修 士 論 文

同期・非同期混合回路方式とその設計手法

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

加藤 孝太郎

2014 年 3 月

修 士 論 文

同期・非同期混合回路方式とその設計手法

指導教員　金子峰雄 教授

審査委員主査　金子峰雄 教授
審査委員　田中清史 准教授
審査委員　井口寧 教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1210018 加藤 孝太郎

提出年月: 2014 年 2 月

概 要

現在のディジタル回路設計は, 設計, 検証の容易さや回路規模などから, グローバルクロック信号を用いた同期式回路方式が主流である. しかし, 半導体集積回路技術の進歩に伴う素子の微細化とシステム大規模化が進むにつれて, 製造時の素子のばらつきや動作環境変化に起因する遅延変動が増大している. 一方で, クロックレスな非同期回路方式は, ハンドシェイク通信によって回路を制御するため, 遅延変動に伴う問題が発生しない. しかしながら, 非同期式回路設計では, ハンドシェイクプロトコルにより速度及び面積オーバーヘッドが大きい. これにより, 遅延量が大きくならない小規模な回路では, 同期式回路に優位性があり, 大きな遅延パスを含む大規模回路や小規模回路であっても, 環境の変化に応じて処理遅延が大きく変動する場合には, 非同期回路設計の方が有利であると予測される.

そこで本研究では, 同期式回路と非同期式回路の利点を活かすことを目的に, 一つの計算アルゴリズムを実行する回路ブロック内に同期・非同期回路を混合させる設計手法の提案を行う. 提案した設計では, 同期回路部がクロック信号によって状態遷移する FSM にて制御され, 非同期回路部が Q 素子のネットワークによって制御されることを基本とし, 両者が連携して正しく計算を実行するためのレジスタ転送レベルの制御構造を「制御器動作グラフ」と名付けたグラフ構造にて表現する. これにより, 同期・非同期間で共用されるデータに付随する制御や, 資源割り当てに付随する制御を表現することができ, 同期・非同期回路の混合を可能にしている.

目 次

第1章　はじめに

1.1　背景と目的

現在のディジタル回路は, グローバルクロック信号を用いて状態遷移を繰り返す同期回路方式が主流であり, 信号のレジスタへの書き込みは, クロック信号によって定められたタイミングにおいてのみ行われてきた. そのため, 同期式回路は, あるクロックの立ち上がりから次のクロックの立ち上がりまでの間に, 書き込まれるべき信号がハザードを持っていたとしても, 次のクロックの立ち上がりまでに, 正しい信号値が確定すれば, それのみがレジスタに書き込まれる. また, クロックサイクルをベースとした回路の動作検証の容易さも同期式回路の利点であると考えられる. しかし, 半導体集積回路技術の進歩に伴う素子の微細化とシステムの大規模化が進むにつれて, 製造時の素子のばらつきや動作環境変化に起因する遅延変動が増大している. これにより, 同期式回路のようなグローバルクロックを用いて, 回路全体を制御する方式では, 信号伝搬においてタイミング誤りなどの問題を引き起こす.

一方で, クロック信号を使用しない非同期回路方式は, 要求信号・応答信号をやり取りするハンドシェイクによって回路を制御するため, 遅延変動に伴うタイミング問題が発生しない高信頼なシステムを実現でき, 与えられた計算を行うときのみ信号遷移を行うため, 回路全域へクロック分配を行う同期式回路に比べ電力消費を低減することができる [1]. しかしながら, 非同期式回路設計では, ハンドシェイクプロトコルに伴う遅延が性能に対するオーバーヘッドとなるため, 速度性能を得ることが難しく, また, 信号値の変化を持って信号値の到着を認識するため, ハザードが発生しない回路設計を行う必要があり, 面積オーバーヘッドも大きい. さらに, クロックを使用しない非同期式回路は, ハザードの発生が回路全体の誤動作となるため, ハザードのない回路設計を行う必要がある [2]. これにより, 遅延量が大きくならない小規模な回路では, 同期式回路に優位性があり, 大きな遅延パスを含む大規模回路や小規模回路であっても, 入力データに依存して処理遅延が大きく変動する場合には, 非同期回路設計の方が有利であると予測される. 従って, 同期式回路と非同期式回路を細粒度で混合することで, 遅延変動に対して正常動作が保証でき, 面積・動作速度・電力効率の良い集積回路が実現可能であると考えた.

そこで, 本研究では, 同期式回路と非同期式回路の利点を活かすことを目的に, 一つの回路ブロック内に同期式回路と非同期式回路を混合させた回路方式の提案を行う. 現在, 同期・非同期混合について大域非同期局所同期 (Globally Asynchronous Locally Synchronous:GALS) システム [3] と呼ばれる設計手法がある. これは固有のクロック信号によって駆動しているいくつかの同期式回路ブロックを非同期バスで接続したものであり, 各通信回路ブロック間の大きな配線遅延が律速遅延となることを防ぐ技術である. 本稿は, この GALS を含み, より一般的な同期・非同期混合回路の合成について論じ, 将来の最適

混合検討を期待するものである. 提案した設計では, 同期・非同期回路の制御器間の通信を行うため, 高位合成の枠組みで, 新規に制御器を合成する. これにより, 同期・非同期回路の制御器間の通信も含めた実行スケジューリング及び, アロケーションが可能となる. これと並行して, 同期・非同期混合回路の動作確認と評価のために, イベントドリブン型のマクロタイミングシミュレータを開発し評価を行った.

1.2 論文構成

本稿の構成を以下に示す.

第 2 章 2 線 4 相非同期式回路

本研究で使用する非同期式回路の回路構成, 素子及び, その動作について説明する.

第 3 章 同期・非同期混合回路

同期式回路と非同期式回路のスケジューリングの違いから, 同期・非同期式回路双方を 1 つの回路ブロック内に合成するための設計手法について解説する.

第 4 章 リソース共有

第 3 章で提案した, 設計手法を基にして, 同期・非同期混合回路のリソース共有方法について説明し, さらに, リソース共有率を高める工夫として考案したマスタースレイブ型レジスタについて説明する.

第 5 章 動作解析

第 3 章, 4 章の設計手法及び, リソース共有方法に従って設計した同期・非同期混合回路の動作解析を行うシミュレーターの概要と, そのアルゴリズムについて説明する.

第 6 章 実験と考察

第 5 章で説明した動作解析シミュレーターを用いて, 同期・非同期混合回路の動作及び, 各配線や部品の遅延値を変更しつつ, ある計算アルゴリズムを実行した際の, 計算アプリケーション実行時間に対する評価, 考察を行う.

第 7 章 まとめと今後の課題

最後に, 本研究のまとめと今後の課題を示す.

第2章　2線4相非同期式回路

現在, ディジタル論理回路で最も多く使用されている同期式回路は, 図 2.1 の (a) のように大域クロック（*global clock*）を用いて, クロック信号のエッジ（信号電圧のレベルが High から Low, あるいは Low から High へと遷移する瞬間）の変化をデータ書き込みのタイミングとして定めている. これにより, 順序回路内の各部で同期をとりながら, 様々な信号を正しいタイミングで伝達することが出来る. これに対し, 非同期式回路は, 図 2.1 の (b) のように各レジスタ間をクロックの代わりに「Handshake」と呼ばれるローカルな繋がりを用いて, 論理回路の各部のタイミングを合わせ, 同期をとりながら動作する. 例えば, Register1, Register2 間（called a “ channel ” or “ link ”）のチャネルコミュニケーション [4] は, 2 本以上の信号線を用いたハンドシェイクプロトコル（handshake protocol）により実現できる. 1 本あるいはそれ以上の信号線は, データ通信を要求（Request）するのに用い, 他の信号線は通信の完了に対する応答（Acknowledge）するために用いる.

2.1　回路モデル

2.1.1　遅延モデル

ディジタル回路設計をする場合, 素子及び配線の遅延に関する何らかの仮定が必要である. 非同期式回路の最大の特徴が配置, 配線, 動作環境の要因による遅延変動を設計時に精密に予測する必要がない点であることを考えると, 「遅延の大きさは有限であるが上限は未知である」と仮定した Delay Insensitive model(DI モデル) を使用することが非同期式回路の設計において理想的であると考えられる. しかしながら DI モデルの下では, 分岐と単一出力素子だけを用いる場合, 実用的な回路は構成できないことが知られている [5]. そこで本研究では, 「遅延の大きさは有限であるが上限は未知であり, 分岐後の配線遅延は等しい」といった仮定を持つ Quasi DelayInsensitive model(QDI モデル) を使用する. 図 2.2 に QDI モデルの当時分岐を用いた信号伝搬の例を示す. X0 は gate1 の出力信号遷移であり, X1 は X0 が伝搬して gate3 へ入力される個所の信号遷移である. X2 も同様に X0 が伝搬して gate2 へ入力される個所の信号遷移である. Y0 は gate2 の出力の信号遷移であり, Y1 は Y0 が伝搬して gate3 へ入力される信号伝搬である. 各信号の伝搬遅延は, delay(X0 → Y1) = delay(X0 → X2)+delay(X2 → Y0)+delay(Y0 → Y1) のように表すことができる. QDI モデルによる等時分岐を適用すると, X1 と X2 の伝搬遅延は delay(X0 → X1) = delay(X0 → X2) である. さらに, delay(X2 → Y0) ＞ 0, delay(Y0 → Y1) ＞ 0 であることから, delay(X0 → X1) ＜ delay(X0 → Y1) が成り立つ. これにより, X1 と Y1 の間には因果関係はないが QDI モデルを用いて等時分岐を仮定することで, X1 が Y1 よりも早く gate3 に入力されることを保証することができる [6].

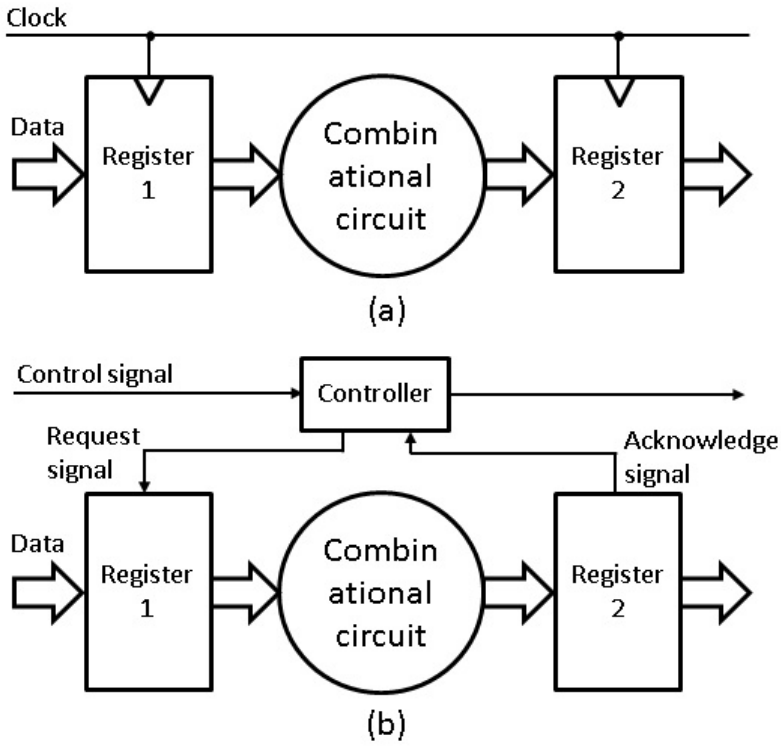

図 2.1: レジスタ間のデータ転送

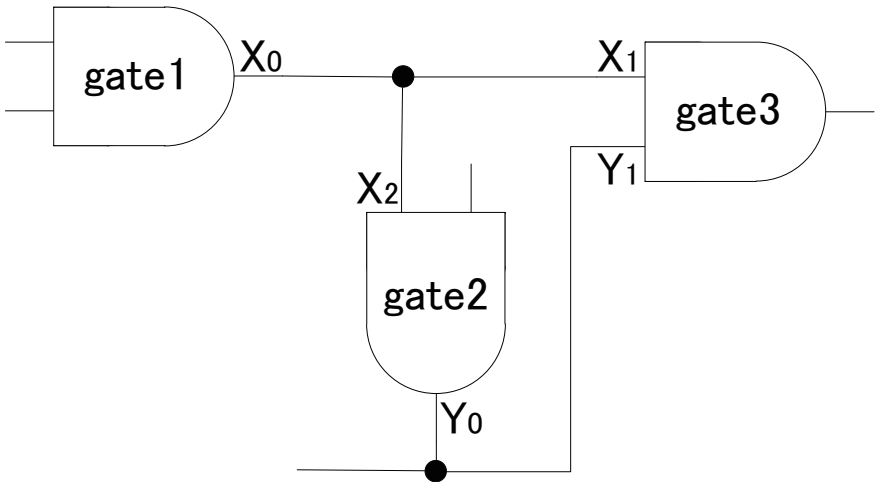

図 2.2: 等時分岐による信号伝搬

2.1.2 マラーの C 素子

C 素子は, 非同期式回路で最も多く使用される記憶素子の一つである. 図 2.3 に素子のシンボル表現 (a) とトランジスタレベルの実現例 (b) を示す. C 素子は, 表 2.1 のように, 入力 a と b の両方が共に 1 であれば出力は 1, 両方が共に 0 であれば出力 c は 0 を出力する. それ以外の入力では, 出力は変化せず, 直前までの出力を保持する [2].

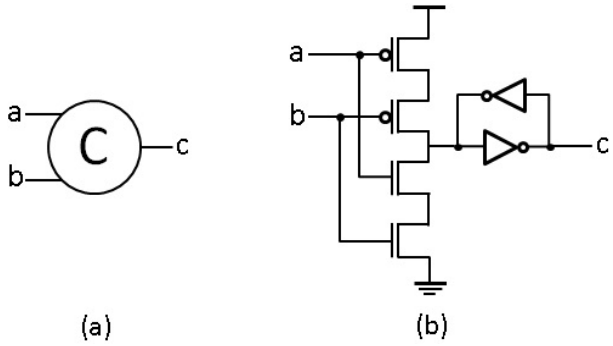

図 2.3: マラーの C 素子 (a) シンボル表現, (b) トランジスタレベルの実現例

表 2.1: C 素子の動作

入力 a	入力 b	出力 c
0	0	0
1	0	不変 (c)
0	1	不変 (c)
1	1	1

2.1.3 4 相非同期式回路

本研究で使用する非同期式回路は, 図 2.4 のように制御部とデータパス部からなるものとする. 各演算器は制御器内の Q 素子と呼ばれる回路ブロックによって制御される. Q 素子は 2 つの NAND ゲートと 1 つの C 素子によって構成されている. C 素子は全入力信号値が一致した時のみ出力値がその入力値に遷移し, 異なる入力信号値がある場合は, 直前まで出力値を保持する記憶素子である.

Q 素子は入力信号 in が $0 \rightarrow 1$(in+) に遷移すると要求信号 req が $0 \rightarrow 1$(req+) になり, レジスタ内のデータを演算器に出力し, その演算結果がレジスタに書き込まれ応答信号 ack が $0 \rightarrow 1$(ack+) に遷移する. この動作を Working Phase と呼ぶ. さらに, Q 素子は ack を受信すると req を $1 \rightarrow 0$(req-) にし, レジスタから演算器へのデータ出力を止め, 演算結果を書き込んだレジスタからの ack が $1 \rightarrow 0$(ack-) になる. この動作を Idle phase と呼ぶ. Q

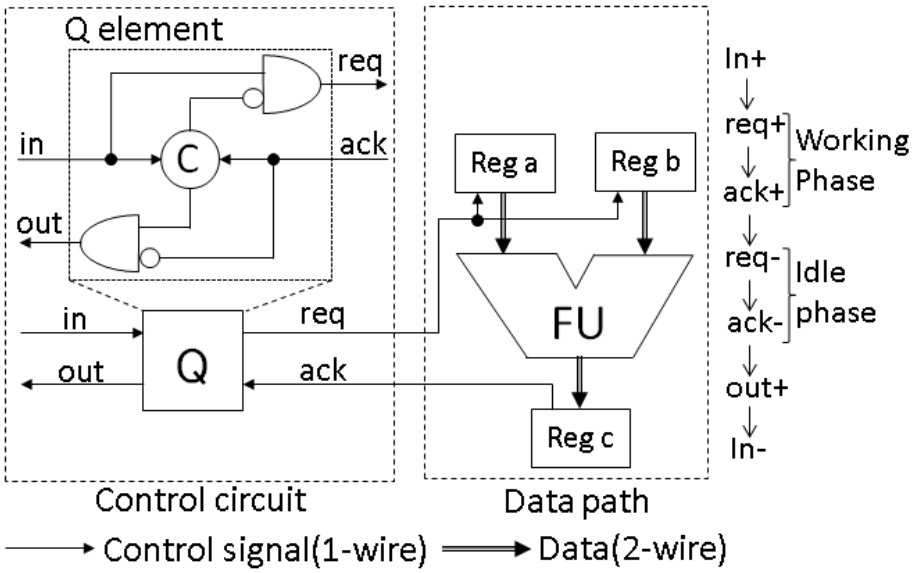

図 2.4: 4 相非同期式回路の制御部とデータパス部間の通信

素子は ack-を受信することで出力 out を 0 → 1(out+) にし, 出力先につながる別の Q 素子に制御を移す. この Working Phase と Idle phase を交互に繰り返す回路を 4 相非同期回路と呼ぶ [4].

