卒　業　論　文

題　　目

動作中に構成が変化する回路の利用法に関する検討

指導教官

小栗 清 教授

平成17年度

長崎大学工学部

情報システム工学科

本田 将之（602436）

論文要旨

今日のコンピュータはそのほとんどがフォンノイマンアーキテクチャを採用している．フォンノイマンアーキテクチャは，ハードウェアとソフトウェアから構成され，ハードウェアは CPU とメモリから構成されている．ソフトウェアを取り替えれば任意の機能を実現でき，この汎用性故に発展を続けてきた．しかし，フォンノイマンアーキテクチャはプログラム論理であるため，ハードウェアの本質である並列性，高速性を活かしきれていない．

PCA(Plastic Cell Architecture) は FPGA よりもさらに汎用性を向上させ，布線論理の世界で CPU とメモリによるコンピュータに匹敵する汎用性を持たせることを目的として開発されたアーキテクチャである．PCA の主な特徴として，非同期回路であることと動的再構成機能を持つことが挙げられる．

本研究室で研究しているビットシリアル PCA はビットシリアル処理に特化した PCA で，ビットパラレル方式で起こるスキューの問題などを解決できると期待して研究を進めている．

そこで，本研究では，PCA の持つ動作中に回路の構成を変化させるという特性を活かした回路の設計手法について提案し，動的要素を追加・削除できる設計ツール tanoqs を用いて実装・シミュレーションを行い，再帰表現を用いることで動作中に回路の構成が変化する回路の設計が行えることを明らかにした．今後，この再帰表現を用いたアプリケーションの幅広い適用について検討する．

目 次

図目次

表目次

第1章

緒 論

現在のコンピュータは，フォンノイマンアーキテクチャ[1,2] が主流となっている．フォンノイマンアーキテクチャは，ハードウェアとソフトウェアから成っており，ハードウェアは CPU とメモリから構成されている．CPU は設計時にその機能が全て決定されるため決まった処理しかできないが，CPU 自身がソフトウェアによってメモリを書き換えることにより，様々な動作を行うことを可能にしている．よって，フォンノイマンアーキテクチャは，メモリという可変部，CPU という組込部から成っているといえる．可変部とは動作中に書き換えることができる部分のことであり，組込部とは最初から決められた処理しか行うことができない部分のことである．この可変部と組込部を持つことが，フォンノイマンアーキテクチャの高い汎用性の源となっている．しかし，フォンノイマンアーキテクチャはプログラム内蔵方式であり，プログラムを逐次的に処理するため処理速度は決して速くないなど，問題が全くないわけではない．

近年は，FPGA(Field Programmable Gate Array)[3] という書き換え可能な LSI が登場し，私たちの身近なところ [4] で使われるなど，徐々にニーズを増やしている．FPGA は，ユーザが何度でもディジタル論理回路の構成をプログラムすることによって変更できるため，ハードウェアの教育用として用いられることも多い．FPGA を用いれば，回路構成の変更は可能になるが，動作中の回路構成の変更は難しい．FPGA で動作中に回路の構成を変更する場合は，ソフトウェアにおけるオーバレイのように，大きな回路図の一部を FPGA にマッピングし，処理が終了したら切り替えて使用する，といった形で行われる．

本研究室では，FPGA よりさらに汎用性の高く，動作中に回路構成を変更できる動的再構成機能を持つ PCA(Plastic Cell Architecture)[9–11] というアーキテクチャについて研究している．PCA はデータフローアーキテクチャであり，フォンノイマンアーキテクチャでメモリが担当している可変部をハードウェアの中に取り込むことによって汎用性を高めている．

本論文では，PCA の動作中に回路の構成が変化するという特徴をどのようにして表現するかという点に加え，この特徴を活かすことの出来るアプリケーションについての検討を行う．

本論文の構成は次の通りである．まず，続く 2 章では本研究の背景について述べるとともに，従来の関連研究についてのサーベイを行う．次に 3 章では従来の研究の問題点を指摘し本研究の目的と意義を明らかにする．その後，4 章で動作中に構成が変化する回路の回路図の表現方法について述べる．次に，5 章にて 4 章にて述べた表現方法に基づいて実際にアプリケーションの設計・実装を行う．最後に 6 章で本研究の結論を述べる．

第2章

背 景

本章では本研究の背景となる非同期回路技術，PCA(Plastic Cell Architecture)，ペトリネット，高速フーリエ変換 (FFT) について述べた上で，関連研究についてのサーベイを行う．

