ

とにする. 各演算 o_i を ASAP (As Soon As Possible) 法, ALAP (As Late As Possible) 法[18]で割り当てた制御ステップをそれぞれ E_i, L_i とすると, o_i は E_i と L_i の間に割り付けられなければならない. この範囲を $range(o_i) = \{j | E_i \leq j \leq L_i\}$ で表し, o_i の**実行可能範囲**と呼ぶことにする.

CDFG 中の条件分岐の集合を $B = \{b_1, b_2, \ldots, b_q\}$ とする. 条件分岐 b_l の条件演算を $o_{cn(b_l)}$ で表す. ここでは, 簡単のために各条件分岐には一つの条件演算が対応しているものとする.

演算 o_i や条件分岐 b_l が含まれる条件分岐のネストのレベルとは, タグ $tag(o_i)$, $tag(b_l)$ の要素数であり, それぞれ $level(o_i), level(b_l)$ と書くことにする.

ステップ j で必要となる種類 t_k の演算器の個数を $N_{k,j}$, 全体として必要となる種類 t_k の演算器の個数を表す整数変数を M_k とする. 演算 o_i がステップ j に割り付けられたかどうかを変数 $x_{i,j}$ で表す. $x_{i,j}$ は o_i がステップ j に割り付けられたときには 1, そうでなければ 0, となるブール変数である. なお, 本文では, 論理和, 論理積を$\vee$, $\wedge$で表し, 算術和, 算術積を$+$, $\times$で表すことにする.

5.4.2 ブール式を用いた定式化

本項では動作記述中に条件分岐がある場合のスケジューリング問題をブール式を用いて定式化する.

4.4節で述べたように, 本手法の目的は, 演算器のコストを最小化することなので, c_k を種類 t_k の演算器のコストとすると, 目的関数は,

$$\text{minimize} \quad C = \sum_{k=1}^{m}(c_k \times M_k) \tag{5.1}$$

と定式化できる.

制約条件については, 文献 [40] で用いられたものと同様, 次の三つを考える.

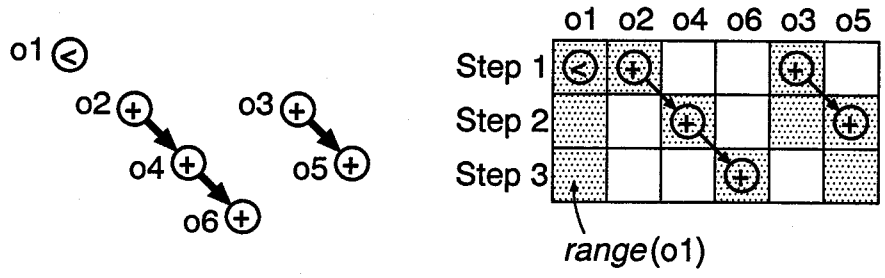
制約条件 1: 制約として与えられた r ステップ内に, すべての演算が割り当てられる.

制約条件 2: すべての演算の先行関係が満足される.

制約条件 3: 各制御ステップで必要な種類 t_k の演算器の個数が, M_k を越えない.

制約条件 1, 2, 3 を表す制約式をそれぞれ $C1$, $C2$, $C3$ とすると, 制約式は以下のように書くことができる:

$$C1 = \bigwedge_{i=1}^{n}\left(\bigvee_{a=E_i}^{L_i}\left(\bigwedge_{j=E_i}^{L_i}\Delta(\delta_{a,j}, x_{i,j})\right)\right) = 1 \tag{5.2}$$



(a) (b)

図 5.6: 図 4.3 の CDFG に対する先行関係グラフと各演算の実行可能範囲：(a) 先行関係グラフ, (b) 各演算の実行可能範囲.

ここで,

$$\Delta(1, x_{i,j}) = x_{i,j}, \Delta(0, x_{i,j}) = \overline{x_{i,j}}, \delta_{a,j} = \begin{cases} 1 & \text{if } a = j \\ 0 & \text{if } a \neq j \end{cases}.$$

$$C2 = (\bigwedge_{o_m \Rightarrow o_n} (\bigwedge_{a=E_n}^{L_m} (\bigwedge_{b=a}^{L_m} (\overline{x_{n,a} \wedge x_{m,b}})))) \wedge (\bigwedge_{o_m \rightarrow o_n} (\bigwedge_{a=E_n}^{L_m} (\bigwedge_{b=a+1}^{L_m} (\overline{x_{n,a} \wedge x_{m,b}})))) = 1. \quad (5.3)$$

$$C3 = \bigwedge_{k=1}^{m} (\bigwedge_{j=1}^{r} (M_k - N_{k,j} \geq 0)) = 1. \quad (5.4)$$

これらの制約式は, 基本的には文献 [40] で定式化されたものと同様であるが, $C3$ に関して, 文献 [40] では条件分岐がない単純な動作の場合についてだけ考えた. ここでは, 条件分岐がある場合に $C3$ をどの様に表現できるかについて焦点をあてる.

条件分岐がない場合 $C3$ の $N_{k,j}$ (ステップ jで必要となる種類 t_k の演算器の個数) は, ステップ j で実行される種類 t_k の演算の個数で表されるので,

$$N_{k,j} = \sum_{i|(o_i \in O_k)} x_{i,j} \quad (5.5)$$

と表せる[40]. 条件分岐がある場合は, 演算間で演算器を共有できる場合があるので, この式で表すことはできない.

条件分岐がある場合に $N_{k,j}$ をどのように表すことができるかを, 例として図 4.5 (a) の動作記述を用いて考える. 同図 (b) の CDFG に対して先行関係グラフは図 5.6 (a) のようになり, 各演算の実行可能範囲は同図 (b) のようになる. ここでも 4.4 節のときと同様, '+' を加算器, '<' を比較器で実行することにし, 加算器, 比較器をそれぞれ t_1, t_2 で表すことにする.

まず, ステップ2で必要となる加算器の個数 $N_{1,2}$ をどのように表すことができるかを考える. 真のパス中の加算でステップ2に割り当てることができるのは o_4 だけであるから, 真のパス中の演算に必要な加算器の個数は $x_{4,2}$ と表すことができる. 一方, 偽のパスについて見てみると, ステップ2には o_3 と o_5 を割り当てることができるので, 偽のパス中の演算に必要な加算器の個数は $x_{3,2}+x_{5,2}$ と表すことができる[1]. 条件演算 o_1 ($range(o_1)=\{1,2,3\}$) がステップ1に割り当てられたとすると, o_4 と o_3, あるいは o_4 と o_5 は互いに排他的となるので, 1個の加算器を共有することができ, ステップ2で必要となる加算器の個数 $N_{1,2}$ は $\max(x_{4,2}, x_{3,2}+x_{5,2})$ と表すことができる. しかし, 条件演算 o_1 がステップ2あるいはステップ3に割り当てられたとすると, 上で述べた演算間の相互排他性はなくなり, したがって, ステップ2で必要となる加算器の個数は $x_{4,2}+(x_{3,2}+x_{5,2})$ と表される. 条件演算 o_1 がステップ s より前に実行されたかどうかは $\bigvee_{1\le j\le s} x_{1,j}$ の値を調べることによってわかる. すなわち, この値が1のときには既に実行が終了しており, 0のときにはまだ終了していないことになる. したがって, $N_{1,2}$ は

$$\begin{aligned} N_{1,2} &= \Big(\bigvee_{1\le j\le 1} x_{1,j}\Big)\times\max(x_{4,2},(x_{3,2}+x_{5,2})) \\ &+ \Big(\overline{\bigvee_{1\le j\le 1} x_{1,j}}\Big)\times(x_{4,2}+(x_{3,2}+x_{5,2})). \end{aligned} \tag{5.6}$$

と表すことができる. 同様に $N_{1,1}$, $N_{1,3}$ は,

$$\begin{aligned} N_{1,1} &= \Big(\bigvee_{1\le j\le 0} x_{1,j}\Big)\times\max(x_{2,1},x_{3,1})+\Big(\overline{\bigvee_{1\le j\le 0} x_{1,j}}\Big)\times(x_{2,1}+x_{3,1}) \\ &= x_{2,1}+x_{3,1} \end{aligned} \tag{5.7}$$

$$N_{1,3} = \Big(\bigvee_{1\le j\le 2} x_{1,j}\Big)\times\max(x_{5,3},x_{6,3})+\Big(\overline{\bigvee_{1\le j\le 2} x_{1,j}}\Big)\times(x_{5,3}+x_{6,3}) \tag{5.8}$$

と表すことができる.

以上の考察に基づき, 一般の場合に $N_{k,j}$ がどのように表されるかについて述べる.

[ケース1] 条件分岐がネストしていない場合

ステップ j で, 条件分岐 b_l 中の演算に必要な種類 t_k の演算器について考える. 条件分岐 b_l の真のパス中の演算に必要な演算器の個数 $NT_{k,j,l}$, 偽のパス中の演算に必要な演算器の個数 $NF_{k,j,l}$ はそれぞれ, タグ中に l, $\bar{l}$ がある演算の個数で表されるので,

$$NT_{k,j,l} = \sum_{i|(o_i\in O_k)\wedge(l\in tag(o_i))} x_{i,j} \tag{5.9}$$

$$NF_{k,j,l} = \sum_{i|(o_i\in O_k)\wedge(\bar{l}\in tag(o_i))} x_{i,j} \tag{5.10}$$

[1]制約条件2により, o_3 と o_5 が同時にステップ2に割り当てられることはないので, $x_{3,2}+x_{5,2}$ の値が2となることはない.