2.1.4 2 線方式

一般に同期式回路は, 図 2.5(a) のようにクロックの遷移タイミング（立ち上がり遷移, あるいは立下り遷移）によってデータの到着を知ることができる. これに対しクロックを使用しない非同期回路は, レジスタ間のデータの到着を決めるタイミング情報をデータの中に埋め込むため, データの符号化が必要となる. そこで, 本研究では, 1 ビットのデータを 2 本の信号線を用いて表現する「2 線方式」と呼ばれる符号化方式を使用して, レジスタ間のデータの到着を判断する.

2 線方式は, 1 本を「肯定線」もう 1 本を「否定線」として使用する. 表 2.2 のように, データ表現”00”をスペーサ, ”01”を有効データ 0, ”10”を有効データ 1 として扱う. さらにこのスペーサ 00 を図 2.5(b) のように, すべての有効データの間に挿入することで, 00 から 01 の変化を「0 の到着」, 00 から 10 の変化を「1 の到着」として判断する. このように, スペーサ 00 というタイミング情報を有効データの間に挿入していくことで, 有効データがレジスタに到着した時に応答信号（Ack）を High にし, スペーサ 00 がレジスタに到着（演算で使用した配線経路が初期化された）ならば応答信号（Ack）を Low にすることで, この応答信号（Ack）を制御器側で受信することで, レジスタ間のデータ転送を制御することが可能となる.

また, 2.1.3 の 4 相非同期式回路と 2 線方式を合わせて, 「2 線 4 相非同期式回路」と呼び, 本研究ではこの回路モデルを使用する.

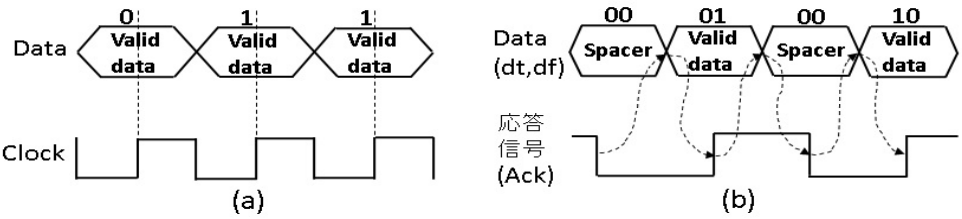

図 2.5: (a) 同期式回路のデータ転送, (b) 非同期式回路のデータ転送

表 2.2: 2 線符号化の状態遷移表

状態	データ表現	データ到着
Spacer	(0, 0)	
Valid "0"	(0, 1)	(0, 0) → (0, 1)
Valid "1"	(1, 0)	(0, 0) → (1, 0)

2.2 記憶素子

2.2.1 ラッチの基本構造

図 2.6 は本研究で使用した 2 つの入力ポートと 2 つの出力ポートを持つ 1 ビットレジスタの例である [7]. このレジスタに対して data A または data B から符号語が入力されると, 応答信号 ack A または ack B を High にする. また, スペーサ (0, 0) が入力されると, それぞれ応答信号 ack が Low になる. 一方, ラッチされたデータを出力する場合, 要求信号 req C または req D を Q 素子からの制御信号によって High にすることで, data C または data D からラッチされていたデータを出力する. また, 要求信号 req が Low になるとスペーサ (0, 0) を出力する.

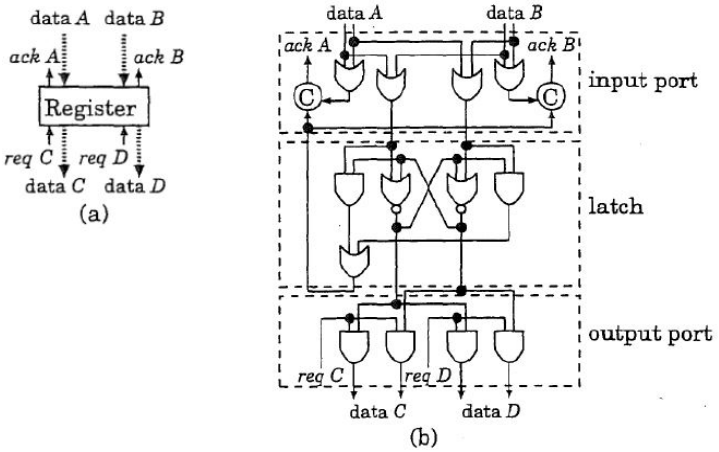

図 2.6: 2 入力 2 出力の 1 ビットレジスタ (a) シンボル表現, (b) ゲートレベルの実現例

第3章　同期・非同期混合回路

同期回路と非同期回路を 1 つの回路ブロック内に混合する際, クロック信号によって動作する同期式回路と, 要求・応答信号の因果関係によって動作する非同期式回路との間で, 必要なタイミング情報をやり取りする必要がある. そこで, レジスタ転送レベルにおいて, クロック信号とハンドシェーク信号をベースに, 同期・非同期回路の制御器間で通信を行い, タイミング誤りなく制御していくための新たな制御器を考案した.

3.1　計算アルゴリズムの切り分け

同期式回路と非同期式回路を 1 つの回路ブロック内に混合させるにあたり, 図 3.1(a) のようなある任意の大きさを持った計算アルゴリズムに対して, 図 3.1(b) のように, 同期演算として動作させる部分と, 非同期演算として動作させる部分の切り分けを行った. なお本稿では, 切り分けの最適化については議論せず, 切り分けが与えられるものとして議論を進める.

3.2　演算スケジューリング

従来の同期・非同期回路のスケジュールには, 以下のような大きな違いがある.

- 同期式回路 : 演算タイミングはデータ依存関係を考慮した, 演算のクロックステップへの割り当てで決まる.
- 非同期式回路 : 演算タイミングはデータ依存関係と演算の実行順で決まる（本研究では, 非同期式回路の各演算毎に図 2.4 に示した Q 素子を割り当てていき, それらの間の制御依存順序で演算タイミングを制御していく）.

このようなスケジュールを図 3.1(b) の計算アルゴリズムに適用した例を, 図 3.2 に示す. 図 3.2 の Q1, Q2, Q3 は, 図 3.1(b) の演算 O1,O2, O3 に割り当てられた Q 素子を示しており, 演算 O4, O5 は同期式回路の各クロックサイクル毎に割り当てられた演算を表している. また, 各記号に接続している矢印は, 各演算毎に必要なデータ（演算結果）を示している. この図 3.2 において同期演算とされた O4, O5 が, それぞれステップ 1, ステップ 2 で実行され, 非同期演算とされた O1, O2, O3 がそれぞれ, Q1 の入力 (d1, d2 の入力), Q2 の入力 (Q1 の完了), Q3 の入力 (ステップの完了と Q2 の完了) にて実行されることを示している.

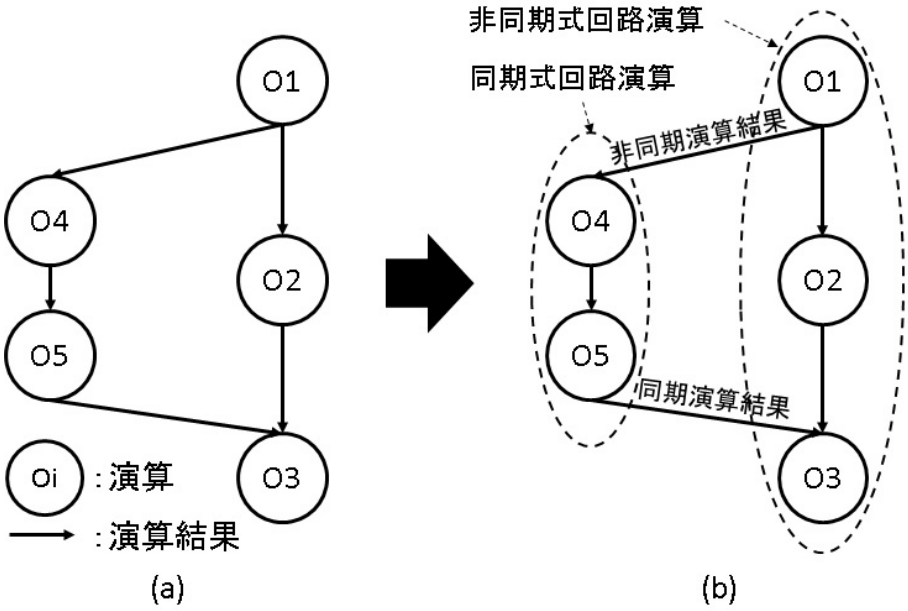

図 3.1: 計算アルゴリズムの切り分け（一例）

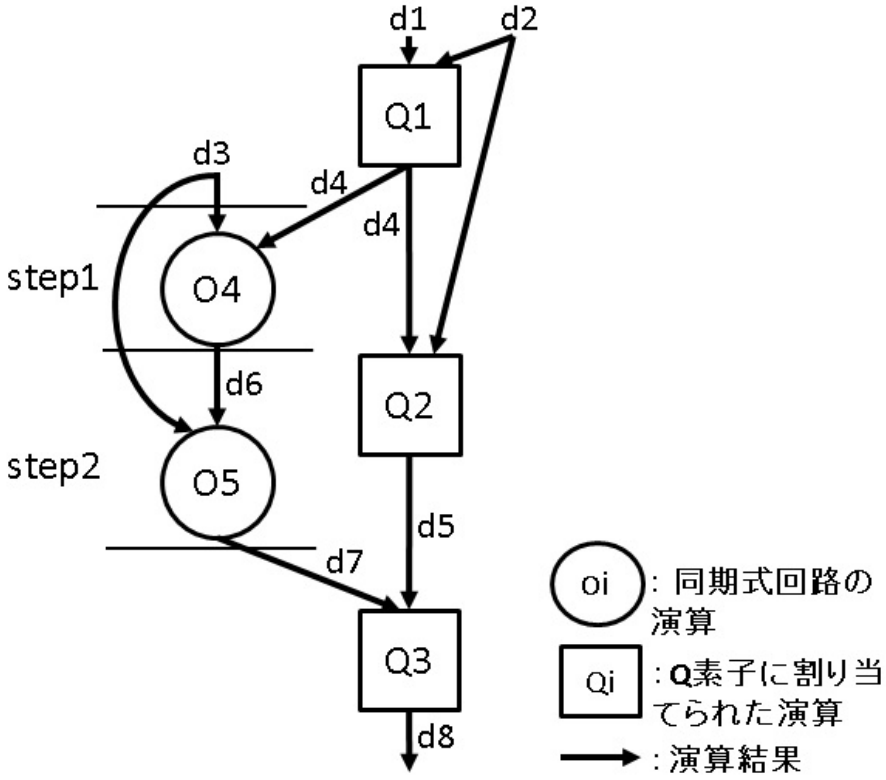

図 3.2: スケジュール済みデータフロー

3.3　制御器動作グラフ

図 3.2 の演算 O4, O5 が割り当てられている step1, step2 では, 各クロックサイクル毎に実行される演算が決定しているため, 資源制約のもとでレジスタや演算器の割り当てを行い, クロックサイクルに合わせて制御器を構成することは容易である. しかし, Q1, Q2, Q3 に割り当てられた非同期演算は, クロックサイクルのような定期的な動作順序は決まっていないため, データ d4 のように 1 つの演算結果を同期・非同期双方の演算で使用するタイミングやや, そのデータを使用した際の演算結果をレジスタに書き込むタイミングなどは, このデータフローからは知ることができない. そこで, 本研究では新規に (1) 同期・非同期双方の制御器間の接続方法と, (2) その接続方法表記方法の提案を行い, 同期. 非同期混合回路の制御器の詳細設計を行った.

3.3.1　演算スケジューリングの場合分け

同期・非同期それぞれの演算順序に着目し, 演算スケジューリングの場合分けを行うと, 図 3.3 のように 4 つの場合分けが可能である. 図 3.3（a), (b) は, 演算の前後が同期・非同期それぞれの演算が連続して実行する場合のデータフローを示しており, 図 3.3（c), (d) は, 演算の前後が同期演算の後, 非同期演算が行われる場合と, 非同期演算の後, 同期演算が行われる場合のスケジューリング済みデータフローを示している.

また, 図 3.3（a), (b), (c), (d) の詳しい動作の特徴を以下に示す.

- (a) の動作
  - 同期演算結果が書き込まれたレジスタにアクセスし, そのデータを同期演算で使用する. また, その演算結果ををレジスタに書き込む.
- (b) の動作
  - 非同期演算結果が書き込まれたレジスタにアクセスし, そのデータを非同期演算で使用する. また, その演算結果をレジスタに書き込む.
- (c) の動作
  - 同期演算結果が書き込まれたレジスタにアクセスして, そのデータを非同期演算で使用する. また, その演算結果ををレジスタに書き込む.
- (d) の動作
  - 非同期演算結果が書き込まれたレジスタにアクセスして, そのデータを同期演算で使用する. また, その演算結果をレジスタに書き込む.

3.3.2　制御器間の接続方法

本章では, 図 3.3 を基にして, 新規に同期・非同期回路の「制御器間の接続方法」と, その「表記方法」を提案する.

図 3.4(a), (c), (e), (g) は, 図 3.2 と同じスケジュール済みデータフローの例であり, 図 3.4(b), (d), (f), (h) は新規に提案する「制御器間接続方法」とその表記方法を示したものである. 以下に図 3.4 の詳しい説明を記載する.

図 3.4(a) 同期演算制御

図 3.4(a) のように同期演算結果を, 同期演算で使用する場合, 同期演算は 1 つのコントロールステップを 1 つの状態（1 つのコントロールステップにいくつかの演算が与えられていたとしても, 1 つの状態 Si として表現する.）とする FSM にして制御され, 図 3.3(b) のように状態遷移図にて記述される. また, 各同期式回路の状態間の矢印は状態遷移を表し, (b) のように描かれた図を「制御器間動作グラフ」と呼ぶ.

図 3.4(c) 非同期演算制御

図 3.4(c) のように非同期演算結果を, 非同期演算で使用する場合, 個々の非同期演算を制御する Q 素子に対応する Q 素子ノードを用意し, 演算間のデータ依存関係 (Oi → Oj) に対応して, 一方の Q 素子ノード (Qi) から他方の Q 素子ノード (Qj2) への有向辺を描き, これを「演算完了信号」と呼ぶ.

図 3.4(e) 同期・非同期間制御 (1)

図 3.4(e) のように同期演算 Oi の演算結果を, 非同期演算 Oj(制御器 Qj) で使用する場合, 非同期式回路側は同期演算が終了したことを知る必要がある. そこで, FSM の状態によって決まる FSM からの出力信号（あるコントロールステップで, 状態 Sk に割り当てられている演算を実行している場合は, 同期回路の FSM から Q 素子への出力が 0 であり, 状態 Sk+1 に遷移した段階で, 同期回路の FSM から Q 素子への出力が 1 になる）を Q 素子への入力として使用し, 同期演算の演算終了を伝達する.

図 3.4(g) 同期・非同期間制御 (2)

非同期演算 Oi(制御器 Qi) の演算結果を, 同期演算 Oj が使用する場合, Q 素子の出力 out を同期式回路の FSM の外部入力として使用することで, 非同期式回路の演算終了を同期式回路側に伝達する. このとき, 同期式回路の FSM を同じ状態 S_{k-1} でループさせておくことで, 非同期演算が確実に終了した後で, 同期式回路の演算 Oj を行うことが可能となる.

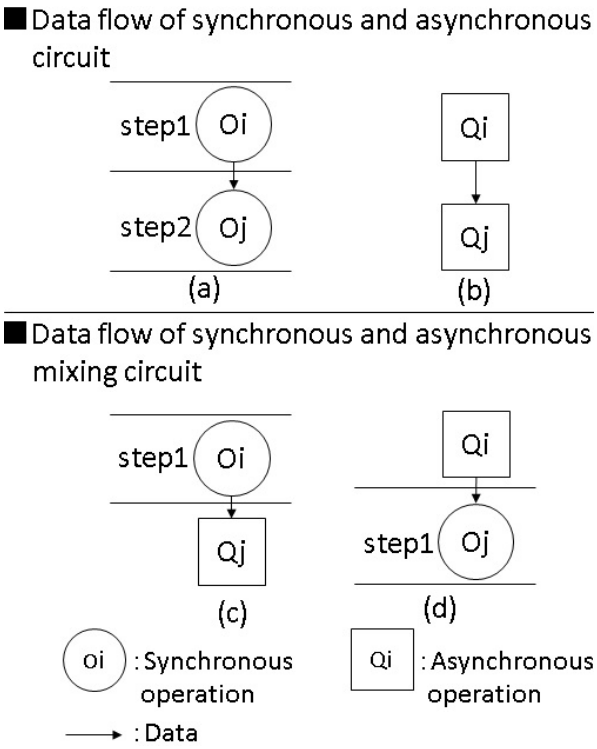

図 3.3: 同期・非同期混合回路のにおけるデータフローのパターン

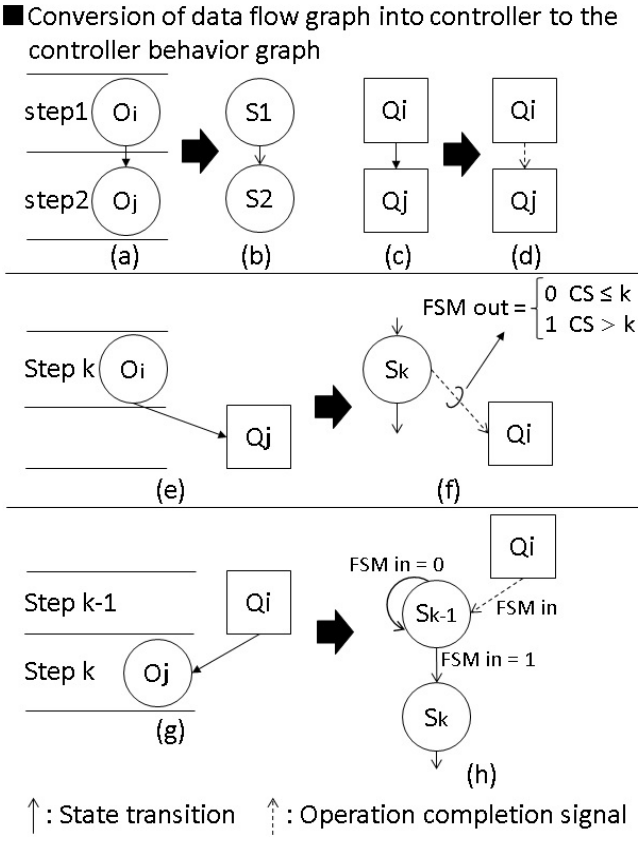

図 3.4: (a), (c) スケジュール済みデータフロー, (b), (d) 制御器動作グラフ

第4章　リソース共有

4.1　1線と2線の取り扱い

同期・非同期回路のデータパスは, それぞれ, 1線方式と2線方式によって構成されている. そのため, 同期・非同期混合回路においては, 1線式及び, 2線式データパスが混載するため, 回路ブロック中で資源共有を行う場合, 1線から2線または, 2線から1線への変換を行い, 同期・非同期混合回路の各素子にデータを入力する必要がある. こうした変換回路の具体例を図 4.1 に示す. 図 4.1 のように2線式データパスを1線式データパスに変換する場合, 2線データパスの肯定線 (T) のみを使用することで, 1線式データパスとして使用する. また, 1線式データパスを2線式データパスに変換する場合, 1線式データパスを2本の信号線に分け, 片方を NOT ゲートで反転させることにより, 2線式データパスと同じ符号語として使用する.