2.1 非同期回路技術

論理回路には，同期回路と非同期回路の二種類がある．同期回路は，周期的な時間信号であるグローバルクロックと呼ばれる信号によって動作する．これに対し，非同期回路はクロックを持たない．非同期回路の設計には，一般に多くの経験と実験が必要となるので，現在のアーキテクチャはほとんど同期回路を採用している．しかし，半導体技術の発展に伴って回路遅延に対する配線遅延の割合が大きくなってきており，クロックを用いるためには様々な問題をクリアする必要がある．こういった背景から，クロックを必要としない非同期回路への注目が高まっている．

2.1.1 非同期回路の特徴

同期回路はクロックを用いるため，クロック周期を決めるには，クロックスキューを考慮する必要がある．クロックスキューとは，クロックの伝播遅延，回路の配線遅延などによって起こる，クロックの到着時間のタイミングのずれのことである．クロックスキューが全くないのが理想だが，実際には回路規模が大きくなればなるほどクロックスキューは大きくなる．この問題が，現状のコンピュータにおいて，クロック周波数の上昇が頭打ちになっていることの原因となっている．対して，非同期回路はクロックを持たないため，このような問題は起こらない．非同期回路では，要素間はクロック信号の代わりにハンドシェイクプロトコル [5] と呼ばれるローカルな繋がりを持つことによって処理を行う．非同期回路ではクロックスキューが起こらないため，回路の大規模化・高集積化が可能となる．

また，同期回路ではパイプライン処理を行うとき，最も最大遅延の大きいブロックに合わせてクロック周波数は決定されるため，他のブロックの処理が速く終わっても全体の動作速度は変化しない．しかし，非同期回路では回路の各ブロックがそれぞれの最大動作速度で動作し，最大遅延の大きいブロックの処理であっても遅延の少ない処理を行う場合もあるため，同期回路に対して総合的に見た平均動作速度の向上が見込める．

同期回路では回路の全ての部分にクロックが分配される．そのため，動作する必要がない回路においても電力が消費されてしまう．これに対し，非同期回路では動作する部分でしか電力は消費しない．よって，同期回路と比べ，回路全体の消費電力を抑えることができる．

上記以外にも，電磁波の放出が少ないことや，回路の追加・変更など設計の自由度が高いことなどが非同期回路の利点として挙げられる．

2.1.2 ハンドシェイクプロトコル

非同期回路では，データのやり取りにクロック信号の代わりにハンドシェイクプロトコル [5] が用いられることは既に述べたが，ここではその詳しい方式について述べる．非同期回路で用いられるハンドシェイクプロトコルには，4 相プロトコル (RZ:return to zero, 図 2.1(a)) と 2 相プロトコル (NRZ:non-return to zero, 図 2.1(b)) がある．

4 相プロトコルでは，まず送信側がデータを送ると同時にリクエスト信号をアサート (アクティブに) し，受信側はデータを受信し終わるとアクノリッジ信号をアサートする．すると送信側はリクエスト信号をネゲート (インアクティブに) し，その信号遷移を検出した受信側がアクノリッジ信号をネゲートする．これで一回のデータ転送が終了する．

それに対して，2 相プロトコルでは，信号の遷移自体に意味を持ち，送信側がリクエスト信号を反転させ，受信側はデータを受信し終わるとアクノリッジ信号を反転させる．これで一回のデータ転送は終了である．よって，2 相プロトコルの一回のデータ転送にかかる時間は 4 相プロトコルよりも短くなる．しかし，一回目のデータ転送と二回目のデータ転送とでは信号の HIGH, LOW が入れ替わり，これを正しく判定する必要があるため回路規模が大きくなり，消費電力も増大する．