と表せる．ステップ j で，条件分岐 b_l 中の演算に必要な種類 t_k の演算器の個数を $NB_{k,j,l}$ とすると，

$$NB_{k,j,l} = \begin{cases} \max(NT_{k,j,l}, NF_{k,j,l}) & : \text{条件演算 } o_{cn(b_l)} \text{ がステップ} \\ & \quad j \text{ より前で実行されるとき} \\ NT_{k,j,l} + NF_{k,j,l} & : \text{条件演算 } o_{cn(b_l)} \text{ がステップ} \\ & \quad j \text{ 以降に実行されるとき} \end{cases} \tag{5.11}$$

という関係がある．条件演算 $o_{cn(b_l)}$ がステップ j より前で実行されるかどうかは，ステップ $E_{cn(b_l)}$ からステップ $j-1$ まで $x_{cn(b_l),i}$ の論理和を取り，その結果が 1 になればステップ j より前で実行され，0 であればステップ j 以降で実行されるということがわかるので，$NB_{k,j,l}$ は，

$$\begin{aligned} NB_{k,j,l} &= \Big(\bigvee_{E_{cn(b_l)} \le i \le \min(j-1, L_{cn(b_l)})} x_{cn(b_l),i}\Big) \times \max(NT_{k,j,l}, NF_{k,j,l}) \\ &+ \Big(\overline{\bigvee_{E_{cn(b_l)} \le i \le \min(j-1, L_{cn(b_l)})} x_{cn(b_l),i}}\Big) \times (NT_{k,j,l} + NF_{k,j,l}) \end{aligned} \tag{5.12}$$

と書くことができる．ここで，式(5.12)中で $x_{cn(b_l),i}$ の論理和を取る際の上限が $\min(j-1, L_{cn(b_l)})$ となっているのは，$L_{cn(b_l)} < j-1$ の場合があるからである．また，$E_{cn(b_l)} \ge j$ のときは

$$\bigvee_{E_{cn(b_l)} \le i \le \min(j-1, L_{cn(b_l)})} x_{cn(b_l),i} = 0 \tag{5.13}$$

とする．

$N_{k,j}$ は，すべての条件分岐に対する $NB_{k,j,l}$ と，条件分岐外の種類 t_k の演算の個数との和で表現できるので，

$$N_{k,j} = \sum_{l|(b_l \in B)} NB_{k,j,l} + \sum_{i|(o_i \in O_k) \wedge (level(o_i)=0)} x_{i,j} \tag{5.14}$$

と表せる．

［ケース 2］ 条件分岐がネスト している場合

レベルが 0 である条件分岐 b_l に関しては，$NB_{k,j,l}$ は式(5.12)と同じである．条件分岐 b_l の真のパス中に別の条件分岐 b_i $(level(b_i) = level(b_l)+1)$ がある場合を考える．このとき，$NT_{k,j,l}$ は，条件分岐 b_i 中の種類 t_k の演算に必要な演算器数と，$level(o_i) = level(b_l)+1$ であるような種類 t_k の演算の個数との和となるので，

$$NT_{k,j,l} = \sum_{\substack{i|(b_i \in B) \wedge (l \in tag(b_i)) \\ \wedge (level(b_i) = level(b_l)+1)}} NB_{k,j,i} + \sum_{\substack{i|(o_i \in O_k) \wedge (l \in tag(o_i)) \\ \wedge (level(o_i) = level(b_l)+1)}} x_{i,j} \tag{5.15}$$

と書くことができる. $NF_{k,j,l}$ も同様に,

$$NF_{k,j,l} = \sum_{\substack{i|(b_i \in B) \wedge (\bar{l} \in tag(b_i)) \\ \wedge (level(b_i) = level(b_l)+1)}} NB_{k,j,i} + \sum_{\substack{i|(o_i \in O_k) \wedge (\bar{l} \in tag(o_i)) \\ \wedge (level(o_i) = level(b_l)+1)}} x_{i,j} \tag{5.16}$$

と書くことができる. $NB_{k,j,i}$ はレベル 0 の条件分岐のときと同様に式 (5.11) の関係があるから, 式 (5.12) と同じ形で表される. したがって, $NB_{k,j,l}$ は式 (5.12), (5.15), (5.16) を再帰的に適用することによって求めることができる.

$N_{k,j}$ は, レベル 0 のすべての条件分岐に対する $NB_{k,j,l}$ と, 条件分岐外の種類 t_k の演算の個数との和で表現できるので,

$$N_{k,j} = \sum_{l|(b_l \in B) \wedge (level(b_l)=0)} NB_{k,j,l} + \sum_{i|(o_i \in O_k) \wedge (level(o_i)=0)} x_{i,j} \tag{5.17}$$

と表せる.

5.5 二分決定グラフを用いた解法

前節で定式化した整数非線形計画問題を, 文献 [40] と同様, 二分決定グラフ (BDD: binary decisioin diagram) [50, 51] を用いてブール式の充足問題として解く.

本節では, まず BDD について概説し, ついで BDD を用いた解法について述べる.

5.5.1 二分決定グラフ

二分決定グラフ (BDD) は, 有向グラフにより論理関数を表現するデータ構造であり, Akers[50] によって考案され, Bryant[51] により効率的な演算算法が提案された. BDD は,

1. 論理関数の標準形となる.

2. 多くの重要な関数が実用的な記憶量で表現できる.

3. 演算効率が良い.

という優れた特長を持っており, VLSI の CAD の様々な応用に用いられるようになった[52]. 一つの論理関数をいくつもの BDD で表すことができるが, ここでは, 特に断らない限り, 既約な順序付き BDD[52] を考えることにする.

例えば, $f = x_1 \wedge x_2 \vee x_3$ を表す BDD は図 5.7 のようになる. この BDD において, 根から出発して, 節点にラベル付けされた変数の値が 0 であれば 0 とラベル付けされた枝 (以後, **0 枝**と

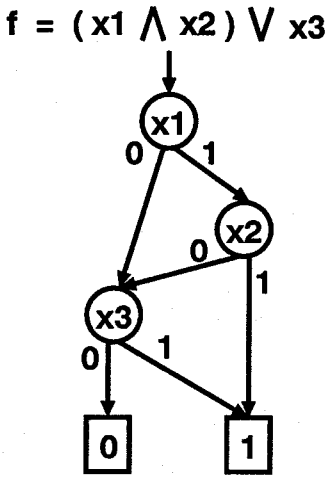


図 5.7: 二分決定グラフ（BDD）の例.

呼ぶ）を, 1 であれば 1 とラベル付けされた枝（以後, **1 枝**と呼ぶ）をたどるという操作を繰り返し, 最終的にたどり着いた節点にラベル付けした論理値が, その変数の割当に対する関数 f の論理値となる. 例えば, $x_1 = 1$, $x_2 = 1$ という変数の割当を考えると, 1 とラベル付けされた節点にたどり着くので, $f = 1$ であることがわかる. また, $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ という変数の割当を考えると, 0 とラベル付けされた節点にたどり着くので, $f = 0$ であることがわかる. 以後, 1 (0) とラベル付けされた終端節点を **1 (0) の定数節点**と呼び, 1 (0) の定数節点へのパスを **1 (0)- パス**と呼ぶことにする.

BDD を用いると, 与えられた論理関数を 1 にする変数の割当を, 変数の個数に比例した計算時間で求めることができる. また, 論理関数を 1 にする変数の割当の組が全部で n 個あったとすると, $n \times$（変数の個数）に比例した時間で求めることができる[51].

5.5.2 解法

5.4.2 項で定式化した $C1$, $C2$, $C3$ を用いて, $F \triangleq C1 \wedge C2 \wedge C3 = 1$ という充足問題を考える. この式を満足する変数値の組は制約条件 1, 2, 3 を満たす実行可能解であり, これらの実行可能解の中で式 (5.1) の目的関数を最小にするものを選択すれば, それが所望の解となる.

ブール関数 F を表す BDD を考える. 5.5.1 項で述べたように, この BDD 上の 1-パスは $F = 1$ を満たす変数の割当を示す. したがって, 前節で定式化した計画問題を, 目的関数を最小にする 1-パスを求める問題として解くことができる.

F を BDD で表現する際に, $C2$ における整数変数 M_k については, 文献 [40] と同様, 0, 1 の値

をとるブール変数 $m_{k,a}$ を導入し,

$$M_k = \sum_{a=1}^{N_k} a \times m_{k,a} \tag{5.18}$$

で表す. ここで, N_k は各ステップにおける種類 t_k の演算の実行可能範囲の重なりの最大値であり, M_k が取り得る最大値を示す. 例えば, 図 5.6 の場合の加算 (t_1) を考えると, ステップ 1 は o_2 と o_3 の実行可能範囲に含まれるので, ステップ 1 における実行可能範囲の重なりは 2 である. 同様に, スッテプ 2, ステップ 3 における加算の実行可能範囲の重なりはそれぞれ 3, 2 となり, したがって, $N_1 = 3$ となる. $m_{k,a}$ は $1 \leq a \leq N_k$ で一つだけ 1 となり, 残りは 0 となるものとする. また, 式中の整数の算術演算は 2 進化符号で扱い, 負の数は 2 の補数で表す.

BDD の変数順序について, 文献 [40] では特に言及していないが, 本手法では, 解を効率良く列挙できるように変数の順序を以下のように定める.

1. $m_{k,a}$ に関する変数を $x_{i,j}$ に関する変数よりも根に近くする.
2. $m_{k,a}$ に関して, 各 k については, a が小さいものをより根に近くする.