4.2　提案レジスタ

ある資源制約のもとでスケジュールを行う場合, (1) 図 4.2 のように, あるレジスタから出力したデータが, 演算器などの組み合わせ回路を通過して, その結果が出力元のレジスタに再び書き込まれる場合が存在する. このとき, 非同期回路の Q 素子などでレジスタを制御している場合, レジスタへの制御が遅れると, 演算終了後に書き込んだデータで再び演算を行ってしまい, 書き込まれたデータで演算が実行されてしまい, 間違ったデータでレジスタが上書きされてしまう恐れがある. さらに, (2) 同期・非同期混合回路において, 同期式回路の演算結果を非同期式演算で使用する場合と非同期式回路の演算結果を同期演算で使用する場合が存在する.

そこで, 本研究で使用する同期・非同期混合回路において, 図 4.3 のように, 1線・2線方式の両方に対応したマスタースレイブ型レジスタを開発した. 図 4.3 のマスタースレイブ型レジスタは, 大きく分けて, 入力部（Input port）, ラッチ部（Latch）, ラッチ制御部（Latch control）の3つで構成されている. このレジスタのラッチ部は図 2.6(b) に示したラッチの基本構造を使用しており, Master 部, Slave 部共に同じ構造である. 入力部は非同

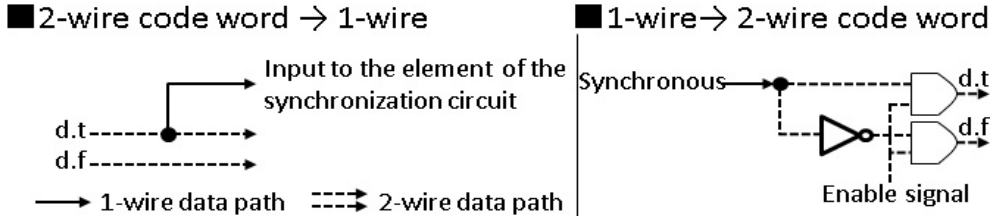

図 4.1: (a)：2線方式→1線方式の変換 (b)：1線方式→2線方式の変換

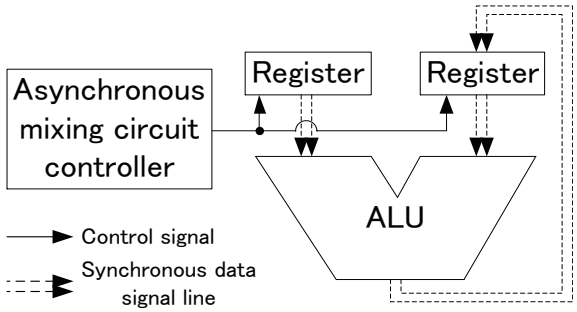

図 4.2: 演算結果がループするレジスタ間のデータ転送

期式回路として動作するときの 2 線方式によるデータ入力と同期式回路として動作する場合の 1 線方式の両方に対応できるように, 別々の入力ポートを使用している. この別々の入力ポートから入力されたデータを, 同期回路, 非同期回路それぞれの制御器の動作タイミングでラッチするために以下のような制御をおこなった.

- ポート 8 あるいは 9 の入力が 0 の場合
  - ポート 0 あるいは 2 から C 素子を通してデータを入力する.
- ポート 8 あるいは 9 の入力が 1 の場合
  - ポート 12 あるいは 13 から, 1 線式のデータとして入力されたデータを 2 つに分け, 片方を反転させて, AND ゲートを通して入力する.

この制御によって, 同期回路の制御器から動作させる場合（クロック信号のタイミングなど）と, 非同期回路の制御器から動作させる場合（Q 素子からの制御信号など）を区別することができる. また, このような同期回路からの制御と, 非同期回路からの制御を行う場合, 以下のように制御する必要がある.

- 同期回路の制御器からのマスタースレイブ型レジスタ制御
  - クロック信号のタイミングでデータを取り込む場合, 図 4.3 上部の C 素子（入力側）内の値が 0 である（idle 化された後）.
- 非同期回路の制御器からのマスタースレイブ型レジスタ制御
  - 同期回路の制御器からの制御信号の入力がない（ポート 8 あるいは 9 の入力が常に 0 の状態）.

さらに, このレジスタの Master 部, Slave 部を制御するために ”負縁トリガー型 2 相制御モジュール ” を使用する [3]. 図 4.4 に負縁トリガー型 2 相制御モジュールの回路図とその略記号を示す.

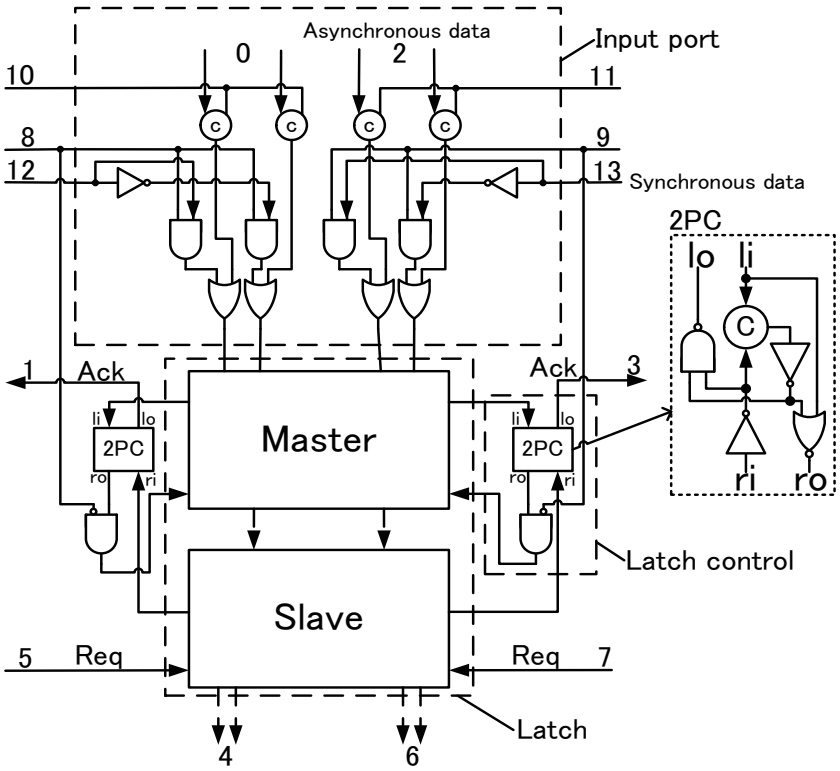

図 4.3: 同期・非同期混合回路に対応したマスタースレイブ型レジスタ（同期・非同期データそれぞれ 2 入力, 2 出力）

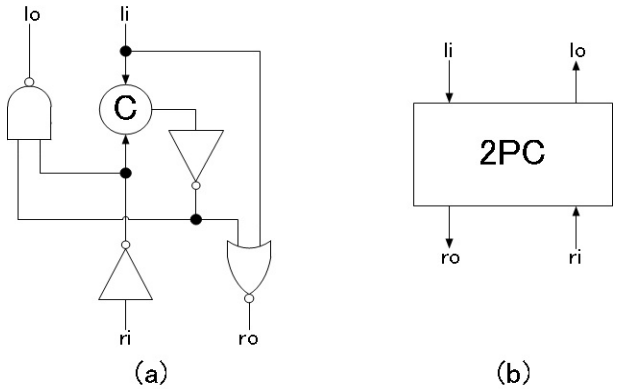

図 4.4: (a) 負縁トリガー型 2 相制御モジュール, (b) 略記号

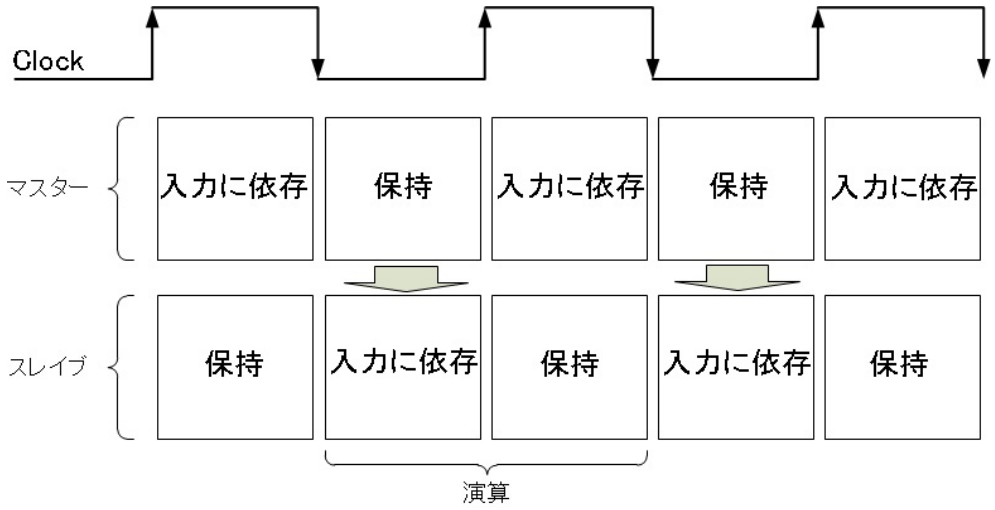

図 4.5: 従来の同期式回路で使用されているマスタースレイブ型レジスタの動作

4.3 従来の同期式回路で使用されているマスタースレイブ型レジスタとの動作の違い

図 4.5 に従来の同期式回路で使用されているマスタースレイブ型レジスタの動作, 図 4.6 に同期・非同期混合回路に対応したマスタースレイブ型レジスタの動作を示す. 図 4.5 に示した従来の同期式回路で使用されているマスタースレイブ型レジスタは, マスター側, スレイブ側が共に「データの保持状態」と「データの入力に依存する状態」を交互に繰り返して動作する. これに対し, 図 4.6 に示した同期・非同期混合回路に対応したマスタースレイブ型レジスタは, マスター側が「データの入力に依存する状態」とき以外は, マスター側, スレイブ側が共に「データの保持状態」を維持する. この保持状態の時にマスター側からスレイブ側にデータの転送が 1 度だけ行われ, その後は転送を行わない. データの転送が 1 度だけとは, スペーサ (0, 0) が入力され, マスター側からスレイブ側にラッチしたデータを移行する際に, 図 4.4 の負縁トリガーによって, 一度データが移行すると, 図 4.4(b) の li が High になり, ro も High になる. これによりマスター側の出力が Low(0, 0) になり, マスター側からスレイブ側へのデータの移行がストップする. さらに, スレイブ側にマスター側からの (0, 0) が入力されると, 図 4.4(b) の ri が Low になり, lo が High になることで応答信号 Ack を返す. また, 図 4.6(b) のように同期回路の制御器から制御される場合, スレイブ内のデータは出力し続ける.

4.4 マスタースレイブ型レジスタの制御方法と回路への効果

図 2.6 に示されている非同期式レジスタは, 符号語が入力されるとその符号語をラッチする. しかし, マスタースレイブ型のレジスタを用いたレジスタ間のデータ転送を行うためには, 制御器からの信号で, レジスタ内のデータの読み書きをそれぞれ制御する必要がある. そこで, 図 4.3 のマスタースレイブ型レジスタに, 図 4.7 のようなゲート素子を追加

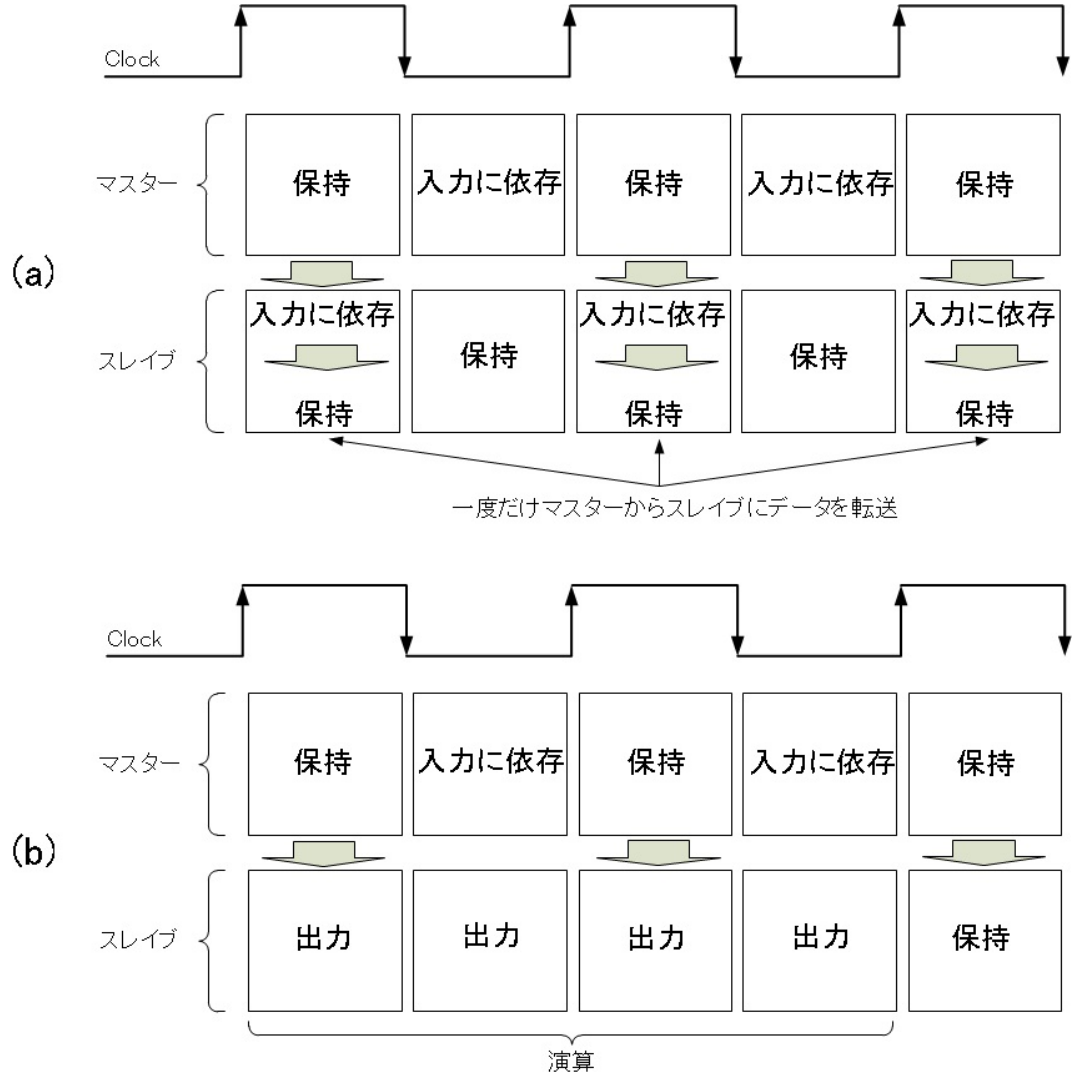

図 4.6: (a) 同期・非同期混合回路に対応したマスタースレイブ型レジスタの基本動作, (b) 同期回路の制御器からの制御する場合の動作

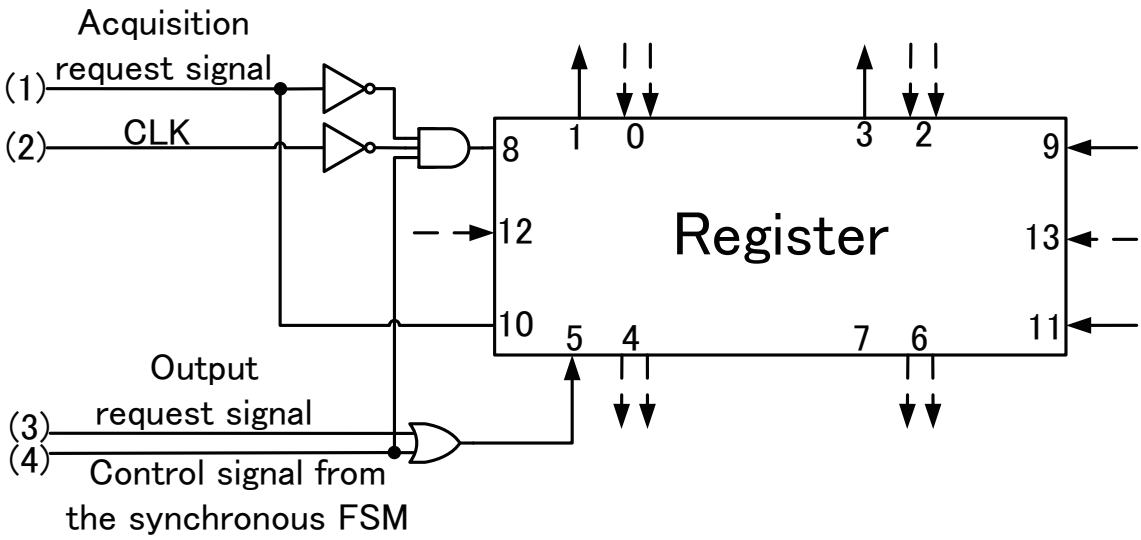

図 4.7: マスタースレイブ型レジスタの制御を行うゲート素子

し, 制御を行う（図 4.7 の Register 内のポート番号は, 図 4.3 のマスタースレイブ型レジスタのポート番号に対応している）. 以下に図 4.7 の制御動作を示す.