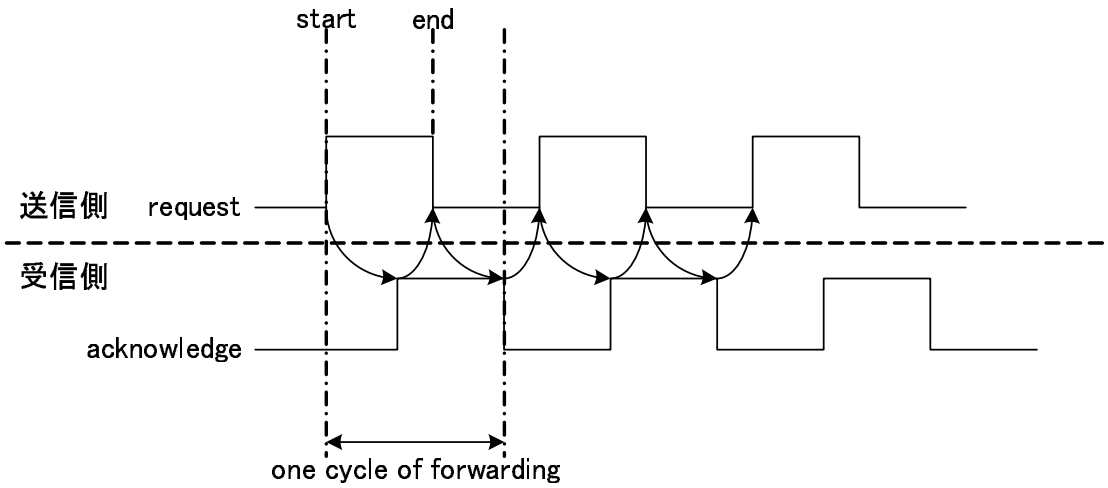
2.2 節で述べる PCA では，構造の簡単な 4 相プロトコルを採用している．

2.1.3 非同期パイプライン設計と Muller の C 素子

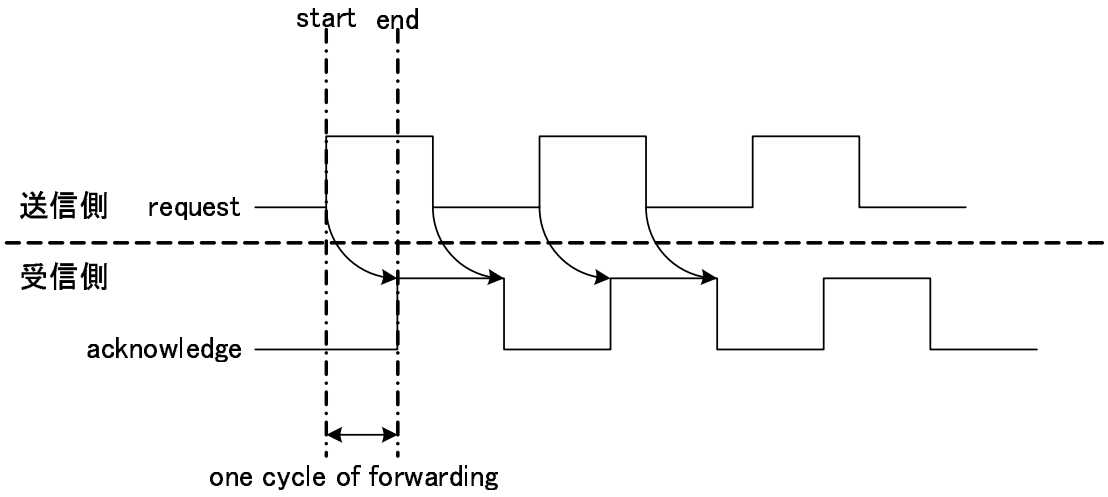
空間的に広がった回路で処理を複数の部分で分担して行うときの効率の良い方法としてパイプラインがある．これは論理回路で処理を行わせる場合の基本的な考え方となっている．同期設計では，パイプライン動作は止まることのできるものとそうでないものに分けられる．前者を 2τ パイプライン，後者を 1τ パイプラインと呼ぶ．

止まることができるようにするためには，後段の処理部が空きになったことを確認してから処理を送る必要があるため，各処理部は最大でも 2 クロックサイクルに 1 度しか動作できない．このことから 2τ パイプラインと呼ばれる．一方，止まることのできないパイプラインは後段の処理が終了することを前提として処理を送るため，各処理部は毎クロック動作することができる．このことから 1τ パイプラインと呼ばれる．

1τ パイプラインは同期回路でのみ考えることが出来るパイプラインであるが，2τ パイプラインの考え方は非同期回路でのパイプラインにも適用させることが出来る．



(a) 4 相プロトコル



(b) 2 相プロトコル

図 2.1 ハンドシェイクプロトコル

Muller の C 素子

Muller の C 素子 [6] は非同期回路の設計を行うのになくてはならない基本素子である．図 2.2 に Muller の C 素子のシンボル図を，表 2.1 に Muller の C 素子の真理値表を示す．