この変数順序にしたがって構築された BDD 上で目的関数を最小にする 1-パスを探索する場合を考える. ある種類 t_h について, p 個の演算器を用いた解が存在しないとき, すなわち $m_{h,p} = 1$ となる解が存在しないとき, $m_{h,p}$ を表す節点から 1 枝をたどると, その終点は必ず 0 の定数節点となる (図 5.8 の $m_{h,1}$ や $m_{h,2}$). したがって, $m_{h,a}$ について $a = 1, 2, 3, \ldots$ と順にたどり, 1 枝が 0 の定数節点に接続していないはじめての節点が $m_{h,q}$ であったとすると, 種類 t_h の演算器に関して必要となる最小個数は q 個であることが判明する (図 5.8 の場合では $q = 3$). この例からわかるように, 節点を探索していく際に,

1°: 1 枝をたどる.

2°: 到達した先が 0 の定数節点ならば, 後戻りして **3°**へ. そうでなければ **4°**へ.

3°: 0 枝をたどる.

4°: 到達した先に探索点を移して **1°**へ.

という手順で探索すれば, $m_{k,a}$ に関する変数に対応する節点をたどり終えた時点で, 各演算器について必要となる最小個数が判明する. このときの計算量は $m_{k,a}$ の変数の個数 (BDD の節点数ではない) に比例する. さらに, $x_{i,j}$ の変数についても探索を続ければ, $x_{i,j}$ の変数の個数に比例した計算量で目的関数を最小とする $x_{i,j}$ の値が求まる. さらに, (変数の個数) × (最適解の個数) に比例した計算量ですべての最適解を求めることができる.

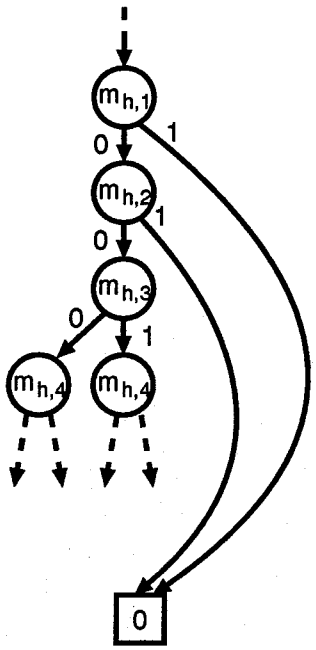


図 5.8: BDD の節点の探索例.

5.6 実験結果

本スケジューリング手法を SPARC station 2（主記憶 32 MB）上に C 言語を用いて実現し, 文献 [25] で用いられた **maha**, **parker**, **wmaha** の三つのベンチマークデータにより本手法の評価を行った. BDD パッケージは文献 [53] の SBDD を用いた.

本手法の結果を表 5.1 に示す. 比較のためにこれまで提案された時間制約型スケジューリング手法の中で最も良い結果を得ることができる *ALPS*[19]の結果も合わせて示す. 図 5.9 に, データ **maha** に対してステップ数を 4, チェイン数を 1 という制約を与えた場合のスケジューリング結果を示す.

ALPS では条件分岐中の演算よりも条件演算を先に実行するという前提で解いているため, 演算のチェインを行わない場合, 7 以下のステップ数でスケジューリングを行うことはできない. 本手法は, このような前提を置いていないため, 演算のチェインを行わない場合でも, 5 ステップで同じ演算器の個数を用いたスケジューリングを得ることができる.

本手法の性能を見るために, 本手法と同様, 条件演算と条件分岐中の演算を並行して実行することを許す *CVLS*[25]のスケジューリング結果を表 5.2 に示す. *CVLS* は資源制約型スケジューリング手法であり, 各実行パスの最適化も行っている. 表中の *total*, *max*, *min*, *ave* はそれぞれ,

```
                                                    ステップ
t1 = in5 - in6;                                     1
t2 = in2 + in3;                                     1
if (in5 != 0) {
    if (t2 != 0) {
        t3 = in1 - 4;                               2
        if (t3 != 0)
            t4 = in2 + 4;                           2
        else
            t4 = in3 - in5;                         2
    } else {
        t3 = in4 - 5;                               1
        t5 = t3 + 5;                                2
        if (t5 != 0)
            t6 = in1 + in2;                         1
        else {
            t7 = in1 - in2;                         1
            t6 = t7 + in1;                          2
        }
        t4 = t6 - in4;                              3
    }
    t6 = t4 + in4;                                  4
} else {
    if (t1 != 0)
        t6 = in2 + 5;                               1
    else
        t6 = 8 - in4;                               1
}
if (t6 != 0)
    out1 = in1 - 5;                                 3
else
    out1 = 8 + in5;                                 3
```

図 5.9: データ **maha** に対してステップ数を 4, チェイン数を 1 という制約を与えた場合のスケジューリング結果.

表 5.1: *ALPS* との比較.

データ	手法	制約		結果	
		ステップ数	チェイン数	加算器数	減算器数
maha	本手法	**3**	**2**	**2**	**2**
	ALPS	3	3	3	3
	本手法	**4**	**2**	**2**	**1**
	ALPS	4	2	2	2
	本手法	**4**	**1**	**2**	**2**
	本手法	**5**	**1**	**1**	**1**
	ALPS	8	1	1	1
wmaha	本手法	**5**	**1**	**3**	**2**
parker	本手法	**3**	**2**	**2**	**2**
	本手法	**4**	**1**	**2**	**2**
	本手法	**5**	**1**	**1**	**1**

表 5.2: *CVLS* の結果

データ	結果（ステップ数）				制約		
	total	*max*	*min*	*ave*	チェイン数	加算器数	減算器数
maha	3	3	1	1.56	3	2	3
	4	4	1	2.38	2	2	1
	4	4	1	2.38	1	2	3
	5	5	2	3.31	1	1	1
wmaha	7	7	2	4.80	1	2	3
parker	4	4	1	2.38	1	2	3
	5	5	2	3.31	1	1	1

全演算を割り当てるために必要となった全ステップ数, 実行パスの最多ステップ数, 最小ステップ数, 平均ステップ数である.

CVLS では, 加算器数 2, 減算器数 3, チェイン数 1 という制約で, スケジューリングした場合に必要なステップ数が 4 となっている. 本手法で, ステップ数を 4, チェイン数を 1 という制約で, スケジューリングした場合は, 加算器数 2, 減算器数 2 という結果を得ている.

各データに対して, 制約を変えて本手法を適用したときの, 制約式を表すのに必要になった変数の個数および BDD の節点の個数, 解の個数, 処理時間を表 5.3 に示す. *CPU1* はすべての解を求めるために必要となった CPU 時間であり, *CPU2* はただ一つの解を求めるようにしたときに要した CPU 時間である. なお, 処理時間にはデータの入力, タグ付け, 先行関係グラフの作成, 制約式の生成, BDD の構築, 解の探索のすべての処理が含まれる. 解の探索のみに要した時間を ( ) 内に示す. データ **wmaha** に対してステップ数 6, チェイン数 1 という制約を与えたと

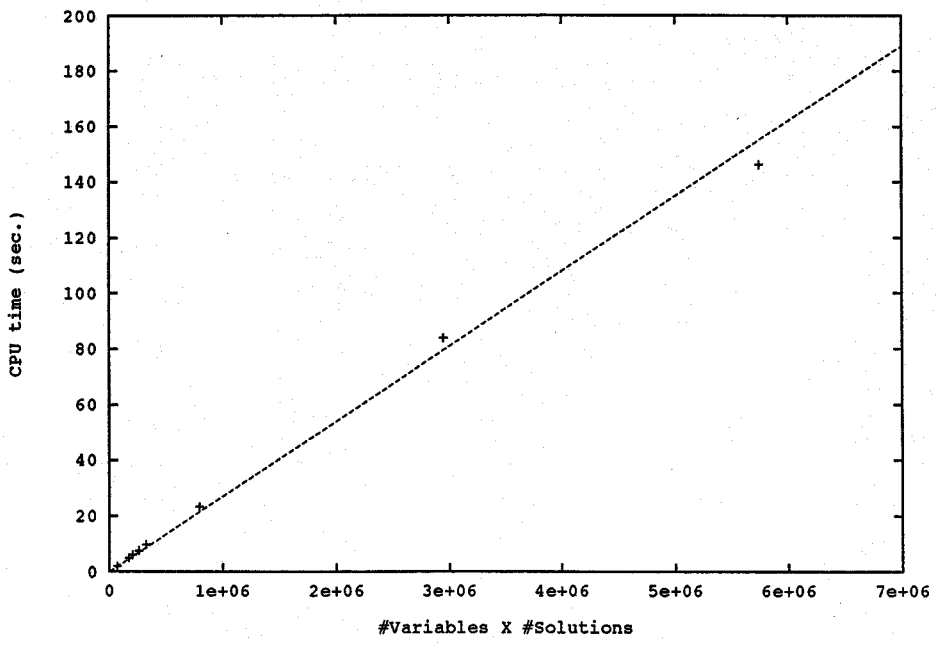


図 5.10: 解の探索時間と（変数の個数）×（解の個数）の関係

きには, BDD の節点数が使用した SBDD パッケージで扱える限界（2^{20}個）を越えてしまい, 解を得ることができなかった.

図 5.10 に, すべての解を求める際の解の探索時間を縦軸に,（変数の個数）×（解の個数）を横軸にしたグラフを示す. このグラフからすべての解を求める際の解の探索時間は（変数の個数）×（解の個数）にほぼ比例していることが確認できる.

表 5.3: 実行結果.

データ	ステップ数	チェイン数	変数の個数	BDD の節点の個数	解の個数	処理時間（秒）	
						CPU1	*CPU2*
maha	3	2	70	3,357	2,904	7.7 (6.1)	1.7
	4	2	92	32,894	32,064	87.3 (84.0)	3.3
	4	1	68	1,032	3,840	9.1 (7.6)	1.5
	5	1	90	11,292	768	4.2 (2.0)	2.2
wmaha	5	1	148	475,283	38,808	161.2 (146.2)	15.2
	6	1	192	—	—	— (—)	—
parker	3	2	70	3,453	4,608	11.4 (9.8)	1.6
	4	1	67	1,191	11,880	24.6 (23.3)	1.5
	5	1	90	13,295	1,920	7.2 (5.0)	2.2

5.7 結言

本章では, 条件分岐を含む動作記述に対する時間制約型スケジューリング手法について考察した. 本手法は, スケジューリング問題を整数非線形計画問題として定式化することにより, 動作記述に任意の構造の条件分岐がある場合でも最小の演算器コストでスケジューリングを行うことができる.