- 同期回路として制御する場合の動作
  - ポート (1), (3) の入力が 0 の状態で,「データの取り込み」を行う場合は, クロック信号の High, Low でポート (2) を制御し,「データの出力」を行う場合は, 同期回路の制御器からポート (4) を制御する.
- 非同期回路として制御する場合の動作
  - 「データの取り込み」を行う場合は, ポート (1) の ON, OFF を制御し,「データの出力」を行う場合は, ポート (3) の ON, OFF を制御する. 非同期回路として制御する場合は, (2) からのクロック入力は行わない.

4.5 演算器の共有

図 2.4 で示したように, 2 線 4 相非同期式回路は, Working Phase と Idle Phase の実行時にレジスタ間のデータパス経路が同一であり, 一度使用したデータパス経路を Idle Phase によって確実に初期化することが可能である. しかし, 同期式回路は, 各クロックサイクル毎にアクセスするレジスタや, 有効データが通る経路が異なる. これにより, 同期・非同期混合回路において, 同期演算と非同期演算が重複する演算経路を使用する場合, 非同期式回路の演算で使用するデータ伝搬経路を初期化することが現状では非常に困難である. そこで, 図 4.8 のように, 同期・非同期回路間で独立した演算経路を使用することとした. なお, 同期演算同士の演算器共有は, 両者の実行ステップに重複がないことが必要であり, 非同期演算同士の演算器の共有には, 演算順序の決定と, 対応する Q 素子間の演算完了信

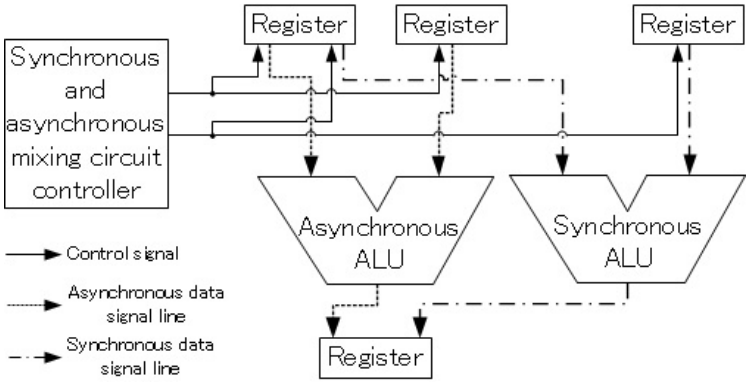

図 4.8: 同期・非同期混合回路のにおけるデータフローの一例

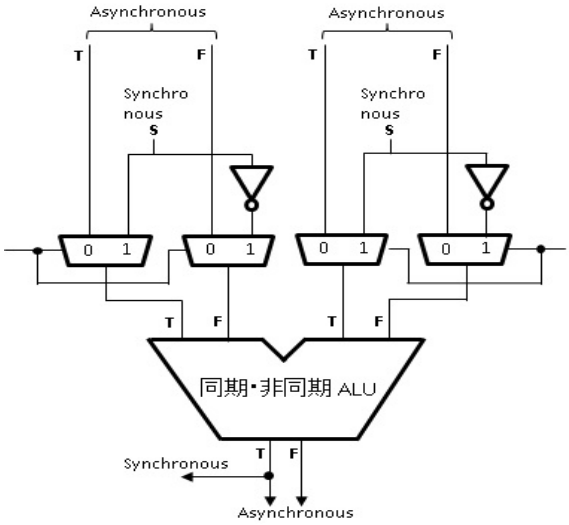

図 4.9: 1 線・2 線データパスに対応した演算器の一例

号の追加が必要であることは, 通常の同期回路構成, 非同期回路構成と同様である. また, 図 4.9 に 1 線・2 線データパスに対応した演算器を示す. 一般に非同期演算器は 2 線データパスの入出力に対応しており, 符号語を用いた演算が行われる. そのため, 非同期式回路で使用される 1 線式データパスのデータも, 図 4.9 のように入力データを 2 本のデータ線に分け, 片方に NOT ゲートを挿入することによって, 非同期演算器で正しい演算を行うことが可能である. これに対し, 同期演算器は 1 線式データパスの入出力に対応しているため, その演算器に対して, 2 線式データパスのデータを入力しても, 正しい演算結果が得られない. そこで, 同期式回路のある実行ステップに割り当てられている演算と, 非同期演算の使用するデータパスが同じ場合, 図 4.9 のようにマルチプレクサで, 演算器への同期・非同期回路双方の入力データを切り替えて使用することにより. 同期演算と非同期演算で演算器共有が可能であると考える.

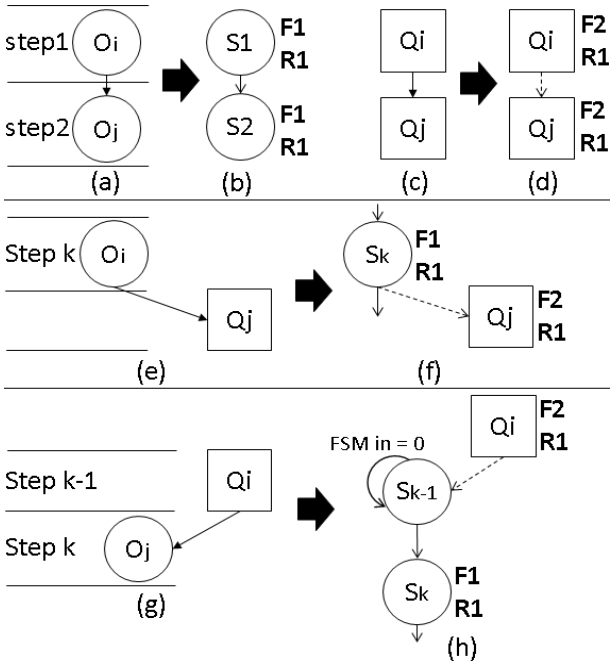

図 4.10: 演算を直列に実行した場合のレジスタの共有

4.6 レジスタの共有

図 4.8 に示したように, 同期式回路と非同期式回路を混合して設計する場合, 同期・非同期双方が同じデータパスを使用すると, そのデータパスを完全に初期化しない状態で, 非同期演算を開始する場合がある. そのため, 同期式回路と非同期式回路の間で, 同一のデータパスを使用する場合以外は, 演算器を共有することは現段階では困難である. そこで, 本章ではおもにレジスタ共有に着目し, 直列演算及び, 並列演算に対するリソース共有の可否について述べる.

- 直列演算におけるリソース共有

  図 4.10 は, マスタースレイブ型レジスタを使うことで可能なレジスタ共有のいくつかの例を示している. なお, 図 4.10 の制御器動作グラフでは, 各状態, または各 Q 素子に割当てられた演算で使用する演算器及びその演算結果を書き込むレジスタを, それぞれ, 状態または Q 素子の右側に記載している.

- 並列演算におけるリソース共有

  図 4.11 に同期回路と非同期回路の演算を並列実行時に, タイミング誤りを起こすレジスタ割り当ての例を示す. 図 4.11(a) のように, 並列実行している演算 Oi, O2 で使用しているレジスタ 1(データ 4), レジスタ 2(データ 2) に対して, 同期・非同期双方のタイミングで演算結果 (d6, d5) を書き込む動作をしようとすると, 完全な同時書き込みはできず, 演算への入力変数と演算の出力変数の間で生存期間の重複が生じてしまい, タイミング誤りを引き起こす. こうしたタイミング誤りを回避するた

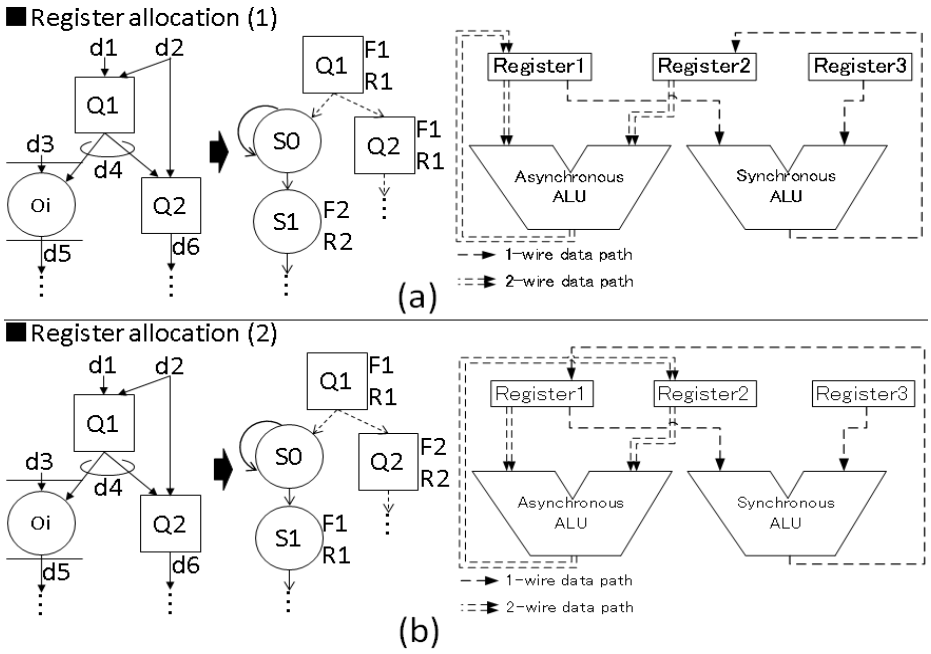

図 4.11: タイミング誤りを起こすレジスタ割り当て

めには, 図 4.12(a)(b)(c) のように, 並列実行する同期, 非同期演算が, 同期・非同期で共用されていないレジスタに値を書き込む構成をとるか, あるいは図 4.12(d)(e)(f) のように一方だけが同期・非同期共用レジスタに書き込むこととして, 必要な制御を追加する構成をとることになる. 後者の場合は, 図 4.13 のように, 図 4.3 で提案したレジスタの, 符号語の入出力を制御する端子に AND ゲートを接続し, 片方の端子に非同期式回路の制御器からの信号を入力し, もう一方を同期式回路の FSM からの演算完了信号とすることで, 同期式回路のある演算の終了をトリガにして, 非同期式回路の演算結果をレジスタ 1 に取り込む [7]. このような, ある入力条件がそろった後で, 次の演算を実行する流れを, 制御器動作グラフで表すと図 4.12(e) の gate のように描く.gate は, 非同期式回路の Working Phase(W2) の演算が終わった後その演算結果をレジスタ 1 に書き込まず, S1 に割り当てられている演算 O1 が終了した後に, レジスタ 1 への書き込みを行う様子を表している. また, I2 は演算後の Idle phase を示しており, Idle phase 終了の演算完了信号を再び同期式回路の制御器の外部入力として使用することで, FSM は状態 S3 に遷移する.

4.7 制御器の設計方法

図 3.2 のフローグラフを例に, 与えられた資源制約のもとで, 制御器動作グラフを設計するための手順を図 4.14 に示す. まず, 図 3.4 のデータ依存関係に従って, リソース共有を考慮せずに, 制御器動作グラフを構成し, 図 4.14(a) を得る. 次に図 4.14(b) のように状態 Si と Q 素子 Qi に対して, すべて異なるレジスタと演算器を割り当てる. ここで, 与えられた資源制約が演算器 2 個, レジスタ 3 個（図 3.2 のデータ d1,d2,d3 が格納されているレジ

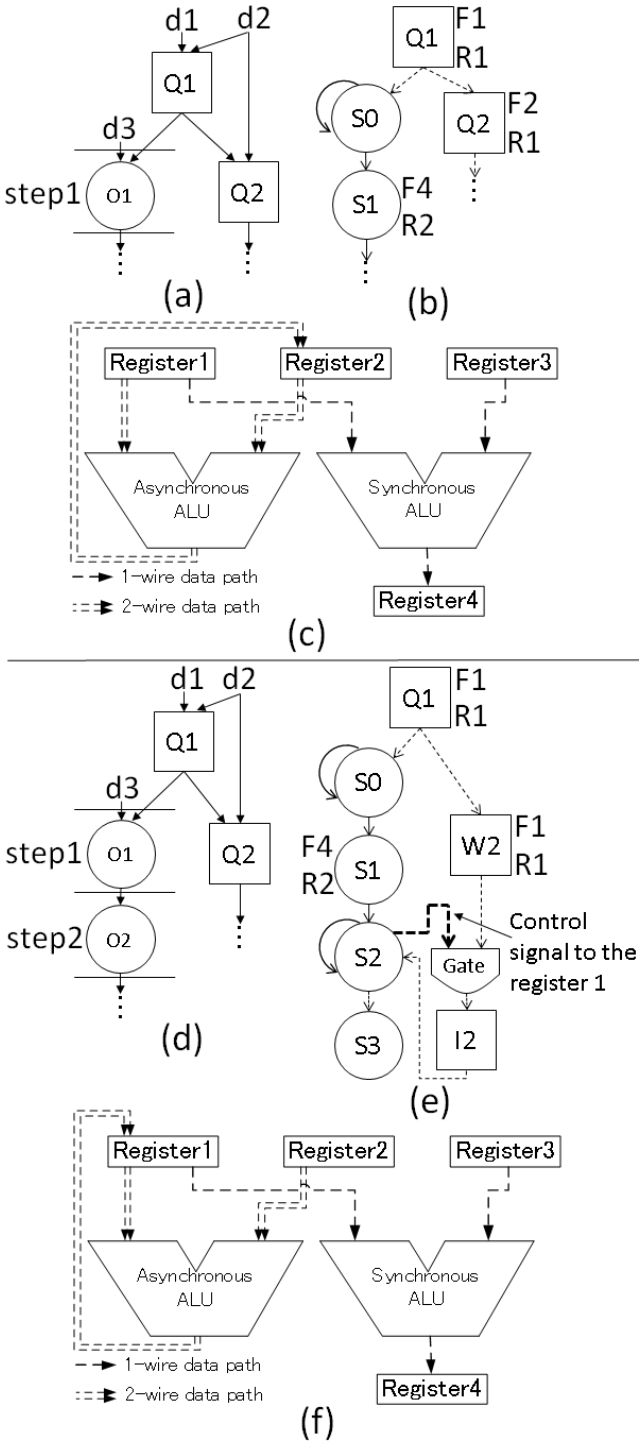

図 4.12: 演算を並列に実行した場合のレジスタ共有

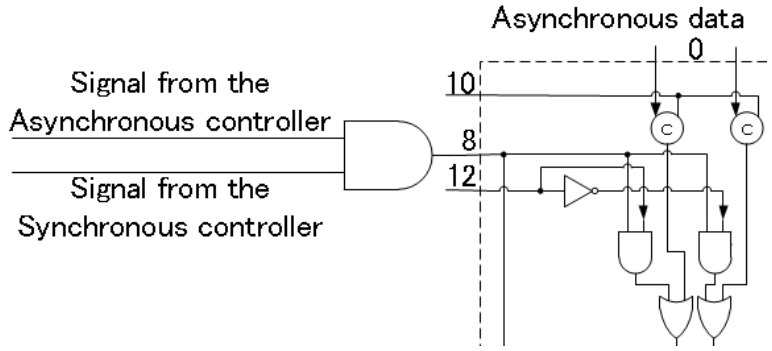

図 4.13: レジスタへの書き込み制御

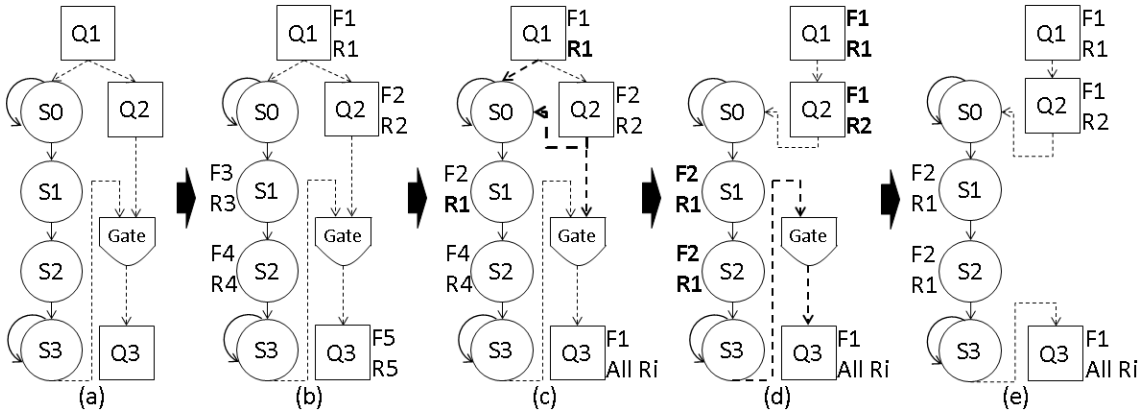

図 4.14: 同期・非同期混合回路の制御器の設計例 (1)

スタ）であるとするならば, 自動的に同期演算群と非同期演算群のそれぞれで演算が直列に実行されることになる. 一方, レジスタについて, 図 4.14(c) のように, Q1 と S1 に割り当てられた演算でレジスタ 1 を共有し, Q2 に割り当てられた演算の後に, S1 に割り当てられた演算を実行する. その際, 新しい演算順序として, Q2 から S0 へ演算完了信号を追加する. ここで, Q1 から S0 への演算完了信号と先程追加した演算完了信号が重複するため, 図 4.14(d) のように, 片方を削除する. 同様に, 図 4.14(c) の Q2 から Gate への演算完了信号と, S3 から Gate への演算完了信号も重複するため, 4.14(d) のように, 片方を削除する. 最後に 4.14(d) の Gate を削除し, 4.14(e) のような, 与えられた資源制約のもとで演算を直列に実行するための, 制御器動作グラフが完成する. また, 与えられた資源制約が演算器 2 個, レジスタ 4 個であるならば, 図 4.15 のように自動的に同期演算群と非同期演算群のそれぞれで演算が並列に実行されることになる.