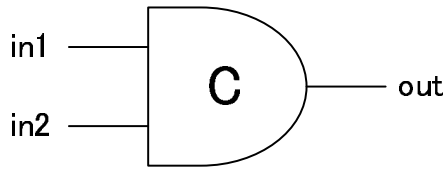


図 2.2 Muller の C 素子

表 2.1 C 素子の真理値表

in1	in2	out
0	0	0
0	1	値の保持
1	0	値の保持
1	1	1

C 素子は，表 2.1 に示したように，入力がともに 0 になったときに出力を 0 に，入力がともに 1 になったときに出力を 1 にする回路である．内部に記憶素子を持っており，2 つの入力が異なっている間は直前の値を保持し出力を変化させない．出力が反転することを C 素子が反応すると呼ぶことにすると，反応させることの出来る入力値というものを考えることが出来る．反応させることの出来る入力値は C 素子を反応させてしまうともはや反応させることの出来る入力値ではなくなる．例えば，C 素子の出力が 0 であるならば，反応させることのできる入力値は 1, 1 となる．両方の入力が 1 となると C 素子が反応して出力が 1 となるが，反応した瞬間にこの C 素子を反応させることが出来る入力値は 0, 0 へと変わる．図 2.3 に C 素子を NAND ゲートと OR ゲートを用いて実現した例を示す．また，この C 素子を用いると 2τ パイプラインが簡単に表現できる．図 2.4 に C 素子を用いて実現した非同期パイプラインを示す．

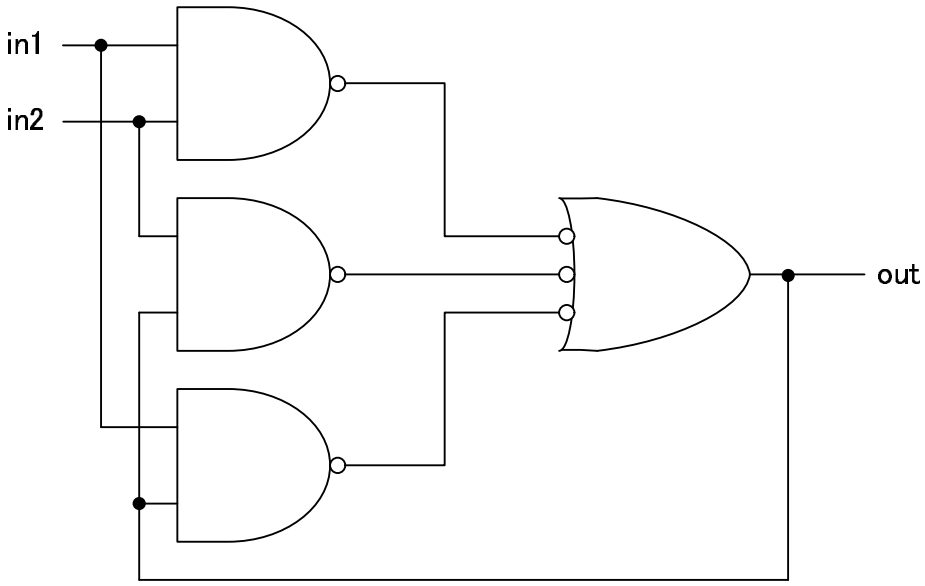


図 2.3 NAND と OR を用いた C 素子の実現例

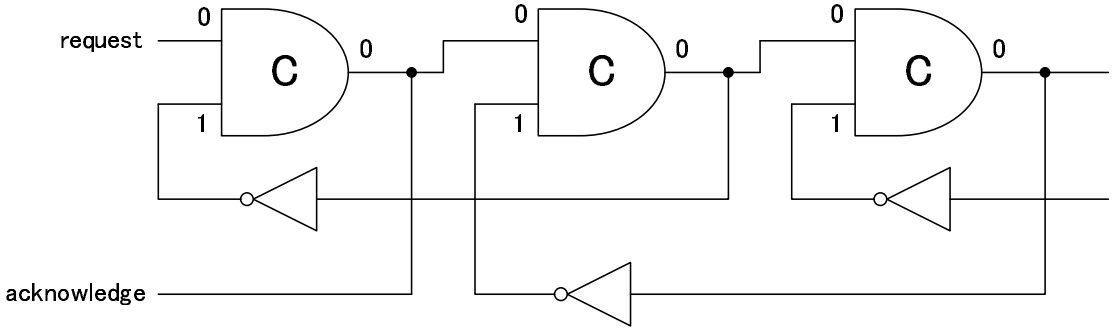


図 2.4 C 素子を用いた非同期パイプライン

2.1.4 遅延モデル

論理回路を設計する際には，素子および配線の遅延モデル [7] を考慮する必要がある．同期回路の場合，設計時に定められたそれらの遅延は未知であるものの，その上限値は既知であると仮定している．