本章では, まず, 定式化の際に必要となる, 演算間の相互排他性や演算間の先行関係という情報を, 与えられたCDFGから抽出する方法について述べ, 次に, これらの情報をもとに整数非線形計画問題への定式化を行った. さらに, この計画問題の制約式をブール式で表現し, 二分決定グラフを用いてすべての解を効率良く列挙する解法を示し, 最後に実験を通じて本手法の有効性を確認した.

本手法は, 最も一般的なスケジューリング問題に対処するものであるが, 問題が大きくなると制約式をBDDで表すことができなくなるという実用上の欠点がある. この問題に対する解決策を次章で述べることにする.

第 6 章

分枝限定法を用いたスケジューリング手法

6.1 緒言

前章で述べたスケジューリング手法は, 目的関数を線形で表すことにより, 効率良くすべての解を列挙することができるが, 変数の個数が増大すると制約式を表す BDD のサイズが急激に大きくなり, 実用上取り扱いが困難となる. そこで, 本章では, より大きなデータが取り扱えるように, 前章で定式化した整数計画問題を変形して, 制約式を表す BDD を小さくすることを考える. 本章で定式化される整数計画問題は, 目的関数が線形で表されないので, 前章と同じ方法で解を求めることができず, 効率の良い分枝限定法を適用することにする. 分枝限定法の処理の効率は,

(i) 限定操作を行うための解のコストの下界を求める関数

(ii) 変数の探索順序

に大きく依存するため, 処理を高速化するためにはこれらを適切に決定しなければならない. (i) については, 解のコストの計算過程を詳しく調べることにより, 適切な関数を見い出し, (ii) については, 探索空間をできるだけ小さくし, できるだけ早い段階で限定操作を行えるような変数順序を考察する.

本章では, まず, 定式化した整数計画問題を変形する手続きについて述べ, 次に, 分枝限定法において探索中に解のコストの下界を求める関数と探索順序について考察する. 最後に, 実験を通じて, 本手法が前章で述べた手法よりも大きいデータに対して有効であることを示し, また, 提案した解のコストの下界を求める関数と探索順序により高速化が達成できることも示す.

6.2 整数計画問題の変形

前章で述べた手法では, 問題の規模が大きくなり変数の個数が多くなると, 制約式を表すBDDが大きくなり, 制約式を表現しきれなくなる. ここでは, 変数の個数を削減してBDDを小さくするために, 種類t_kの演算器の総数を表していた整数変数M_kを使わずに, 制約条件として, 制約条件1と制約条件2だけを用い, 制約条件3を目的関数に組み込むことにより整数計画問題を変形する.

種類t_kに関して必要となる演算器の個数を$N_{k,j}$(ステップjで必要となる種類t_kの演算器の個数)を用いて表すと, $\max_{1\leq j\leq r} N_{k,j}$と書くことができるので, 式(5.1)の$M_k$の代わりに$\max_{1\leq j\leq r} N_{k,j}$を用いて,

$$\text{minimize} \quad C' = \sum_{k=1}^{m}(c_k \times \max_{1\leq j\leq r} N_{k,j}) \tag{6.1}$$

を目的関数として用いる. なお, 式(6.1)の$N_{k,j}$は式(5.17)で計算される.

制約条件としては, 前章で用いた制約条件1, 2を用いる. これら制約条件を表す制約式は式(5.2), (5.4)と同じで,

$$C1 = \bigwedge_{i=1}^{n}(\bigvee_{a=E_i}^{L_i}(\bigwedge_{j=E_i}^{L_i}\Delta(\delta_{a,j}, x_{i,j}))) = 1 \tag{6.2}$$

$$C2 = (\bigwedge_{o_m\Rightarrow o_n}(\bigwedge_{a=E_n}^{L_m}(\bigwedge_{b=a}^{L_m}(\overline{x_{n,a}\wedge x_{m,b}})))) \wedge (\bigwedge_{o_m\rightarrow o_n}(\bigwedge_{a=E_n}^{L_m}(\bigwedge_{b=a+1}^{L_m}(\overline{x_{n,a}\wedge x_{m,b}})))) = 1 \tag{6.3}$$

である.

6.3 分枝限定法を用いた解法

6.3.1 解法

前節で定式化した整数計画問題を解くために, ブール関数$F' \triangleq C1 \wedge C2$を表すBDDを作成し, このBDDに対して, 分枝限定法を適用して解(式(6.1)の目的関数を最小にする1-パス)を求める. この手続きBranch_and_Boundを図6.1に示す.

この手続き中で用いられる変数すべてをベクトルで表すことにする. 例えば, 図5.6(b)のように各演算の実行可能範囲が求まっている場合であれば, $\mathbf{x} = (x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{4,2}, x_{6,3}, x_{3,1}, x_{3,2}, x_{5,2}, x_{5,3})$となる.

```
     procedure Branch_and_Bound (v) :
 1:  begin
 2:      F' を表す BDD を構築する
 3:      C_0 := ∞; C* := ∞;
 4:      x_0 := (0,...,0); x* := (0,...,0);
 5:      DFS (BDD の根);
 6:  end

     procedure DFS (v) :
 7:  begin
 8:      if 節点 v が 1 の定数節点である then
 9:          C_0 := C*;
10:          x_0 := x*;
11:          return;
12:      else if 節点 v が 0 の定数節点である then
13:          return;
14:      else
15:          0 枝 (v,u) を選ぶ;
16:          DFS (u);
17:          1 枝 (v,w) を選ぶ;
18:          x̂ := x*;
19:          x̂ の中で, 節点 v にラベル付けされた変数 x_{i,j} に対応する要素の値
             を 1 とする;
20:          Ĉ := Lower_Bound (x̂);
21:          if C_0 < Ĉ return;
22:          C* := Ĉ;
23:          x* := x̂;
24:          DFS (w);
25:          x* の中で, 変数 x_{i,j} に対応する要素の値を 0 にする;
26:          return;
27:      endif
28:  end
```

図 6.1: 手続き Branch_and_Bound.

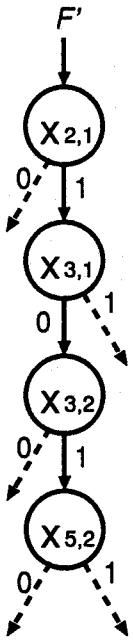


図 6.2: 図 5.6 の例に対する BDD の探索例.

根を始点とする探索が 1 の定数節点に達したとき, 求まった 1-パス上の枝の値をその枝の始点の変数に割り当てて得られる系列を**完成解**と呼ぶ. 図 5.6 の例では $\mathbf{x} = (1,0,0,1,1,1,1,0,1,0)$ は一つの完成解である. この完成解のコストとは, 式 (6.1) の変数に得られた値を代入することによって求まる値である. 処理中のある時点で求まった完成解の中で最小のコストのものを**暫定解**と呼ぶ. 図 6.1 では, 暫定解を $\mathbf{x_0}$ というベクトルで表し, その暫定解のコストを C_0 で表している.

定数節点以外のある節点 v に探索が到達したときに, 根からその節点までのパス上の枝の値をその枝の始点の変数に割り当て, それ以外の変数には 0 を割り当てて得られる系列を**部分解**と呼び, 部分解においてまだ探索されていない節点の変数のことを**自由変数**と呼ぶ. 図 5.6 の例で, 図 6.2 のように BDD が探索されて, $x_{5,2}$ の節点に到達したときを考える. このとき, $x_{2,1} = x_{3,2} = 1$, $x_{3,1} = 0$ と定まっているので, 部分解は $(\underline{0},\underline{0},\underline{0},1,\underline{0},\underline{0},0,1,\underline{0},\underline{0})$ となり, 自由変数は $x_{1,1}, x_{1,2}, x_{1,3}, x_{4,2}, x_{6,3}, x_{5,2}, x_{5,3}$ である (自由変数に対応する 0 には下線を付けて示している). Lower_Bound() は, 得られた部分解から, その部分解の自由変数の値を決めることによって得られる完成解のコストの下界を計算する関数である. 図 6.1では, 部分解を $\mathbf{x}^*$, $\hat{\mathbf{x}}$ というベクトルで表し. これらの部分解から Lower_Bound() を用いて計算した解のコストの下界を, それぞれ C^*, $\hat{C}$ で表している.

手続き Branch_and_Bound では次のようにして解を探索する: 初期状態ではすべての変数の値を 0 にして (どの演算もどのステップに割り当てられていないことを意味する) (4 行目), BDD の根から縦形探索 (depth-first search) [49] を適用する (5 行目). いま DFS(v) が呼び出されて節点 v に探索が到達したとする (以下, 探索が到達した節点のことを**探索点**と呼ぶ). このときの部分解は $\mathbf{x}^*$, この部分解を用いて計算した解のコストの下界は C^* である. 節点 v から 0 枝を通って節点 u に到達した場合には, 単に探索点を節点 u に移す (15–16 行目). 節点 v から 1 枝を通って節点 w に到達した場合には, $\mathbf{x}^*$ 中の節点 v が表す変数を 1 にした部分解 $\hat{\mathbf{x}}$ を用いて解のコストの下界 $\hat{C}$ を計算し (17–20 行目), 暫定解のコスト C_0 と比較して, それ以上探索を進めても暫定解より良い完成解が得られない ($C_0 < \hat{C}$) とわかれば, 後退して他の解を探索する (21 行目). もし $C_0 > \hat{C}$ であれば, $\hat{C}$, $\hat{\mathbf{x}}$ をそれぞれ C^*, $\mathbf{x}^*$ にコピーして探索点を w に移す (22–24 行目). 探索が 1 の定数節点に到達すると, 暫定解 $\mathbf{x_0}$ を更新する (8–10 行目). この探索がすべて終了した時点での暫定解 $\mathbf{x_0}$ が最適解となる.