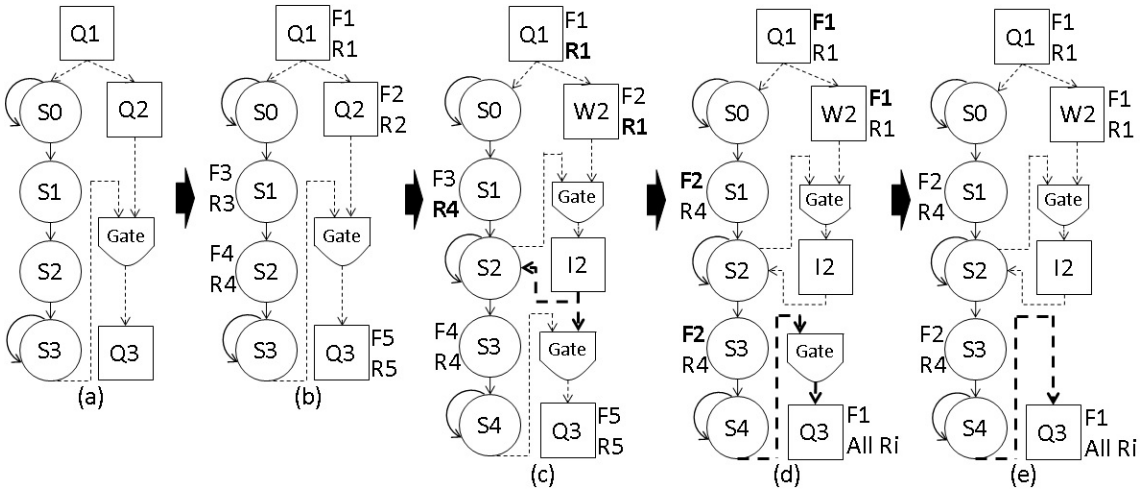

図 4.15: 同期・非同期混合回路の制御器の設計例 (2)

第5章　同期・非同期混合回路の評価

本研究では, まず (1) 第 3 章で提案した制御器動作グラフの作成方法にしたがって, ある計算アプリケーションを実行する同期・非同期混合回路の設計を行った. さらに, その (2) 回路の各「配線端子」, 「部品」及び, 「部品端子」に対してすべて番号付けを行い, その番号をテキスト化する. 最後に (3) 本研究で新規に開発した回路の動作解析を行うイベントドリブン型シミュレータで読み込んで, 各配線及び, 素子の遅延値をばらつかせることで, 遅延変動に対する同期・非同期混合回路の耐性及び, 基本動作を検証する. 本章では, 同期・非同期混合回路の動作解析を行うシミュレータについて述べる.

5.1　イベントドリブン型シミュレータの概要

本研究で開発したイベントドリブン型シミュレータの概要を図 5.1 に示す. また, 以下に示した動作手順で同期・非同期混合回路の動作確認および, 遅延解析を行っていく.

1. 初めに, 第 3 章で提案した制御器動作グラフの作成方法に従って作成した同期・非同期混合回路に対して, 図 5.1(a) のように各「配線端子」, 「部品」及び, 「部品端子」に番号付けを行う. さらに, このとき割り振られた番号を基に, 各部品端子番号に従って, 「部品番号」, AND ゲートや OR ゲートなどの「部品タイプ」, 1 線か 2 線かの「配線の種類」, 「配線番号」, 部品に接続されている「配線端子番号」を図 5.1(a) 下図のようにテキスト化し, シミュレータで読み取り, 配列に格納していく.

2. 1. で読み取った回路情報にしたがって「配線端子遷移リスト」を作成する. この配線端子遷移リストは, 回路情報から読み取ったすべての配線端子の遷移状態（0 or 1）を構造体配列の中に記憶する. この配線端子遷移リストを, 今後の処理で参照していくことによって, 各部品端子の遷移状態を知ることが可能である.

3. 次にシミュレータへの入力データとして「イベント」を作成する. このイベントは「配線番号」, 「配線端子番号」, 「配線端子遷移状態（配線端子に対してどのような制御信号及び, 演算結果が流れているかを表す値）」, 「遷移時刻（正規分布に従うランダムな遅延値であり, 配線及び, 部品毎に決められた平均, 分散, 上限, 下限を専用の関数に与えることで, 遅延値の計算を行い, それを遷移時刻として使用する）」の 4 つの値を持った構造体である. このイベントの具体的な例を図 5.1(b) に示す. 例えば, ある NOT ゲートに対してイベントを発生させる場合, 初期入力として「配線番号 1, 端子番号 0, 遷移状態 1, 遷移時刻 2.4」を入力データとすることで, シミュレータは「時刻 2.4 に配線番号 1 の端子 0 が, 0 から 1 に遷移する」ということを認識することができる.

4. 3. で入力されたイベントに対して, そのイベントが「配線端子 0 の遷移」なのかを判断する. もし, 配線端子 0 の遷移イベントならば, 「配線端子イベント関数」を用いて図 5.1(b) の (1) のように, 配線端子 0 以外の配線端子が 0 または 1 に遷移する新しいイベントを発生させる. この時, 遷移時刻は配線番号 1 の伝搬遅延時間分を足して表現する. また, もし入力されたイベントが配線端子 0 以外のイベントならば, それは配線端子の遷移でもある一方で, 部品端子の遷移を表すイベントでもある. そのためこのようなイベントは「部品端子イベント関数」を用いて, 図 5.1(b) の (2) のように部品への入力に対する処理を行い, 部品内の処理時間を次のイベントの遷移時刻に加えて追加する. 「配線端子イベント関数」と「部品端子イベント関数」についての詳しい説明は, 5.2 章で述べる.

このように, 配線端子の遷移なのか, 部品端子の遷移なのかを判断して新しいイベントを追加していくことで回路動作をシミュレートしていく. また, このイベントは遷移時刻の早いものから順に処理されいく.

5.2 アルゴリズム

本研究で使用するイベントドリブン型シミュレータは, 大きく分けて「配線端子遷移リスト関数」, 「配線端子イベント関数」, 「部品端子イベント関数」の 3 つで構成されている. この 3 つの関数についての説明を以下に示す.

1. 配線端子遷移リスト関数

   図 5.2, 図 5.3 に main 関数と配線端子遷移リスト関数のフローチャートを示す. 配線端子遷移リスト関数は, main 関数内の circuit_infomaition_read 関数で回路情報を読み取った後, その回路情報が持っている配線数及び, 配線端子数を用いてリストの作成を行う. 図 5.3 のように, 回路情報から読み取った配線端子数分の配線端子遷移リスト wtt[i].term を作成し, 部品個数と, その部品の端子数から配線端子がいくつ存在するか数え, check_array に格納していく. 次に check_array に格納された個数にしたがって配線端子数分のメモリを確保し, wtt_l[配線番号].term[端子番号] を作成する.

   この配線端子遷移リストは現在の配線端子の遷移状態を表しているため, 下記で説明する「配線端子イベント関数」や「部品端子イベント関数」などから参照することによって, 配線端子の遷移状態の重複や部品端子への入力状態などを判断することが可能である.

2. 配線端子イベント関数

   図 5.4 に配線端子イベント関数のフローチャートを示す. 配線端子イベント関数は, 図 5.3 の main 関数内で, 異常終了やプログラムの終了条件（決められた演算回数分のループ）に当てはまるまでループし続ける関数である. 図 5.4 のように, 登録されているイベント（配線番号, 配線端子番号, 遷移状態, 遷移時刻の 4 つを 1 つの構

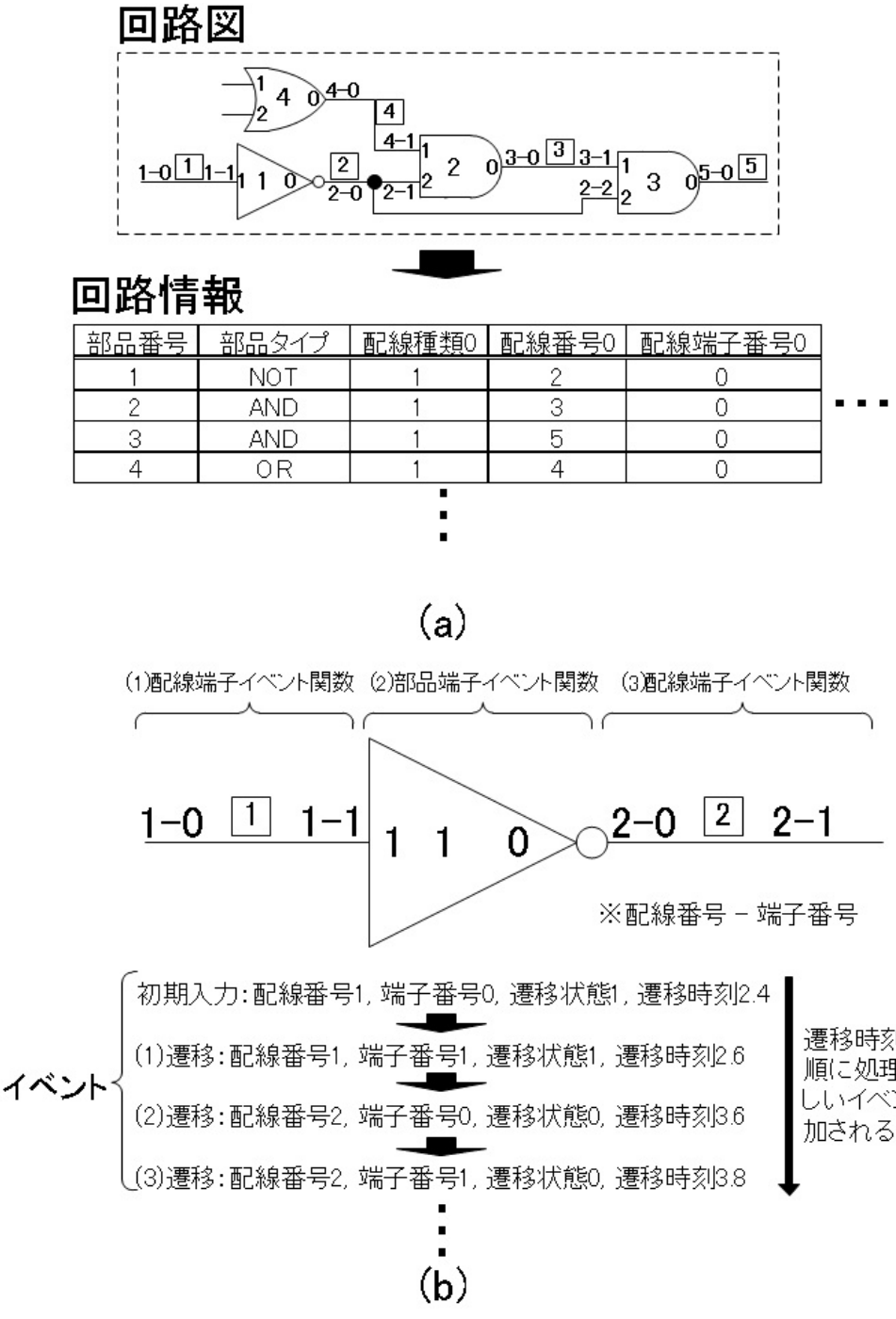

図 5.1: (a) ナンバリングされた回路図とテキスト化された回路情報の一例. (b) イベントの概念

造体として, 複数のイベントがリンク構造によって接続されている）中で, 遷移時間が一番早いものから順に選択（イベントは追加していく時点で, 上から一番遷移時間が早いもの順にソートされている）し, 変数に取り込む. 次に, もし同じ遷移状態を持ったイベントならば, 選択したイベントを削除し, 違う遷移ならば配線端子遷移リストの更新を行う. 更に選択したイベントの配線端子番号が 0 ならば, それは少なくとも部品端子の出力端子に接続されているため, 次の遷移はその 0 番の配線端子番号以外の遷移である. そのため, その配線タイプが 1 線なのか 2 線なのか判断し, それぞれの伝搬遅延を加算し, それを新しい遷移時刻として新しいイベントの追加を行う. また, もし配線端子番号が 0 以外の場合, それは少なくとも部品端子の入力端子に接続されているため, 部品端子イベント関数 parts_terminal_event を実行する.

3. 部品端子イベント関数

図 5.5 に部品端子イベント関数のフローチャートを示す. 部品端子イベント関数は, 図 5.4 の配線端子イベント関数で, 選択したイベントの配線端子番号が 0 以外の時に実行される. 図 5.5 のように, 始めに遷移時刻が最も早いイベントを選択し, その情報にしたがって図 5.2main 関数の circuit_infoemaition_read 関数の実行時に読み取った回路情報から部品タイプを検索し, それぞれの部品タイプ毎の関数を実行する. この部品タイプ毎の関数は選択された遷移時刻が最も早いイベントの配線端子番号と, 配線端子遷移リストを参照することによって, 部品の入力端子に対する出力端子の遷移を決定する. その一例を図 5.6 に示す.

図 5.6 は Q 素子への入出力を処理する関数である. すべての部品関数は, まず部品端子イベント関数 parts_terminal_event() から「部品番号」, 「部品端子番号」, 「部品端子個数」, 「選択したイベントの部品端子に接続している配線端子の遷移時刻」の 4 つの情報を引き継ぐ. この情報を基にして, 部品端子分のメモリを確保し, 配線端子遷移リストを参照しながら, 各部品端子の遷移状態を「部品端子の構造体 t_p_n_storage.term[i]」に格納していく. ここまでの処理はすべての部品毎の関数で共通しており, この部品端子の遷移状態の情報を基にして, 部品出力端子の遷移を決定していく. この Q 素子であれば, t_p_n_storage.term[0] が入力 in の入力端子, t_p_n_storage.term[1] が要求信号 req の出力端子, t_p_n_storage.term[2] が応答信号 ack の入力端子 t_p_n_storage.term[3] が出力 out 信号端子の遷移状態を表しており, c_value は, Q 素子内にある C 素子のラッチ状態を表している. この 4 つの部品端子の遷移状態に従って, 以下に示す Q 素子の条件文 (a), (b), (c) によって新たなイベントを追加していく.

(a) in+, req-, ack-, c_value=0 の時, req+のイベントを追加（つまり, 選択したイベントが, Q 素子の入力 in に接続されている配線端子の場合, その端子の遷移状態が 1 ならば, 図 2.4 の Q 素子の動作に従って, 要求信号 req を出力する端子に接続されている, 配線端子を 1 にするイベントを追加する）.

(b) in+, req+, ack+, c_value=0 の時, req-のイベントを追加

(c) in+, req-, ack-, c_value=1 の時, out+のイベントを追加

このように, 主に 3 つの関数を実行していくとで, 各配線及び, 部品の端子の遷移状態を随時追っていき, 同期・非同期混合回路の動作シミュレーションを行う.