一方，非同期回路の場合，素子および配線の遅延は可変であり，その上限は未知であると仮定する．非同期回路の遅延モデルには以下のような種類のものがある．

- DI(Delay Insensitive) モデル

  DI モデルは,「遅延の大きさは有限であるが上限は未知である」と仮定したモデルである．

- QDI(Quasi Delay Insensitive) モデル

  QDI モデルは，DI モデルを修正したものであり，配線の分岐先の信号遷移はほぼ同時に起こると仮定したモデルである．

- SDI(Scalable Delay Insensitive) モデル

  SDI モデルは，DI モデルや QDI モデルよりも楽観的なモデルであり,「回路要素の絶対的な遅延変動の大きさに上限はないが，相対的な遅延変動の幅には上限がある」と仮定したモデルである．

DI モデルや QDI モデルでは，実際には起こりそうもない遅延変動に対しても正しい動作を保証するので，信頼性は高いものの十分な速度性能を得ることが難しい．対して，SDI モデルでは，動作の信頼性を確保しつつ，速度性能を向上させることができる．

このような理由から，2.2 節で述べる PCA は SDI モデルを採用している．

2.1.5 束データ方式

同期回路では，外部から入力されるクロック信号をトリガーとして順序回路へデータを書き込むが，非同期回路はクロック信号を用いないため，他の方法で演算の完了を検知し，

順序回路への書き込みを行う必要がある．一般に，演算を行うデータパスの遅延は，実行される命令・データによって異なり，ビット毎にもばらつきがある．そのため，非同期回路では，大きく分けて以下の 2 種類の設計方式が存在する．

- 2 線 2 相式
- 束データ方式

この 2 種類の方式について以下に説明する．

2 線 2 相式

2 線 2 相式 (図 2.5(a)) は，1 ビットのデータを肯定，否定の 2 本の信号線対を用いて表現する方式である．(1, 0) で論理値 1，(0, 1) で論理値 0，(0, 0) で 1, 0 どちらでもないスペーサを表す．(1, 1) は不正値とする．実行値 ((1, 0) と (0, 1)) とスペーサ (0, 0) を交互に送ることでデータ転送を行う．

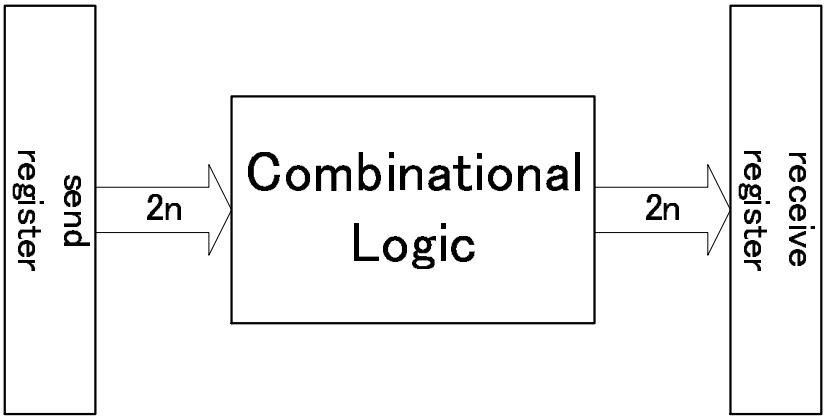
2 線 2 相式は有効データの到着そのものが書き込みのトリガーとなるため，遅延変動に対する信頼性が高い．しかし，この方式では，N ビットのデータを表現するのに 2N ビットの信号線を必要とするため回路量が増大する．また，有効データの間に無効データを挟む必要があるため，処理におけるオーバーヘッドが大きい．