ここで問題となるのは, 部分解からどのように解のコストの下界を計算するかということである. 下界として, 単純に式 (6.1) に部分解の変数の割当を代入して求めようとすると, 探索がより深く進んだときの方が, 求まった解のコストの下界の値が小さくなってしまう場合がある. 例えば, 図 6.3 (a) の CDFG に対するスケジューリングを行っているときに, 同図 (b) のように BDD が探索された場合を考える. ここで, '+' を加算器 (t_1), '<' を比較器 (t_2) で実行するものとし, 式 (6.1) の c_k については $c_1 = c_2 = 1$ とする. 同図 (b) で, 探索が $x_{1,2}$ の節点に到達したとき, $x_{6,1} = x_{4,3} = x_{5,3} = 1$ と決定されており, この段階で同図 (c) のようなスケジューリングが得られている. このときの解のコストの下界は, 式 (6.1) を用いた場合, 加算器が 2 個, 比較器が 1 個必要であると計算され, $C^* = 3$ となる. さらに, 探索が進み, $x_{2,1}$ の節点まで到達したとすると, 条件分岐 b_1 の条件演算 o_1 がステップ 2 に割り当てられるので, o_4 と o_5 は 1 個の加算器を共有でき, したがって $C^* = 2$ となり, 探索が進んだ方が解のコストの下界が小さくなってしまう. このような状況が起こると, 最適解にたどり着く経路を探索中であるにも関わらず, 中途で解のコストの下界を大きく見積もり過ぎて, その探索を打ち切ってしまうということになり, 正しく解を求めることができない.

この計算過程を詳しく見てみることにする. 式 (6.1) の $N_{k,j}$ は式 (5.17) を用いて計算され, その中の $NB_{k,j,l}$ は式 (5.12) を用いて計算されるが, $NB_{k,j,l}$ の値は条件演算の実行ステップにしたがって, $\max(NT_{k,j,l}, NF_{k,j,l})$ か $NT_{k,j,l} + NF_{k,j,l}$ が選ばれる. 図 6.3 (c) の段階では, 条件分岐 b_1 の条件演算 o_1 に対応する変数 $x_{1,1}$, $x_{1,2}$, $x_{1,3}$ の値がまだ決まっていない, すなわち 0 なので, ステップ 3 で必要な加算器の個数 $N_{1,3}$ は $N_{1,3} = NB_{1,3,1} = NT_{1,3,1} + NF_{1,3,1} =$

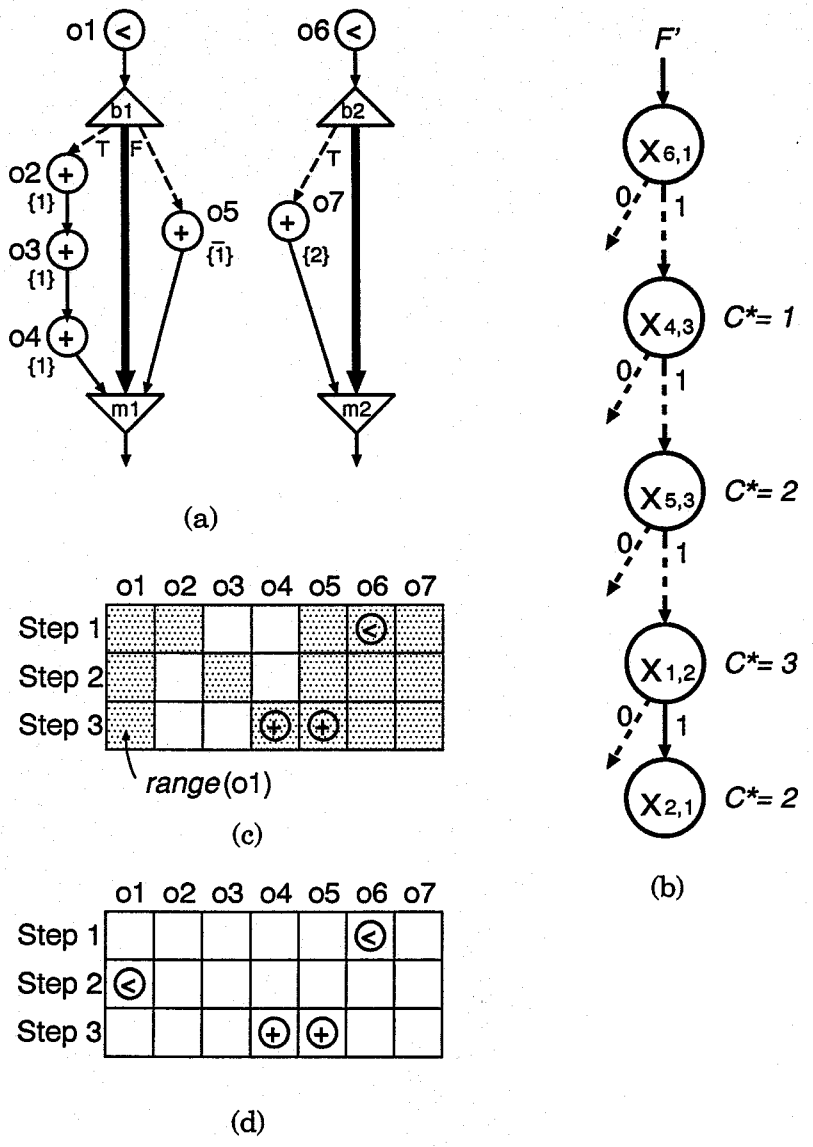


図 6.3: 式(6.1)で解のコストの下界が正しく計算できない例: (a) CDFG, (b) BDD の節点の探索例, (c) 探索が $x_{1,2}$ の節点に到達したときのスケジューリング, (d) 探索が $x_{2,1}$ の節点に到達したときのスケジューリング.

$x_{4,3}+x_{5,3}=1+1=2$ となる. この後, 条件演算 o_1 がステップ 2 に割り当てられることにより, $N_{1,3}=NB_{1,3,1}=\max(NT_{1,3,1},NF_{1,3,1})=\max(x_{4,3},x_{5,3})=\max(1,1)=1$ となり, o_1 が割り当てられる前よりも値が小さくなるのである.

そこで本手法では, 条件演算の実行ステップが決まっていない場合は, $NB_{k,j,l}$ の値として $\max(NT_{k,j,l},NF_{k,j,l})$ を選択することにし, 解の下界を大きく見積もり過ぎることがないようにする. すなわち, 条件演算 o_i に対応する変数の値が既に決まっていれば 1, そうでなければ 0, となるフラグ $assigned(o_i)$ を導入し, 式 (5.12) の代わりに,

$$\begin{aligned} NB_{k,j,l} &= (\overline{assigned(o_{cn(b_l)})} \vee \bigvee_{E_{cn(b_l)}\le i\le \min(j-1,L_{cn(b_l)})} x_{cn(b_l),i}) \\ &\quad \times \max(NT_{k,j,l},NF_{k,j,l}) \\ &+ (assigned(o_{cn(b_l)}) \wedge \overline{\bigvee_{E_{cn(b_l)}\le i\le \min(j-1,L_{cn(b_l)})} x_{cn(b_l),i}}) \\ &\quad \times (NT_{k,j,l}+NF_{k,j,l}) \end{aligned} \tag{6.4}$$

を用いて解のコストの下界を計算する.

6.3.2 二分決定グラフの変数の順序

分枝限定法の処理の効率は, 変数の探索順序 (ここでは BDD の変数順序) に大きく依存する. 本手法では, 効率良く解を探索するために, BDD の変数の順序を以下の基準に基づいて決定する.

[基準 I] 実行可能範囲の小さい演算に対応する変数を根に近くする.

[基準 II] 条件演算に対応する変数を根に近くする.

一般に, 分枝限定法においては選択の幅の小さいものから順に値を決定することにより, 探索空間を小さくできる. そこで, 基準 I を適用する. 図 6.4 は基準 I の効果を示す例である. 先行関係のない二つの演算 o_1 $(range(o_1)=\{1,2\})$ と o_2 $(range(o_2)=\{1,2,3\})$ があるとき, 制約式 $F'=C1\wedge C2$ を表す BDD はどちらの演算に対する変数を根に近く配置するかにより図 6.4 (a) あるいは図 6.4 (b) のようになる (0 の定数節点およびそれにつながるすべての枝は描いていない). この BDD 上で解の探索を行っている際に, 1 枝 (太い枝で示している) を通る度に, 解のコストの下界を計算しなければならない. 図 6.4 (a) の場合, 最悪の場合 9 回計算をしなければならないが, 図 6.4 (b) の場合は, 最多の場合でも 8 回で済む.