5.3 信号伝搬速度における問題点とその改善策

本研究で開発したイベントドリブン型シミュレータは, 上記で説明したように, 遷移時刻が最も早い順にイベントとして登録され, 処理されていく. このため, 同じ配線端子であっても遷移状態が異なれば, 違うイベントとして登録される. しかし, 図 5.7(a) に示した演算器のように, 入力データ a, b の伝搬遅延（矢印横の数字）と, そのデータが演算器内を通過する伝搬遅延が異なる場合がある. 例えば図 5.7(b) のように, 演算に必要なデータ（図 5.7(b) ではデータ a, b）が入力され, 2 つのデータのうち伝搬遅延が小さい信号 b が出力 c に届き, 「不確定な演算結果」が出力される. その後, 2 つのデータが出力 c に到達し, 「正しい演算結果」が出力される. このとき, 不確定な演算結果と正しい演結果の出力が演算器内部の伝搬遅延によって前後し, 入力だけに着目して演算器の出力端子のイベントを追加しても, その結果が本来, データ入力のタイミングによって引き起こされた結果と異なるというシミュレータ上の問題が存在する. そこで, この問題の解決策として, 演算器の回路図上のシンボル表現を図 5.8(a) のように表現する. この演算器内の信号伝搬遅延は部品ではなく, 演算器を模した配線番号 3, 4 に与える. そのため, 一般のデータパスまたは制御信号線とは別に配線同士を接続する必要がある. しかし, 現段階では配線端子番号 0（部品の出力端子に接続されている配線端子）以外の端子はすべて部品の入力端子に接続されているものとして定めているため, 配線同士を接続して, その情報をシミュレータ内に取り込むことはできない. そのため, 遅延値が 0 の部品（AID-E1）を遅延値が異なる配線同士の接続に使用する（5.8(a) の配線番号 1 と 3, または配線番号 2 と 4）. また, 演算器内の信号伝搬遅延によって, 「正しい演算結果」と「不確定な演算結果」が入力データのタイミングに関係なく出力される現象を解決するために, 演算器内部の信号伝搬遅延をもった配線番号 3, 4 を, 遅延値が 0 の部品に接続する. これによって, 5.8(b) の 5.8(a) に対するイベントの追加例のように, 部品 AID-E1 を通過後, 演算器に模した配線番号 3, 4 を通過するデータの伝搬遅延に関わらず, 部品 AID-E2 通過後は, 正しい演算結果のイベントの次に不確定な演算結果が追加される. このようにイベントが追加される順序を固定することによって, 演算結果の出力後, その結果がレジスタに書き込まれる場合, 不確定な演算結果が先にレジスタに格納された後で, 正しい演算結果が書き込まれる.

更に, 制御信号線及び, データ信号線においても, イベント登録時の信号伝搬遅延によってはデータの追い越しが発生する. その様子を図 5.9 に示す. 例えば図 5.9(a) のように OR ゲートの各配線に対して遅延値を与えた場合, 入力信号 a が b よりも先に届き, OR ゲートの中と通過する際に, 内部の信号伝搬遅延によって入力順序とは逆に入力信号 b が a よりも先に出力 c に届いてしまう可能性がある（このような信号の流れをデータ追い越しと呼ぶ）. そのため, データ追い越しが発生した場合, 図 5.9(c) のように配線番号及び配線端子番号が同じで, 遷移状態は異なり, 遷移時刻が異なるイベントが追加される. このま

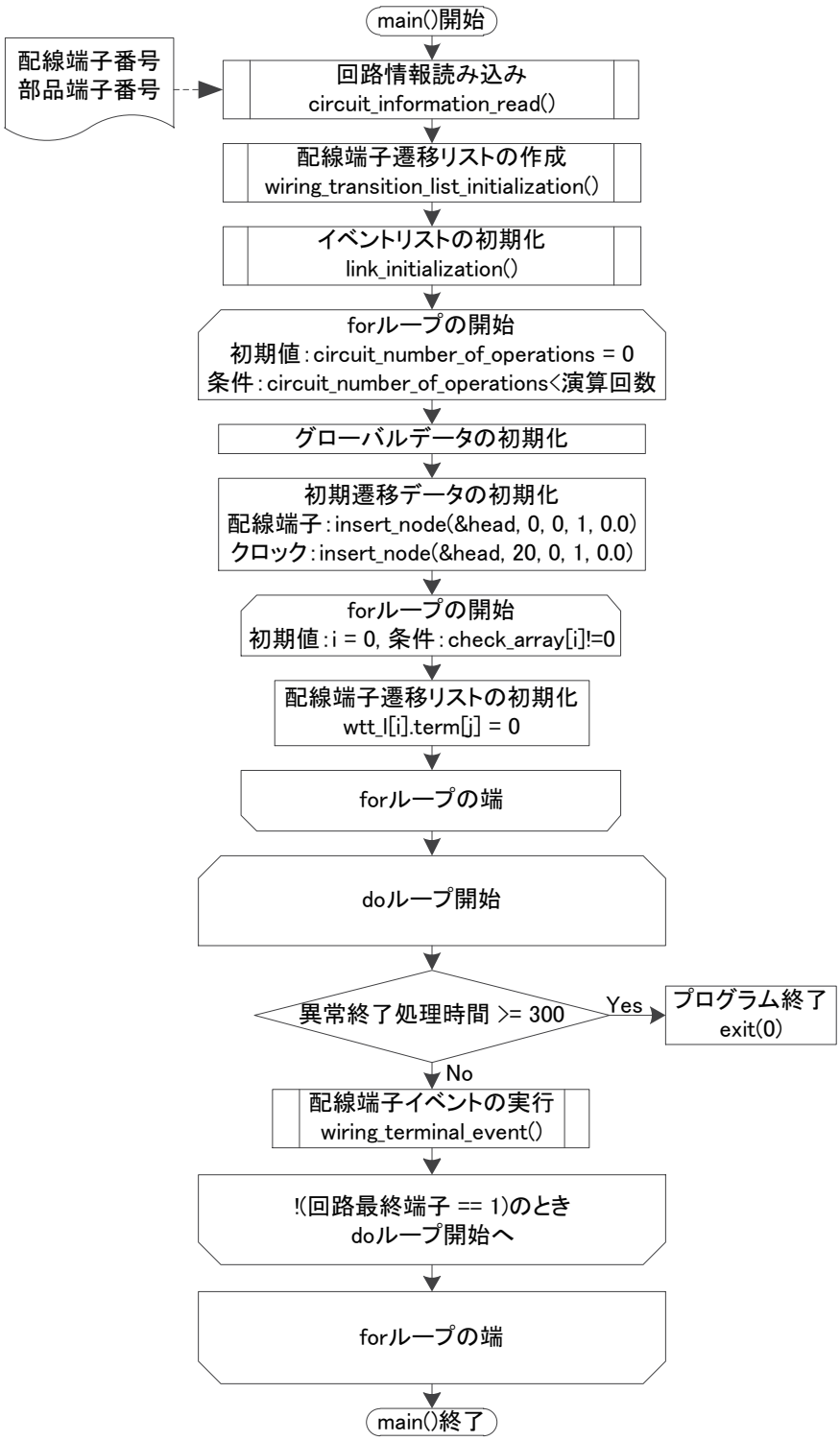
main()開始
配線端子番号
部品端子番号
回路情報読み込み
circuit_information_read()
配線端子遷移リストの作成
wiring_transition_list_initialization()
イベントリストの初期化
link_initialization()
forループの開始
初期値：circuit_number_of_operations = 0
条件：circuit_number_of_operations<演算回数
グローバルデータの初期化
初期遷移データの初期化
配線端子：insert_node(&head, 0, 0, 1, 0.0)
クロック：insert_node(&head, 20, 0, 1, 0.0)
forループの開始
初期値：i = 0, 条件：check_array[i]!=0
配線端子遷移リストの初期化
wtt_l[i].term[j] = 0
forループの端
doループ開始
異常終了処理時間 >= 300
Yes
プログラム終了
exit(0)
No
配線端子イベントの実行
wiring_terminal_event()
!(回路最終端子 == 1)のとき
doループ開始へ
forループの端
main()終了

図 5.2: main() 関数

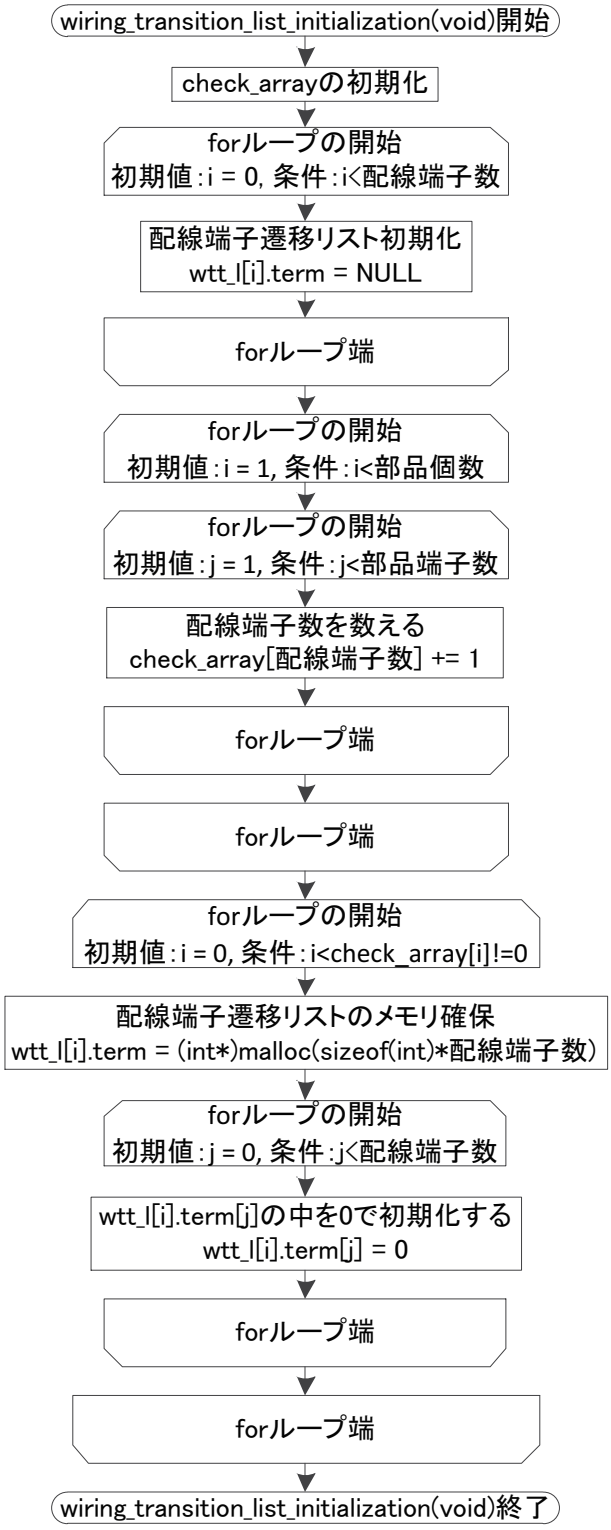

図 5.3: 配線端子遷移リスト関数

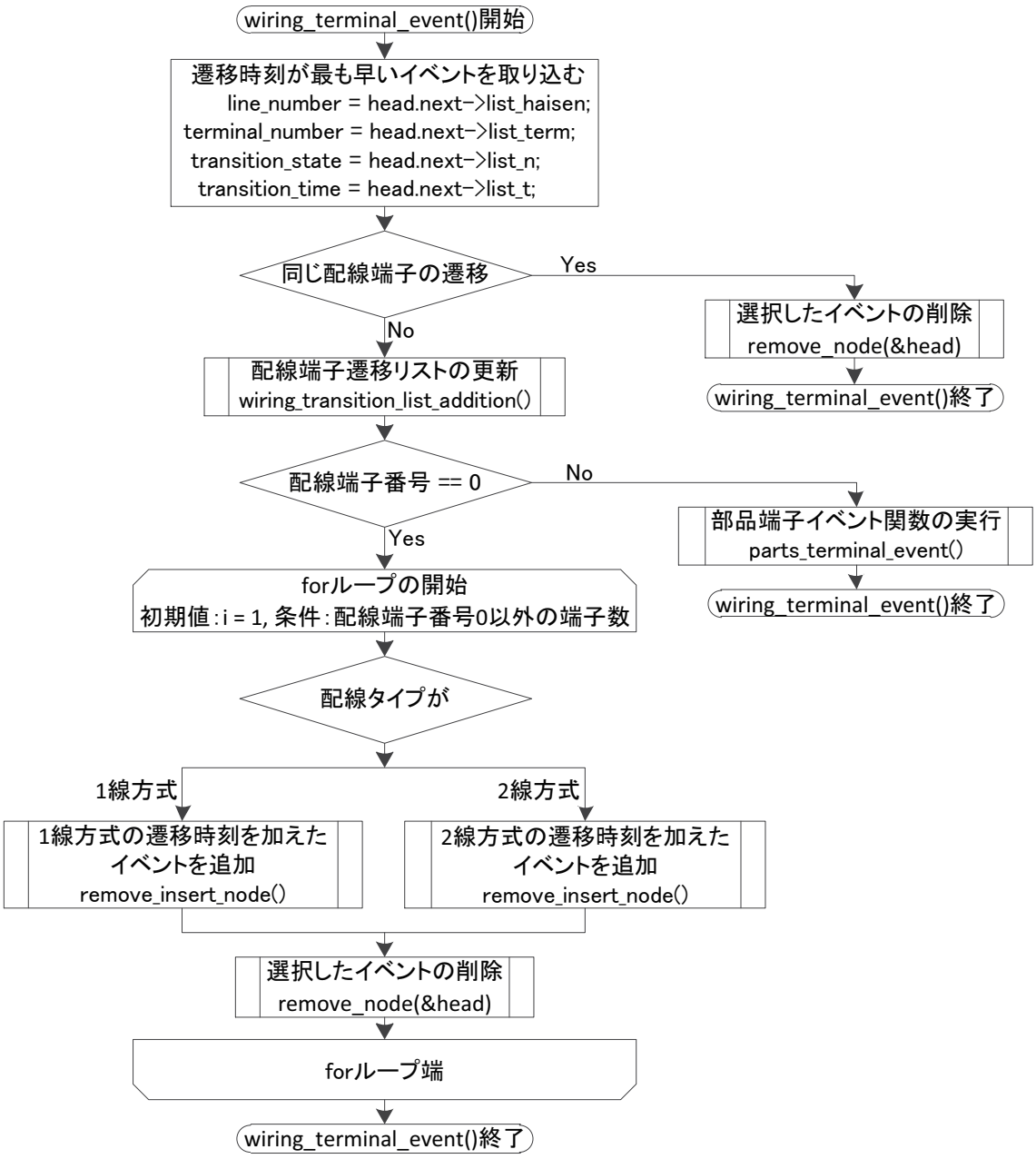

図 5.4: 配線端子イベント関数

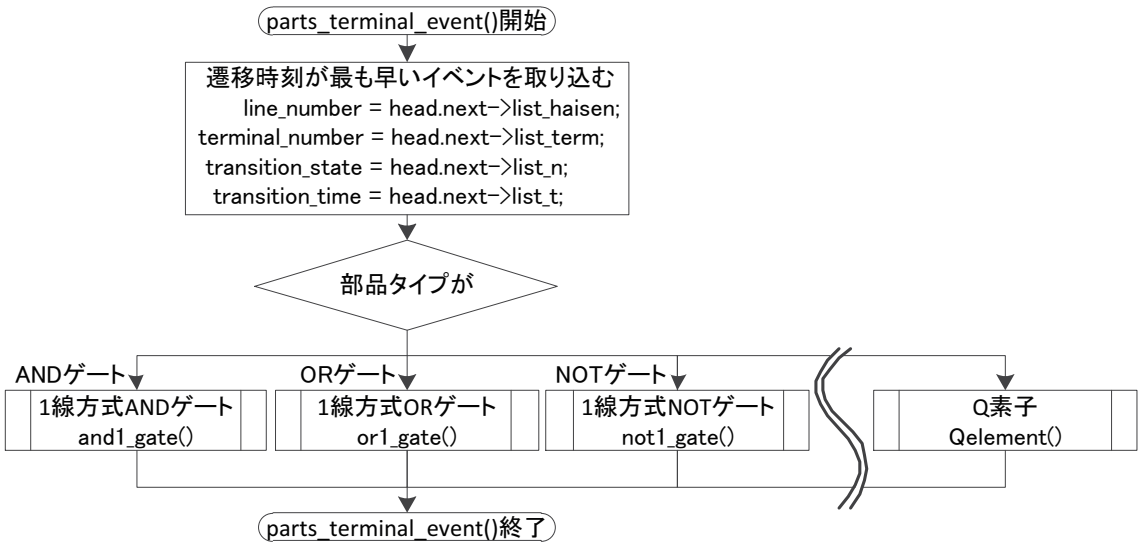

図 5.5: 部品端子イベント関数

ま配線番号及び, 配線端子番号が同じイベントを処理すると, 入力タイミングに対して出力結果が異なる順序で遷移していく可能性がある. そこで, このデータ追い越しが発生し, 配線番号と配線端子番号が同じ場合は, 遷移時刻が早いイベントを残し, 遷移時刻が遅いイベントを削除する. これによって部品への入力タイミングに対して, 正しい値を出力するイベントを追加することが可能となる.

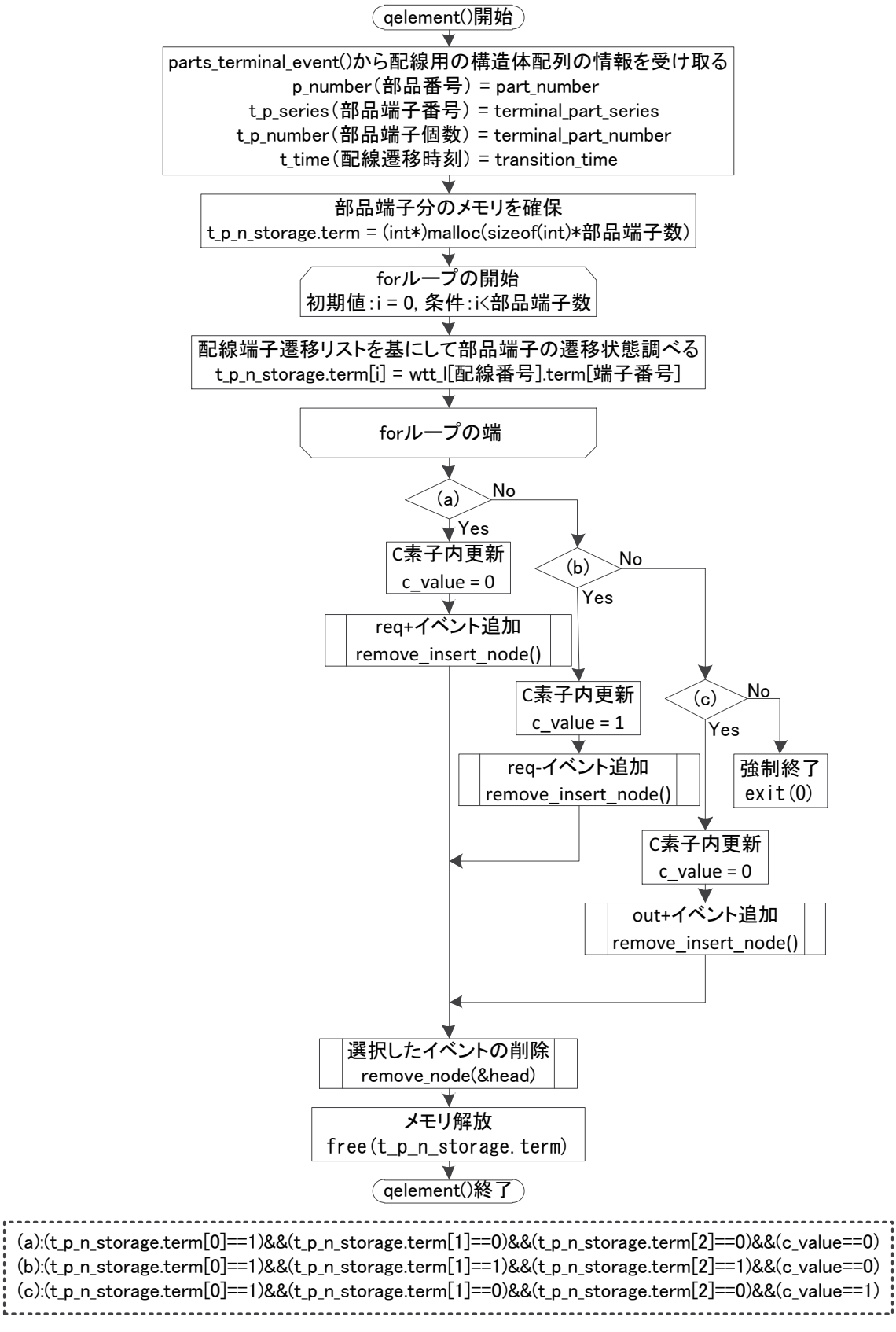
qelement()開始
parts_terminal_event()から配線用の構造体配列の情報を受け取る
p_number（部品番号） = part_number
t_p_series（部品端子番号） = terminal_part_series
t_p_number（部品端子個数） = terminal_part_number
t_time（配線遷移時刻） = transition_time
部品端子分のメモリを確保
t_p_n_storage.term = (int*)malloc(sizeof(int)*部品端子数)
forループの開始
初期値：i = 0, 条件：i<部品端子数
配線端子遷移リストを基にして部品端子の遷移状態調べる
t_p_n_storage.term[i] = wtt_l[配線番号].term[端子番号]
forループの端
(a)
No
Yes
C素子内更新
c_value = 0
(b)
No
Yes
req+イベント追加
remove_insert_node()
C素子内更新
c_value = 1
(c)
No
Yes
req-イベント追加
remove_insert_node()
強制終了
exit(0)
C素子内更新
c_value = 0
out+イベント追加
remove_insert_node()
選択したイベントの削除
remove_node(&head)
メモリ解放
free(t_p_n_storage.term)
qelement()終了
(a):(t_p_n_storage.term[0]==1)&&(t_p_n_storage.term[1]==0)&&(t_p_n_storage.term[2]==0)&&(c_value==0)
(b):(t_p_n_storage.term[0]==1)&&(t_p_n_storage.term[1]==1)&&(t_p_n_storage.term[2]==1)&&(c_value==0)
(c):(t_p_n_storage.term[0]==1)&&(t_p_n_storage.term[1]==0)&&(t_p_n_storage.term[2]==0)&&(c_value==1)

図 5.6: Q 素子の関数

■演算器内信号伝搬遅延によって発生する出力データの誤り

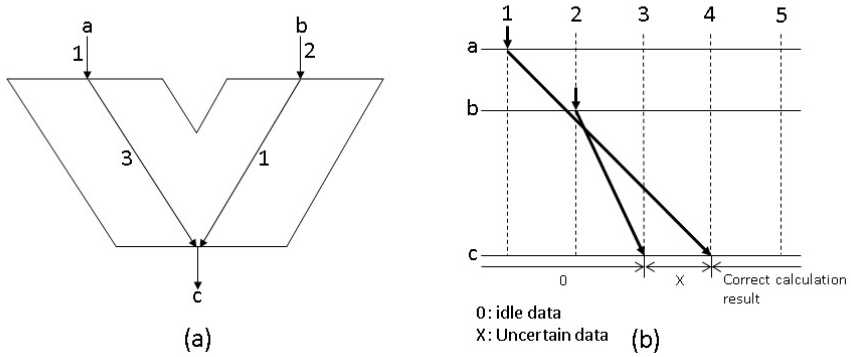

図 5.7: 演算器内の信号伝搬と出力誤り

■出力データ誤りに対する解決策

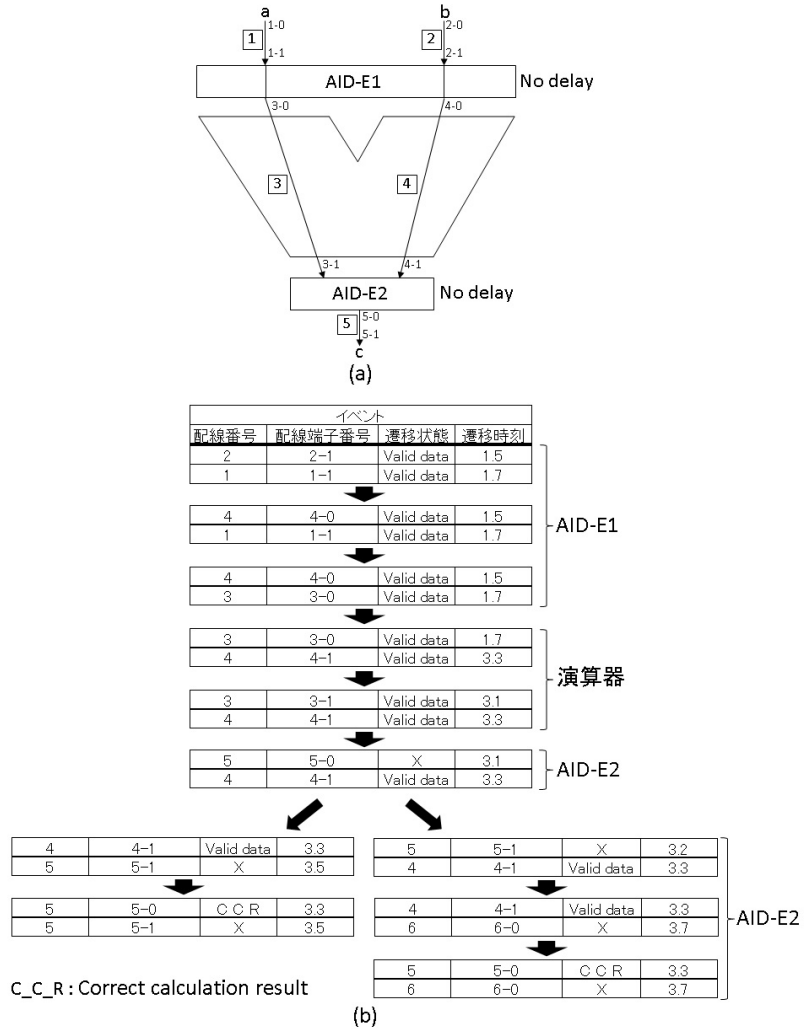

図 5.8: 演算器出力誤りの改善策とイベント

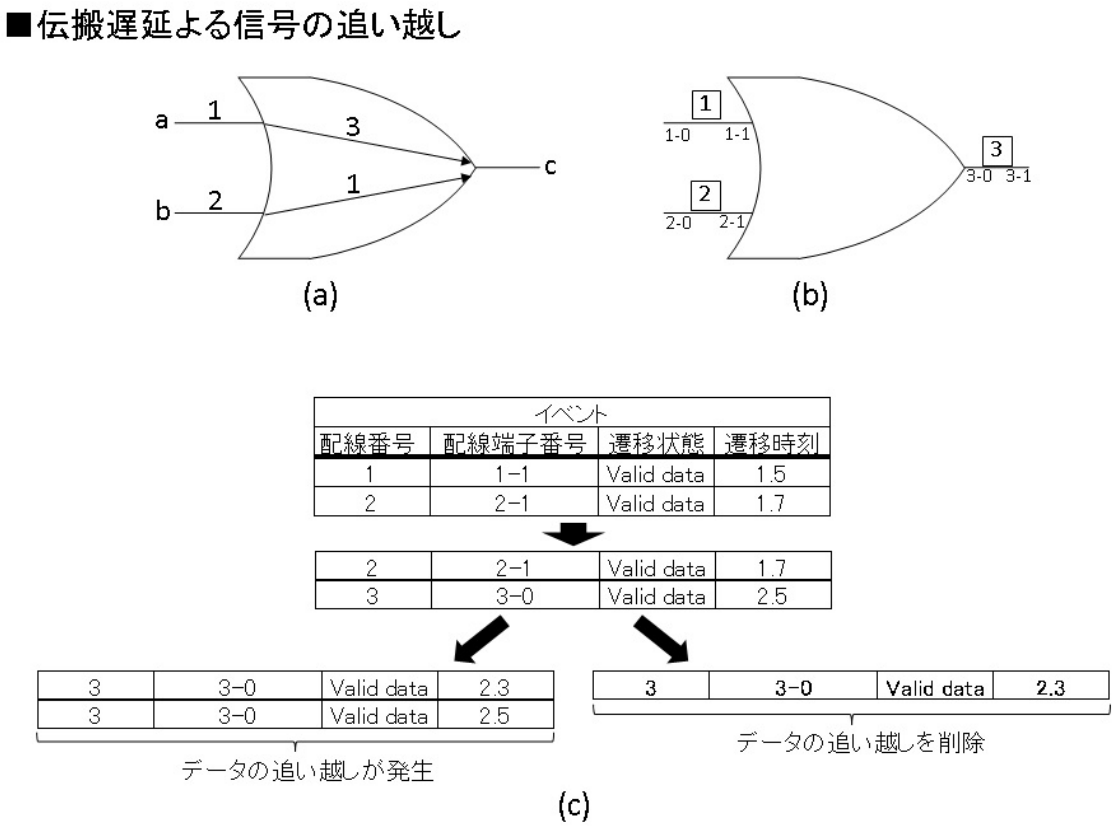

図 5.9: 演算器出力誤りの改善策とイベント

第6章　実験と考察

6.1　評価方法

同期・非同期混合回路の評価方法を以下に示す.

1. 図 3.2 のスケジューリング済みデータフローに対して, 図 6.1(a) のように設計し, 図 6.1(b) の制御器と, これによって制御される同期・非同期混合回路の設計.

2. 同期・非同期混合回路専用のイベントドリブン型シミュレータ（5 章）を使用して, 各配線及び, 部品の遅延量（正規分布に従うランダムな値）を任意に設定することによって,「計算アプリケーション実行の様子の確認」.

3. イベントドリブン型シミュレータ内の各配線及び, 部品の遅延量をばらつかせ, 計算アプリケーションの実行時間を算出するモンテカルロシミュレーションによる「統計的実行時間分布の算出」. このとき使用したクロック周波数 Tc は, 1 万回のモンテカルロシミュレーションで, レジスタへの書き込み誤りが起こらない最小の周期.

4. 1. で設計した同期・非同期混合回路を例に, イベントドリブン型シミュレータ内で同期式回路の演算器と非同期式回路の演算器に対して別々の遅延量を設定し, 1 つの演算器の平均遅延時間設定あたり, 1 万回のモンテカルロシミュレーションを実施し,「同期・非同期混合回路の動作の特徴」と「同期式回路の演算時間と非同期混合回路の演算時間の違いによる演算実行時間変化の様子の確認」.

以下では, 上記の 4. について設定した演算器の平均遅延による実験結果について考察していく.

6.2　実験結果と考察

この実験で使用した演算器の平均遅延とそのばらつき度合い及び, クロック周波数 Tc を図 6.2 に示す. 図 6.2(a) は, 同期演算器の平均遅延を 10 で固定し, 非同期演算器の平均遅延を変動させた場合のクロック周波数 Tc を表したものであり, 図 6.2(b) は, 非同期演算器の平均遅延を 20 で固定し, 同期演算器の平均遅延を変動させた場合のクロック周波数 Tc を表したものである. また, 6.3 は, 同期式回路の演算器の平均遅延を 10 に固定し, 非同期式回路の演算器の平均遅延 10, 20 に対するアプリケーション時間の分布であり, 図 6.4 は, 同期式回路の演算器の平均遅延を 10, 20, 非同期式回路の演算器の平均遅延 20（非同期式回路の演算器は 1 ビットのデータを 2 本の信号線を用いて符号化しているため、一般の同期式回路の演算器に比べて演算実行時間が長いと考え, 初めから非同期回路の演算器の平均遅延を 20 に固定した）に固定した場合のアプリケーション時間の分布である. まず, 同

期・非同期混合回路の特徴として, 図 6.3 の同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=29.0, 分散=平均/3 のグラフのように演算器平均遅延のばらつきが大きい場合, 実行時間の分布の山が複数箇所現れる場合が存在する. これは, 図 6.1(b) のように Q1 の演算が終了し, Q1 からの出力信号 out+が, 同期式回路の制御器に入力された時, 同期式回路の FSM の状態 S0 から S1 への状態遷移が「最短の周期で同期し, フリップフロップの状態遷移が起こった」場合と「1 周期遅れて同期した」場合の実行時間の差であり, それぞれの山の平均値を比較すると, この実行で使用したクロック周波数 Tc=29 分の差があることがわかる. また, 6.3 の同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=2.0, 分散=平均/6 のグラフのように演算器平均遅延のばらつきが小さい場合, は, ある一定の周期で同期して FSM の状 4 態遷移が起こる. この時, 6.3 の「同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=29.0, 分散=平均/3 のグラフ（最短周期で同期したグラフ）」と「同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=2.0, 分散=平均/3 のグラフ」を比べると, それぞれの演算時間の平均が近いことがわかる. これは 6.5 のようにクロックステップ毎の Tc=24 と Tc=29 の場合の同期タイミングを比較すると, Tc=29 の最短周期で同期したタイミングと 1 周期遅れて同期したタイミングの間に, Tc=24 の同期するタイミングが存在する. このため, 同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=24.0, 分散=平均/6 のグラフのクロック周波数 Tc=24 を 29 に変更すれば, 同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=29.0, 分散=平均/3 のグラフの最短周期で同期する時刻あたりで同期し, 演算実行時間も早くなると考えた. その実験結果を図 6.6 に示す. 図 6.6 を見ると, 同期演算器の平均遅延=10, 非同期演算器の平均遅延=10, Tc=24.0, 分散=平均/6 のグラフのクロック周波数 Tc=24 を 29 に変更した方が, 演算実行時間が Tc=29 に比べて早くなっていることがわかる. これにより, 回路構造とばらつきの大きさによっては, Tc の値が大きい方が演算実行時間が短くなる可能性があり, 演算実行時間に対して設定した最小クロック周期 Tc（同期式回路のレジスタへの書き込み対して, タイミング誤りが起こらない最小のクロック周期）が必ずしも最適とは限らないと考えられる. 次に, 図 6.4 の同期演算器の平均遅延=20, 非同期演算器の平均遅延=20, Tc=248.0, 分散=平均/3 のグラフを見ると, 演算器の平均遅延が大きく, 遅延ばらつきが大きい場合は, それに伴ってクロック周波数 Tc の値も大きくなり, その分, 実行時間も大幅に長くなる. また, 図 6.3, 図 6.4 を比べると, 全体的に, 非同期回路の演算器の平均遅延を固定した図 6.4 の方が実行の平均時間が長いことわかる. これにより, 非同期式回路の演算器の平均遅延とクロック周波数との差が小さい場合は, Q 素子の出力信号 out+が同期式回路の FSM に入力され, 同期するまでの時間が短い頻度が高いため, 全体の実行時間が短い. また, 非同期式回路の演算器の平均遅延とクロック周波数との差が大きい場合は, 1 周期遅れて同期する頻度が多くなることに全体の演算時間が長くなると考えられる.