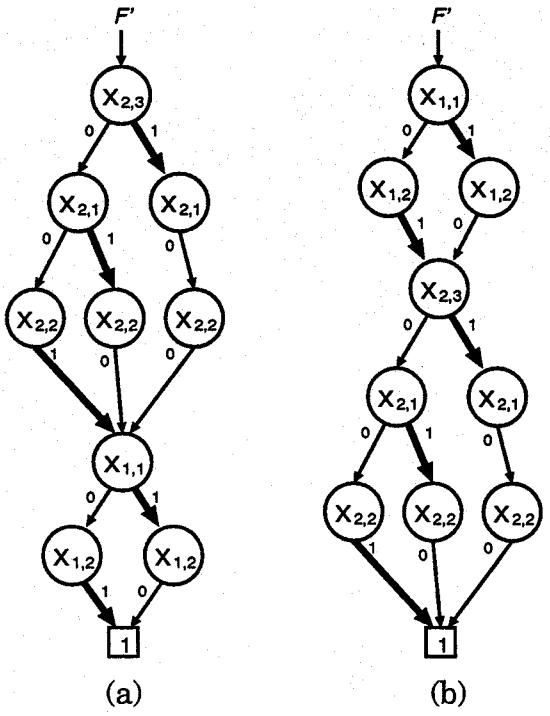


図 6.4: 基準 I の効果を示す例:(a) 基準 I を適用しない場合の BDD,
(b) 規則 I を適用した場合の BDD.

基準 II は早く厳密な解のコストの下界を計算できるようにすることをねらったものである. 図 6.5 の例を用いて基準 II の効果について述べる. 図 6.5 (a) の CDFG に対して BDD の変数順序を図 6.5 (b) のようにした場合を考える. なお, '+' を加算器 (t_1), '<' を比較器 (t_2) で実行することにし, 式 (6.1) の c_k については $c_1 = c_2 = 1$ とする. また, 現在の暫定解のコストは $C_0 = 3$ とする. いま, 探索が節点 $x_{5,2}$ に到達したとすると, $x_{3,2} = x_{6,1} = 1$ と決まっており (演算 o_3 がステップ 2 に, 演算 o_6 がステップ 1 に割り当てられており (図 6.5 (d)), 加算器と比較器が 1 個ずつ必要になるので $C^* = 2$ である. 次に, $x_{5,2}$ を 1 として (演算 o_5 をステップ 2 に割り当てたとして) $\hat{C}$ を計算すると (図 6.1 の 20 行目), 条件分岐 b_1 の条件演算 o_1 の実行ステップがまだ決定されていないため, ステップ 2 で必要となる加算器の個数は, 式 (5.17), (6.4) から $N_{1,2} = NB_{1,2,1} = \max(x_{3,2}, x_{5,2}) = \max(1,1) = 1$ となり, $\hat{C} = N_{1,2} + N_{2,1} = 1 + 1 = 2$ となる. いま $C_0 = 3$ であるから $C_0 < \hat{C}$ という条件が満たされず (図 6.1 の 21 行目), $x_{5,3}$ 以降の探索を続けなければならない. しかし, もし基準 II に従って変数の順序が決定していれば (図 6.5 (c)), $x_{5,2}$ を 1 として $\hat{C}$ を計算すると, 条件分岐 b_1 の条件演算 o_1 の実行ステップが 2 と決まっているので (図 6.5 (e)), ステップ 2 で必要となる加算器の個数は, 式 (5.17), (6.4) から

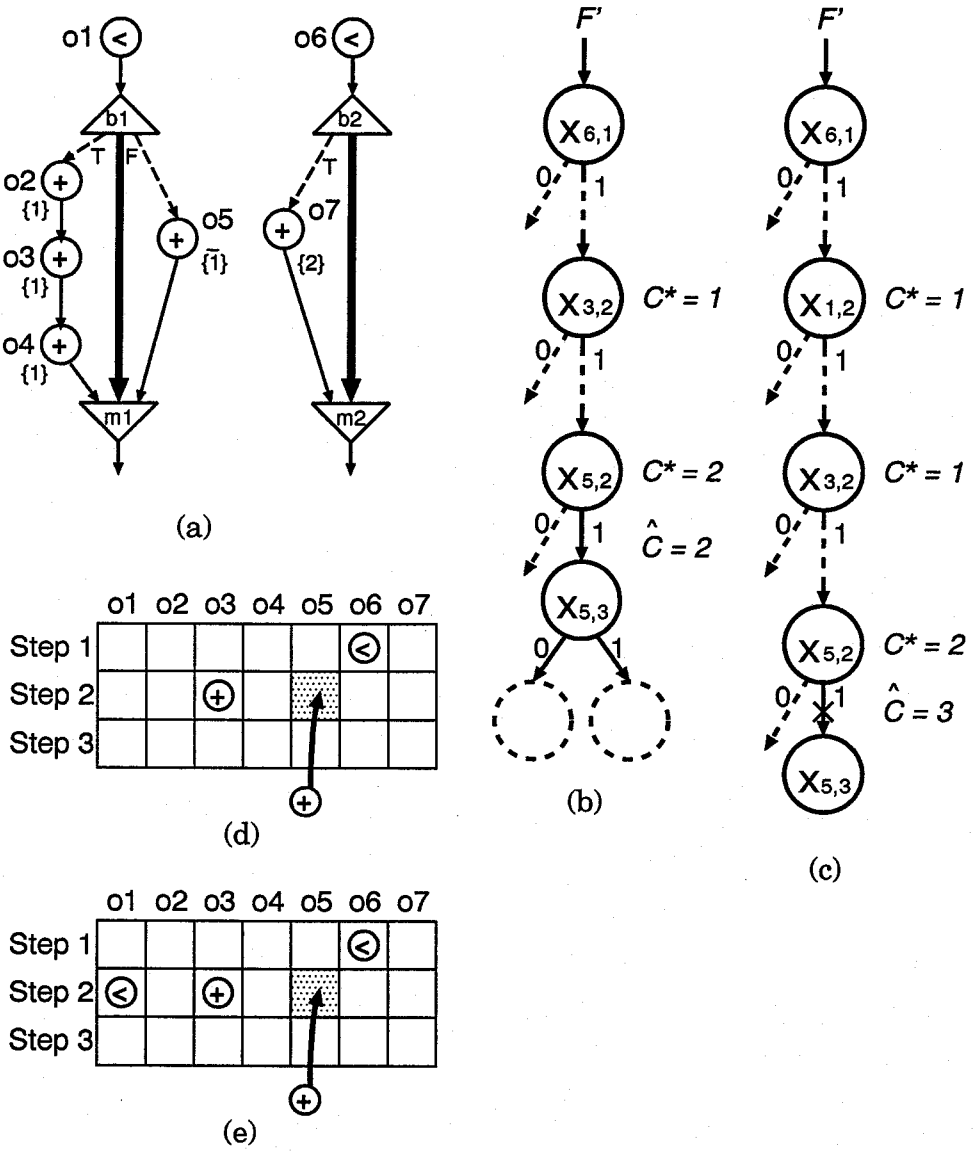


図 6.5: 基準 II の効果を示す例 : (a) CDFG, (b) 基準 II を適用しない場合の BDD, (c) 基準 II を適用した場合の BDD, (d) (b) に対応するスケジューリングの途中経過, (e) (c) に対応するスケジューリングの途中経過.

表 6.1: 本手法で得られた解の結果.

データ	ステップ数	チェイン数	加算器数	減算器数
maha	3	2	2	2
	4	2	2	1
	4	1	2	2
	5	1	1	1
wmaha	5	1	3	2
	6	1	2	2
parker	3	2	2	2
	4	1	2	2
	5	1	1	1

$N_{1,2} = NB_{1,2,1} = x_{3,2} + x_{5,2} = 1 + 1 = 2$ となり, $\hat{C} = N_{1,2} + N_{2,1} = 2 + 1 = 3$ となる. その結果 $C_0 < \hat{C}$ という条件が満たされ, 探索はこの節点より先に進む必要がないことを判断できる.

なお, 基準の優先順位は基準 I の方を高くするものとする.

6.4 実験結果

本スケジューリング手法を SPARC station 2 (主記憶 32MB) 上に C 言語を用いて実現し, 前章で用いた **maha**, **parker**, **wmaha** を用いて評価を行った. ここでも, BDD パッケージとして, 文献 [53] の SBDD を用いた.

本手法は, 前章で述べた手法における制約式 $C3$ を目的関数に取り込んでいるため, スケジューリングの結果は同じである. さらに, 制約式を表すための BDD を小さくできるため, より大きい問題を解くことができる. 例えば, **wmaha** でステップ数を 6 としたときに, 前章で述べた手法では解けなかったが, 本手法では解くことができた.

各データに対して, 制約を変えて本手法を適用したときの, 制約式 F' を表すのに必要になった変数の個数および BDD の節点の個数, 処理時間を表 6.2 に示す. 比較のため, 前章で述べた手法 (ここでは, **解法 1** と書く) で一つの解を求めたときの処理時間も示す.

次に, 本手法において BDD を生成する際の基準 I, II の有効性を評価するための実験を行った. 表 6.3, 6.4 に, 適用する基準を変えたときの生成された BDD の節点の個数, および処理時間を示す. *No* は基準を適用しない場合, *I* は基準 I のみを適用した場合, *II* は基準 II のみを適用した場合, *II+I* は基準 II の優先順位を高くして基準 I と II を両方適用した場合, *I+II* は基準 I の優先順位を高くして基準 I と II を両方適用した場合を示す. なお, 表 6.4 の $> 20,000$ は, 解の探索に 20,000 秒以上必要であることを意味する.

表 6.2: 実行結果.

データ	ステップ数	チェイン数	変数の個数	BDD の個数	処理時間（秒）	
					本手法	解法 1
maha	3	2	54	116	1.7 (0.5)	1.7
	4	2	76	414	2.3 (1.0)	3.3
	5	2	98	1,130	2.1 (0.7)	14.7
	6	2	120	2,562	2.5 (1.2)	—
	4	1	54	55	1.4 (0.1)	1.5
	5	1	76	163	1.5 (0.3)	2.2
	6	1	98	430	1.7 (0.4)	7.9
	7	1	120	991	1.8 (0.5)	54.5
wmaha	5	1	122	306	5.8 (4.4)	15.2
	6	1	166	2,785	44.5 (42.8)	—
parker	3	2	54	116	1.5 (0.2)	1.6
	4	1	54	55	1.4 (0.1)	1.5
	5	1	76	163	2.7 (1.5)	2.2

表 6.3: BDD の変数順序に関する基準を変えた場合の BDD の節点数の変化.