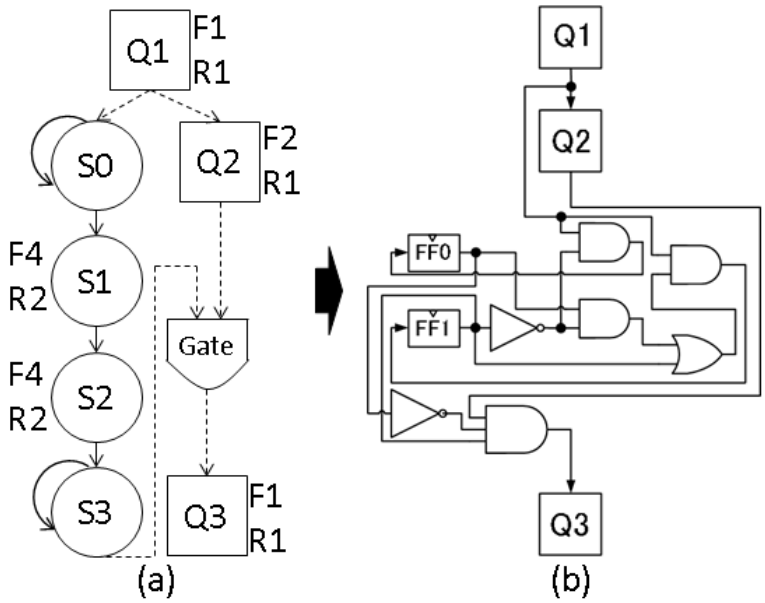

図 6.1: 制御器動作グラフを基にした制御器設計の一例

同期演算器の平均=10(固定)		
分散(σ)	m/6	m/3
上限	1.5m	2.0m
下限	0.5m	0.5m
非同期演算器の平均遅延		
10	Tc=24.0	Tc=29.0
12	Tc=24.0	Tc=29.0
14	Tc=24.0	Tc=29.0
16	Tc=24.0	Tc=29.0
18	Tc=24.0	Tc=29.0
20	Tc=24.0	Tc=29.0

(a)

非同期演算器の平均=20(固定)		
分散(σ)	m/6	m/3
上限	1.5m	2.0m
下限	0.5m	0.5m
同期演算器の平均遅延		
10	Tc=24.0	Tc=29.0
12	Tc=27.0	Tc=33.0
14	Tc=29.0	Tc=36.0
16	Tc=33.0	Tc=40.0
18	Tc=37.0	Tc=44.0
20	Tc=41.0	Tc=48.0

(b)

図 6.2: (a) 同期演算器の平均遅延を 10 で固定し, 非同期演算器の平均遅延を変動させた場合のクロック周波数 Tc (b) 非同期演算器の平均遅延を 20 で固定し, 同期演算器の平均遅延を変動させた場合のクロック周波数 Tc

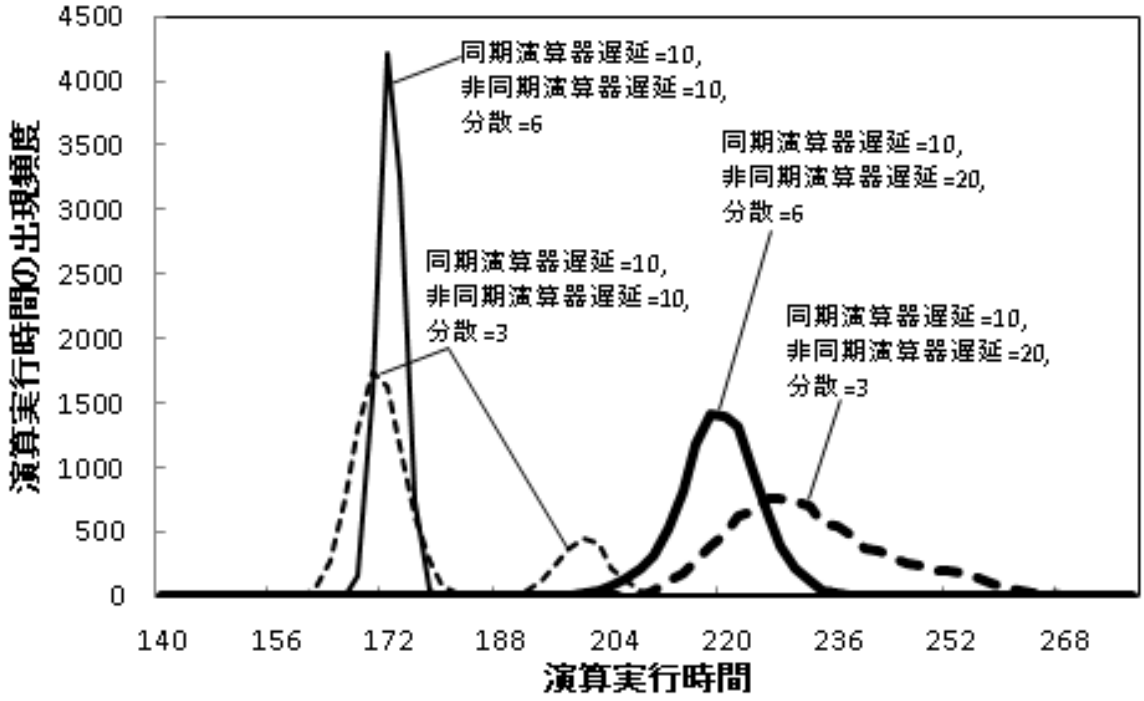

図 6.3: 実線：平均遅延=m, 分散=m/3, 上限=2.0*m, 下限=0.5*m, 同期演算器の平均遅延=10, 非同期演算器平均遅延=(10, 20) 点線：分散=m/6, 上限=1.5*m, 下限=0.5*m, 同期演算器の平均遅延=10, 非同期演算器平均遅延=(10, 20)

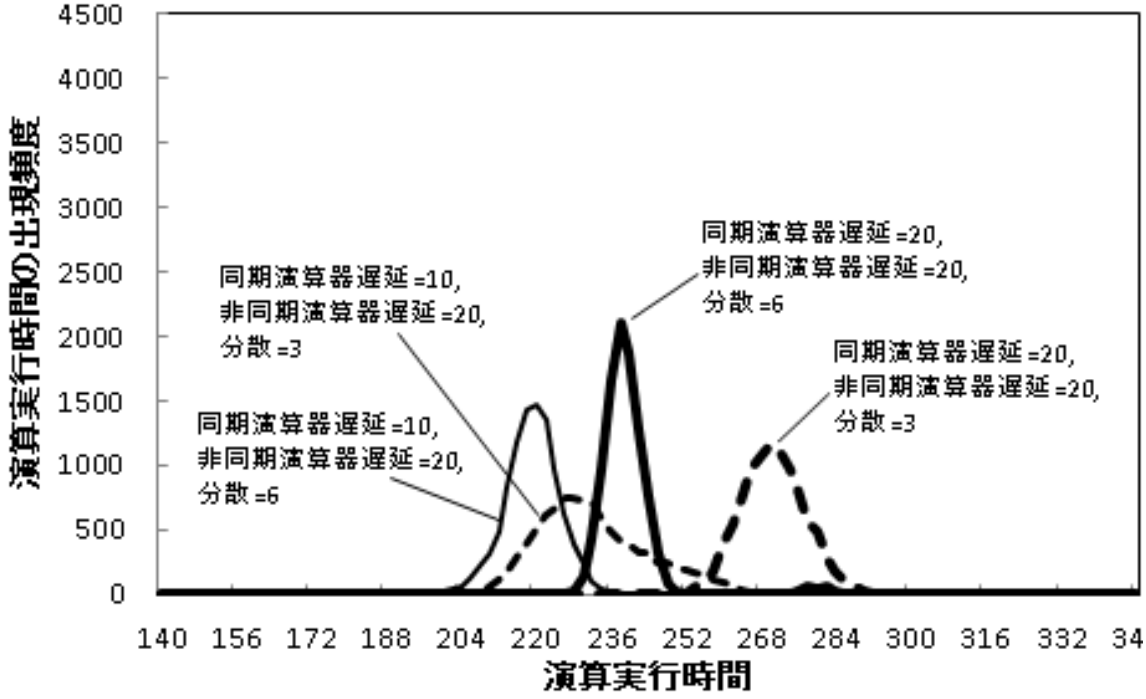

図 6.4: 実線：平均遅延=m, 分散=m/3, 上限=2.0*m, 下限=0.5*m, 同期演算器の平均遅延=(10, 20), 非同期演算器平均遅延=10 点線：分散=m/6, 上限=1.5*m, 下限=0.5*m, 同期演算器の平均遅延=(10, 20), 非同期演算器平均遅延=20

図 6.5: クロック周波数 Tc=24 と Tc=29 に設定した際の各クロックステップ毎の同期タイミング

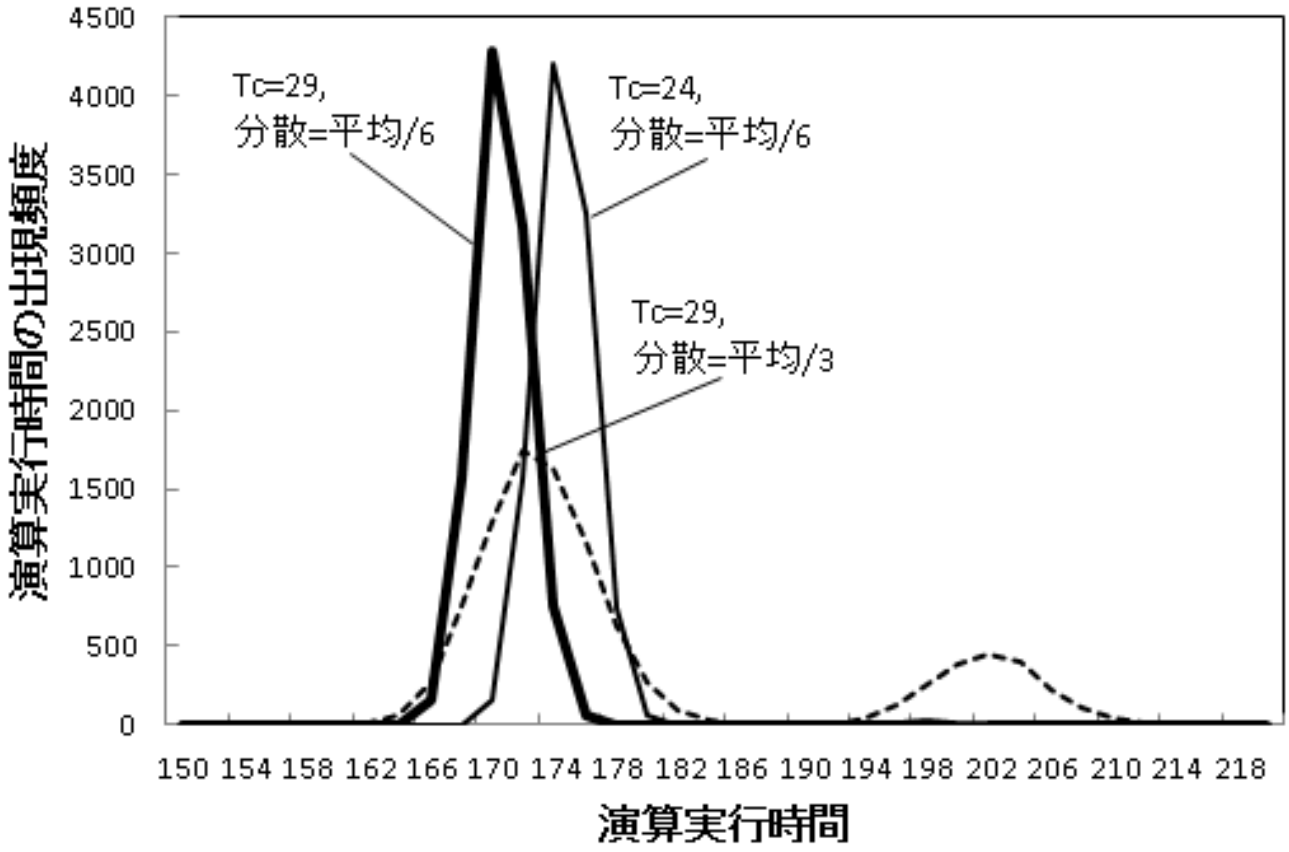

図 6.6: クロック周波数 Tc=24 を Tc=29 に変更した際の演算実行時間の違い

第7章　まとめと今後の課題

7.1　まとめ

本研究では, 同期式回路と非同期式回路の利点を活かすことを目的に, ある計算アルゴリズムに対して, 同期演算として実行する部分と非同期式回路として演算を行う部分に切り分け, 1 つの回路ブロック内に混合させるための設計手法の提案を行った. 提案した設計では, 同期回路部がクロック信号によって状態遷移する有限状態機械 (FSM) にて制御され, 非同期回路部が Q 素子のネットワークによって制御されることを基本とし, 両者が連携して正しく計算を実行するためのレジスタ転送レベルの制御構造を「制御器動作グラフ」と名付けたグラフ構造にて表現した. これにより, 同期・非同期間で共用されるデータに付随する制御や, 資源割り当てに付随する制御を表現することができ, 同期・非同期回路の混合を可能にしている. 同期・非同期間でのデータのやり取りについて, 同期式回路の演算結果を非同期式演算で使用する場合と, 非同期式回路の演算結果を同期式演算で使用する場合が存在する. そこで, 同期式回路での 1 線式データ（1 本の信号線が 1 ビットの情報を持つ）及び, 非同期回路での 2 線式データ（2 本の信号線にて 1 ビットの情報を持つ）のデータ入力と, クロック信号及びハンドシェイク信号の両方で制御可能なマスタースレイブ型レジスタを開発し, 同期・非同期間でのデータ共有やレジスタ共有を可能にした. この一方, 同期演算と非同期演算が重複する演算経路を使用する場合, 非同期式回路の演算で使用するデータ伝搬経路を初期化することが現状では非常に困難であることから本研究では, 同期・非同期回路間で独立した演算経路を使用することとした. また, 同期・非同期混合回路の合成では, 実装すべき計算アルゴリズムを入力とし, (1) 計算アルゴリズム中の演算の同期演算, 非同期演算への切り分け, (2) 同期演算のスケジュール, (3) 制御器動作グラフの作成, (4) 資源割り当て（資源共有）の決定と, それに付随して必要となる制御の追加を経て最終的な制御器動作グラフを完成させる. この後, 資源割り当てに基づいて同期・非同期データパスの構造を定め, 制御器動作グラフに基づいて FSM, Q 素子ネットワーク構造及び, 両者間の制御信号線構造が定まって回路合成が完了する.

混合方式や回路設計手法の検討と並行して, 同期・非同期混合回路の動作確認と評価のために, イベントドリブン型のマクロタイミングシミュレータを開発した. これにより, 回路中の各部に設定した遅延量の下で, 計算アプリケーションの実行の様子の確認と, 実行に要する時間の算出や, モンテカルロシミュレーションによる統計的実行時間分布の算出を可能としている. 回路設計と動作検証の実験では, 同期演算と非同期演算とが別々の平均遅延時間を持つものとし, 1 つの平均遅延時間設定あたり 1 万回の試行を行うモンテカルロシミュレーションを実施して, 計算アプリケーションの実行時間の分布を求めた. これにより, レジスタ転送レベルにおける同期・非同期混合回路の動作の特徴と, 同期演算と非同期演算の実行時間の違いによる計算アプリケーション実行時間の変化の様子を確認

することができた.

7.2 今後の課題

本研究では, 同期・非同期混合のデータフローの基となる計算アルゴリズムにおいて, 同期・非同期演算の切り分けの最適化を議論せずに, 制御器動作グラフの設計を行った. そのため, 設計した同期・非同期混合回路においても, 同期・非同期回路間で使用する部品の個数や種類, 配線長などの最適化は行っていない. 今後は, 今回の演算器遅延を変動させた場合の同期・非同期回路の有効性を踏まえ, 与えられた計算アルゴリズムに対して, 同期式回路として演算を行う部分と, 非同期式回路として演算を行う部分のより明確な基準を議論していき, 同期・非同期混合回路にたいして, 高位合成の枠組みを利用した設計最適化手法の開発も必要であると考える.

また, 同期・非同期制御器間の接続構造が, 回路全体のオーバーヘッドになっている可能性あることから, 今後, より効率のよい制御構造を考案していく必要があると考える.

第8章　謝辞

本研究を進めるにあたり, 終始熱心にご指導を頂いた卒業論文指導教員の金子峰雄教授に感謝致します. また, 本研究を進めるにあたり, 多くのご助言をいただいた北陸先端科学技術大学院大学 情報科学研究科 助教 張任遠氏に感謝致します.

参考文献

[1] 南谷　崇：非同期式マイクロプロセッサの動向, 情報処理, Vol.39, No3, pp.181-186(1998).

[2] 齋藤　寛：非同期式回路の設計技術 Techniques for asynchronous Circuit Design, Fundamentals Review Vol.3 No.3

[3] 唐木　信雄：非同期回路設計のすすめ, Design Wave Magazine, vol.10, no.7 pp.64-69, jul.2005

[4] J.Spars,S.Furber：Principles of Asynchronous Circuit Design, Kluwer academic publishers,2002.

[5] Martin, A.J.：The Limitations to Delay-Insensitivity in Asynchronous Circuits, Advanced Research in VLSI (Proc. 6th MIT Conf.), pp. 263-278 (1990)

[6] 唐澤　陽平, 桑子　雅史, 新塚　稔央,　横山　孝典：Clock Gating に基づいた GALS 型システムにおける回路性能を考慮したインターフェースの構成法,　電子情報通信学会論文誌 D Vol. J94-D No.1 pp.324-333 (c) (社)　電子情報通信学会 2011

[7] Koji OHASHI, Mineo KANEKO： Dual-Rail Two-Phase Asynchronous Datapath Synthesis Based on Aggressive Register Sharing Model. School of Information Science，Japan Advanced Institute of Science and Technology 1-1 Asahidai，Nomi，Ishikawa 923-1292 Japan.