データ	ステップ数	チェイン数	節点数				
			No	*I*	*II*	*II + I*	*I + II*
maha	3	2	124	108	166	118	116
	4	2	368	353	592	400	414
	5	2	819	907	1,522	1,030	1,130
	6	2	1,546	1,976	3,239	2,219	2,562
	4	1	55	55	55	55	55
	5	1	198	171	231	173	163
	6	1	499	448	635	452	430
	7	1	1,028	1,016	1,401	997	991
wmaha	5	1	295	308	332	295	306
	6	1	3,415	2,838	4,241	3,017	2,785
parker	3	2	124	108	166	118	116
	4	1	55	55	55	55	55
	5	1	198	171	231	173	163

表 6.4: BDD の変数順序に関する基準を変えた場合の処理時間の変化.

データ	ステップ数	チェイン数	処理時間 (秒)				
			No	*I*	*II*	*II + I*	*I + II*
maha	3	2	22.8	1.8	5.9	1.9	1.7
	4	2	206.2	2.1	47.5	3.1	2.3
	5	2	713.9	3.0	326.5	2.6	2.1
	6	2	4,601.7	5.3	2,251.2	3.8	2.5
	4	1	15.3	1.5	5.8	1.5	1.4
	5	1	117.3	1.9	8.2	1.6	1.5
	6	1	70.9	2.3	16.8	1.8	1.7
	7	1	132.8	2.6	32.0	2.2	1.8
wmaha	5	1	$> 20,000$	21.9	$> 20,000$	192.1	5.8
	6	1	$> 20,000$	138.1	$> 20,000$	2769.1	44.5
parker	3	2	23.1	1.3	5.5	1.6	1.5
	4	1	15.7	1.4	5.8	1.6	1.4
	5	1	126.9	2.5	9.9	1.9	2.7

表 6.3 から生成された BDD の節点の個数は基準を変えてもほとんど変わらないことがわかる. 一方, 処理速度について見てみると, 基準 I を単独で用いてもかなりの処理速度の向上が得られているが, 基準 II と組み合わせることにより (*I+II* の場合), さらに処理速度が向上していることがわかる. 特に **wmaha** の例では, 基準 I だけの場合よりも 3 倍以上の高速化が得られている.

6.5 結言

本章では, より大きなデータが取り扱えるように, 前章で定式化した整数計画問題を変形し, BDD の規模をより小さくし, 分枝限定法を適用する手法について考察した.

まず, 定式化した整数計画問題を変形する手続きについて述べ, 次に, 分枝限定法における探索順序と探索中に解のコストの下界を求める関数を導入し, 探索操作の高速化を図った. 最後に, 実験を通じて本手法が大きいデータに対して有効であることを確認した.

本章で考察した手法においても, BDD を用いて制約式を表現しているために, さらに問題が大規模になると, 取扱いが実用上困難となるという場合が生じる. さらに, 解法に分枝限定法を用いているために, 処理時間が著しく増大することがある. したがって, より大規模な問題に対しては, 厳密解を得られなくなる可能性があるが, 解法にヒューリスティック手法を組み入れた効率の良い手法を考案する必要があり, これが今後の課題である.

第 7 章

結論

本研究の目的は, VLSI の設計において, 設計の効率化, 高品質化を図るための設計自動化技法について考察することであった. 本論文では, 特に, レイアウト設計における CMOS 論理セルの自動生成手法および機能・論理設計における高位合成のスケジューリング手法について考察した.

本研究で得られた主要な結果を以下に要約する.

1. CMOS 論理セルの自動生成システムを, if–then 型の規則で処理を記述する知識ベースシステムとして構築した. これにより, 設計規則や設計制約に変更がある場合に, 規則を追加, 修正することにより柔軟に対応できることを示した.

2. 高位合成のスケジューリング問題に対して, 動作記述中に条件分岐がある場合に適用可能な, 最適スケジューリング手法を提案した. これは, スケジューリング問題を整数非線形計画問題として定式化し, 二分決定グラフというデータ構造を用いて, すべての解を効率良く列挙するものである. 従来手法との比較実験により, 本手法の有効性を示した.

3. 条件分岐がある場合のスケジューリング問題に対して, より大きな問題を取り扱うために, 分枝限定法を適用する手法を提案した. この中で, 分枝限定法における探索順序と探索中に解のコストの下界を求める関数についても考察を行い, 探索操作の高速化を実現している. さらに, 実験を通じて, 大規模な問題に対して本手法の高速性を明らかにした.

今後に残された課題としては, それぞれ以下のものが挙げられる.

(I) CMOS 論理セルの自動生成

現在のスタンダードセル方式などの設計方式では, 与えられた論理回路をライブラリ中のセルを用いて再構築している. 本論文で述べた手法と論理合成システムを統合化し, 合成された論理回路に見合ったセルを自動生成することができれば, より柔軟な VLSI 設計が可能になると考えられる. また, 電気的特性をも考慮したセルの自動生成に関する研究も今後の課題となる.

(II) 高位合成のスケジューリング

本論文では, 第 6 章で大きい問題を取り扱うための手法について考察したが, この手法においても計画問題の制約式を二分決定グラフで表現して解いているために, 問題の規模がさらに大きくなると実用上取り扱うことができなくなる. さらに, 分枝限定法を用いているために, 本質的に処理時間の増大はまぬがれない. したがって, より大規模な問題に対しては, ヒューリスティック手法を組み入れた効率の良い手法が必要になる. さらに, 実用的なシステムを構築するためには, 条件分岐だけではなく, 処理に繰り返しがある場合の取り扱いや, スケジューリングに適した記述の最適化などについても研究を進める必要があり, これらが今後の課題である.

謝辞

本研究の全過程を通じて, 終始御懇切な御指導と御教示を賜わった大阪大学工学部情報システム工学教室白川功教授に心から感謝申し上げます. また, 同教室石浦菜岐佐助教授には適切な御指導, 御討論をいただき, 心から感謝申し上げます.

大学院博士前期, 後期両課程に在学中に御指導, 御教示を賜わった大阪大学工学部電子工学教室吉野勝美教授, 濱口智尋教授, 西原浩教授, 尾浦憲治郎教授, 児玉慎三教授, 春名正光教授, 情報システム工学教室寺田浩詔教授, 藤岡弘教授, 西尾章治郎教授, 鷹田憲久教授, 鈴木胖教授, 大型計算機センター熊谷貞俊教授, 産業科学研究所溝口理一郎教授, 角所収名誉教授, 塙輝雄名誉教授, 裏克己名誉教授に厚く御礼申し上げます.

大学院博士後期課程において研究を行なうにあたり, シャープ株式会社浅田篤副社長, 生産技術開発推進本部安永龍洋本部長, 元同本部生産技術開発センター所長道畑一三氏, 同本部精密技術開発センター山岡秀嘉所長, 同センター藤原康弘副所長には機会を与えて頂き, また, 技術本部河田亨副本部長, 同本部情報技術研究所第 1 研究部千葉徹部長, 生産技術開発推進本部精密技術開発センター第 4 研究部佐原謙一部長, 同部神戸尚志主任研究員には御理解ある御配慮, 御指導, 御鞭撻を頂きました. ここに深く感謝致します.

第 2 章, 第 3 章について適切な御指導, 御教示を賜わった中央大学築山修治教授に深く感謝致します. また, 有益な御討論を頂いたリコー株式会社の若林謙一氏に深く感謝致します.

第 4 章, 第 5 章, 第 6 章における研究を行なうにあたり, 貴重な実験データと実験結果を提供頂いた日本電気株式会社の若林一敏氏に深く感謝致します. また, SBDD に基づく論理関数処理プログラムを提供して頂いた日本電信電話株式会社の湊真一氏に深く感謝致します. また, 有益な御討論を頂いた大阪大学大学院学生山崎年樹氏, 中村哲氏に心から感謝致します.

大阪大学工学部情報システム工学教室白川研究室の山井成良助手, 重弘裕二助手, 尾上孝雄助手, 鹿島有紀子氏, 大学院学生豊永昌彦氏, 鈴木等氏, 長尾明氏, 岡田圭介氏, 宇野裕史氏, 安齋勝矢氏, 高津正道氏, 正城敏博氏はじめ, 白川研究室の諸氏, ならびに大阪府立大学大学院学生山内仁氏には種々の面で御協力頂きました. 厚く御礼申し上げます.

シャープ株式会社生産技術開発推進本部精密技術開発センター第4研究部の藤本徹哉係長, 竹田信弘主任, 冨田常雄主任, 野田浩明主任, 吉岡智良副主任, 山内貴行副主任, 山口雅之副主任, 横山昌生副主任, 高橋瑞樹副主任, 同センターシミュレーション技術開発プロジェクトの谷貞宏係長, 中政道副主任はじめ, 第4研究部およびシミュレーション技術開発プロジェクトの諸氏には, 多くの御助言, 御協力, 御援助を頂きました. 厚く御礼申し上げます.

参考文献

[1] 電子回路の CAD, 電子通信学会 (1973 年).

[2] 小林勉, 須藤常太, 細田泰弘: "LSI-CAD (I)," 電子情報通信学会誌, vol. 70, no. 12, pp. 1291–1297 (1987 年 12 月).

[3] 上田和宏, 須藤常太: "LSI-CAD (II) —LSI のレイアウト CAD—," 電子情報通信学会誌, vol. 71, no. 1, pp. 80–87 (1988 年 1 月).

[4] 渡辺孝博: "LSI レイアウト自動設計の現状と可能性," 電子情報通信学会誌, vol. 76, no. 7, pp. 774–782 (1993 年 7 月).

[5] 白川功: "VLSI の Physical Design の現状と今後," 電子情報通信学会技術研究報告, VLD93-111 (1994 年 3 月).

[6] 三橋隆, 後藤宣之: "レイアウト CAD の動向: 性能指向レイアウトを中心として," 電子情報通信学会回路とシステム軽井沢ワークショップ, pp. 199–204 (1994 年 4 月).

[7] 松永裕介: "論理合成の動向," 電子情報通信学会回路とシステム軽井沢ワークショップ, pp. 193–198 (1994 年 4 月).

[8] 藤本徹哉, 野田浩明, 竹田信弘, 山口雅之, 神戸尚志: "信号処理カスタム LSI のトップダウン設計について," 電子情報通信学会技術研究報告, VLD92-92 (1993 年 3 月).

[9] 真鍋義文, 萩原兼一, 都倉信樹: "単調減少論理関数を実現する CMOS 回路に対する配置問題," 電子情報通信学会技術研究報告, CAS84-103 (1984 年 10 月).

[10] T. Uehara and W. M. vanCleemput: "Optimal layout of CMOS functional arrays," *IEEE Trans. Comput.*, vol. C-30, no. 5, pp. 305-312 (May 1981).

[11] R. Nair, A. Bruss and J. Reif: "Linear time algorithms for optimal CMOS layout," in *VLSI : Algorithms and Architectures*, pp. 327-338, North-Holland (1985).

[12] Y.J. Kwon and C.M. Kyung: "A heuristic algorithm for minimal area CMOS cell layout," in *Proc. Joint Technical Conference on Circuits and Systems*, CAS87-68 (Jul. 1987).

[13] R. L. Maziasz and J. P. Hays: "Layout optimization of CMOS functional cells," in *Proc. 24th ACM/IEEE Design Automation Conf.*, pp. 544-551 (Jun. 1987).

[14] D. D. Gajski, N. D. Dutt, A. C.-H. Wu and S. Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[15] I. Radivojević and F. Brewer, "Symbolic techniques for optimal scheduling," in *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 145–154 (Oct. 1993).

[16] A. C. Parker, J. Pizarro and M. Mlinar, "MAHA: A program for datapath synthesis," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, pp. 461–466 (Jun. 1986).

[17] C.-J. Tseng, R.-S. Wei, S. G. Rothweiler, M. M. Tong and A. K. Bose, "Bridge: A versatile behavioral synthesis system," in *Proc. 25th ACM/IEEE Design Automation Conf.*, pp. 415–420 (Jun. 1988).

[18] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Comput.-Aided Design Integrated Circuits and Syst.*, vol. 8, no. 6, pp. 661–679 (Jun. 1989).

[19] J.-H. Lee, Y.-C. Hsu and Y.-L. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," in *Proc. IEEE Int. Conf. on Comput.-Aided Design '89*, pp. 20–23 (Nov. 1989).

[20] I.-C. Park and C.-M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," in *Proc. 28th ACM/IEEE Design Automation Conf.*, pp 680–685 (Jun. 1991).

[21] R. Potasman, J. Lis, A. Nicolau and D. Gajski, "Percolation based synthesis," in *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 444–449 (Jun. 1990).

[22] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Comput.-Aided Design Integrated Circuits and Syst.*, vol. 10, no. 1, pp. 85–93 (Jan. 1991).

[23] T. Kim, J. W.S. Liu and C. L. Liu, "A scheduling algorithm for conditional resource sharing," in *Proc. IEEE Int. Conf. on Comput.-Aided Design '91*, pp. 84–87 (Nov. 1991).

[24] K. Wakabayashi, and T. Yoshimura, "A resource sharing and control synthesis method for conditional branches," in *Proc. IEEE Int. Conf. on Comput.-Aided Design '89*, pp. 62–65 (Nov. 1989).

[25] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proc. 29th ACM/IEEE Design Automation Conf.*, pp. 112–115 (Jun. 1992).

[26] S.H. Huang, Y.L. Jeang, C.T. Hwang, Y.C. Hsu, and J.F. Wang, "A tree-based scheduling algorithm for control-dominated circuits," in *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 578–582 (Jun. 1993).

[27] 山田晃久, 白川功, 築山修治: "CMOS 論理セルジェネレータの一手法," 電子情報通信学会技術研究報告, CAS87-114 (1987 年 8 月).

[28] A. Yamada, I. Shirakawa, S. Tsukiyama and S. Shinoda: "An expert system for mask pattern generator of CMOS logic cells," in *Proc. Joint Technical Conference on Circuits/Systems, Computers and Communications*, pp. 299-304 (Nov. 1988).

[29] 山田晃久, 築山修治, 白川功, 神戸尚志: "知識ベースシステムによる CMOS 論理セル自動生成," 電子情報通信学会論文誌 A, vol. J72-A, no. 12, pp. 1236–1242 (1988 年 12 月).

[30] 山田晃久, 築山修治, 白川功: "知識ベースシステムによる CMOS 論理セル自動生成," 電子情報通信学会春季大会, SA-7-5 (1989 年 3 月).

[31] I. Shirakawa, S. Tsukiyama, S. Shinoda, A. Yamada and T. Kambe: "An expert system for mask pattern generator of CMOS logic cells," in *Proc. 7th European Conference on Circuit Theory and Design*, pp. 324-328 (Sep. 1989).

[32] 山崎年樹, 山田晃久, 石浦菜岐佐, 白川功: "論理関数処理による条件分岐制御のあるデータパスのスケジューリング手法," 電子情報通信学会秋季大会, SA-3-3 (1993 年 9 月).

[33] 山田晃久, 山崎年樹, 神戸尚志, 石浦菜岐佐, 白川功: "条件分岐制御のあるデータパスのスケジューリング手法," 電子情報通信学会技術研究報告, VLD93-52 (1993 年 10 月).

[34] 山田晃久, 山崎年樹, 神戸尚志, 石浦菜岐佐, 白川功: "動作記述に条件分岐があるデータパスのスケジューリング手法," 電子情報通信学会回路とシステム軽井沢ワークショップ, pp. 49–54 (1994年4月).

[35] 山田晃久, 山崎年樹, 神戸尚志, 石浦菜岐佐, 白川功: "条件分岐を含む動作記述に対するスケジューリング手法," 情報処理学会DAシンポジウム'94, pp. 21–26 (1994年8月).

[36] A. Yamada, T. Yamazaki, N. Ishiura, I. Shirakawa, and T. Kambe: "Datapath scheduling based on integer nonlinear programming," in *Proc. Symp. on Nonlinear Theory and its Applications*, pp. 291–294 (Oct. 1994).

[37] A. Yamada, T. Yamazaki, N. Ishiura, I. Shirakawa, and T. Kambe: "Datapath scheduling for conditional resource sharing," in *Proc. IEEE Asia-Pacific Conf. on Circuits and Systems*, pp. 169–174 (Dec. 1994).

[38] A. Yamada, T. Yamazaki, N. Ishiura, I. Shirakawa and T. Kambe, "Datapath scheduling for behavioral description with conditional branches," to appear in *IEICE Trans. Fundamentals*, vol. E77-A, no. 12 (Dec. 1994).

[39] M. Rim and R. Jain, "Representing conditional branches for high-level synthesis applications," in *Proc. 29th ACM/IEEE Design Automation Conf.*, pp. 106–111 (Jun. 1992).

[40] T. Miyazaki, "Boolean-based formulation for data path synthesis," in *Proc. IEEE Asia-Pacific Conf. on Circuits and Systems*, pp. 201–206 (Dec. 1992).

[41] 黒田忠広, 桜井貴康: "回路設計の指針を提示, 相次いで登場する新技術を体系化," 日経マイクロデバイス別冊低電力LSIの技術白書…1ミリ・ワットへ挑戦, pp. 78–97, 日経BP社 (1994年).

[42] J. H. Kim and J. McDermott: "TALIB: An IC layout design assistant," in *Proc. National Conf. on Artificial Intelligence*, pp. 197–201 (1983).

[43] J. H. Kim, J. McDermott and D. P. Siewiorek: "Exploiting domain knowledge in IC cell layout," in *IEEE Design and Test*, pp. 52–64 (Aug. 1984).

[44] J. H. Kim and J. McDermott: "Computer aids for IC design," in *IEEE Software*, pp. 38–47 (Mar. 1985).

[45] "VAX OPS5 Reference Manual," DEC (Sep. 1985).

[46] 吉村紀美: "ルール・ベース・エキスパートシステム・システム構築ツールの「基本形」OPS5," 日経コンピュータ別冊 (1985年11月).

[47] 稲葉則夫: "商品化相次ぐエキスパート・システム開発用ツールを比較する," 日経エレクトロニクス, pp. 153–175 (1985年11月).

[48] 辻井潤一: "プロダクションシステムとその応用," 情報処理, vol. 20, no. 8, pp. 735–743 (1979年8月).

[49] 伊理正夫, 白川功, 梶谷洋司, 篠田庄司他: 演習グラフ理論 —基礎と応用—, コロナ社 (1983年).

[50] S. B. Akers: "Binary Decision Diagrams," *IEEE trans. Comput.*, vol. C–27, no. 6, pp. 509–516 (Jun. 1978).

[51] R. E. Bryant: "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, (Aug. 1986).

[52] 石浦菜岐佐: "二分決定グラフとは," 電子情報通信学会回路とシステム軽井沢ワークショップ, pp. 155–160 (1992年4月).

[53] S. Minato, N. Ishiura and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," in *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 52–57 (Jun. 1990).