束データ方式

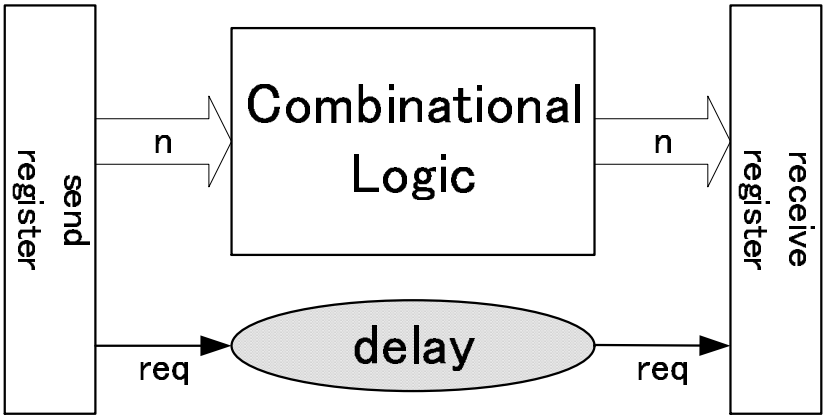
束データ方式 (図 2.5(b)) は，同期回路と同様に 1 線式データパスを用い，順序回路へのタイミング信号のみハンドシェイクにより生成する，同期式と 2 線 2 相式の中間に位置する設計方式である．

束データ方式はクロックの代わりとなるストローブ信号を 1 線式データパスに付加した構造になっている．ストローブ信号は遅延素子を用いて生成され，その大きさはデータ処理遅延の最悪値に基づいたものとなるため，束データ方式では回路単体の速度性能は最悪遅延により制限される．

2.2 節で述べる PCA では，束データ方式を採用している．これは制御部の回路と配線が複雑になるのを避けるためである．



(a) 2 線 2 相式



(b) 束データ方式

図 2.5 2 線 2 相式と束データ方式

2.2 PCA(Plastic Cell Architecture)

PCA(Plastic Cell Architecture)[12] は，FPGA よりもさらに汎用性を向上させ，布線論理の世界で CPU とメモリによるコンピュータに匹敵する汎用性を持たせることを目的として開発されたアーキテクチャである．

PCA の主な特徴として，非同期式であることと，動的再構成機能を持つことが挙げられる．

NTT，長崎大学，京都大学，会津大学などで研究が進められており，具体的な実現例としては，NTT で開発された PCA-1[13, 14]，PCA-2[15]，京都大学で開発された PCA-Chip2[16]，長崎大学で開発中のビットシリアル PCA などがある．

2.2.1 計算機の汎用性について

ここでは，計算機の汎用性について述べる．

コンピュータが発展してきた理由は汎用性にある．コンピュータはその汎用性故に様々な分野で使用・応用され，今ではコンピュータなしでは我々の生活は成り立たない．

今日のコンピュータはそのほとんどがフォンノイマンアーキテクチャを採用している．フォンノイマンアーキテクチャは，ハードウェアとソフトウェアから構成され，ハードウェアは CPU とメモリから構成されている．ソフトウェアを取り替えることでメモリを書き換えることができ，これが汎用性の源となっている．この汎用性によって，機能を階層的に，たとえば，OS，ミドルウェア，アプリケーションプログラムというように構成することが可能になり，汎用であるが故に多くの人々の努力を集約できた．その結果，コンピュータは発展を続けることができた．

決定を後回しにすること

汎用であることの重要性はプログラミング言語の発展にも見てとることができ，それは決定を後回しに出来る能力がどのように変化したかに沿って説明することが出来る．高級言語において，この後天的決定性をどの程度満たしているかを図 2.6 に示す．

まず，プログラム内蔵方式が登場する以前は，機能を変更できない段階である．次はプログラムを変更すれば機能を変更できる段階である．この段階を象徴するものとしてプログラミング言語の FORTRAN がある．初期の頃の FORTRAN では，メモリは静的にしか確保できず，配列の大きさなどをあらかじめ見積もっておく必要があり，実際に動作させその見積もりが間違っていることが分かると，プログラムの変更を行わなければならなかった．その次の段階では，C 言語を例に挙げると，データを必要になった時点で割り付け (malloc)，不要になったら回収する (free) という機能が考案され追加された (メモリのヒープ管理)．この結果，データについてはあらかじめ必要量を予測するということが不要となった．しかしながら，機能の変更については，C 言語では広範囲の関数を見直す必要が生じるため，機能については前もって十分な検討をして上で決定しておく必要があった．そこで次の段階のオブジェクト指向言語 (C++, JAVA など) では，機能の変更や追加を行う機能 (クラスと継承) が考案され追加された．すなわち，機能の決定を遅らせることが可能になった．最近では，さらに次の段階，すなわち設計時に考慮されていなかった

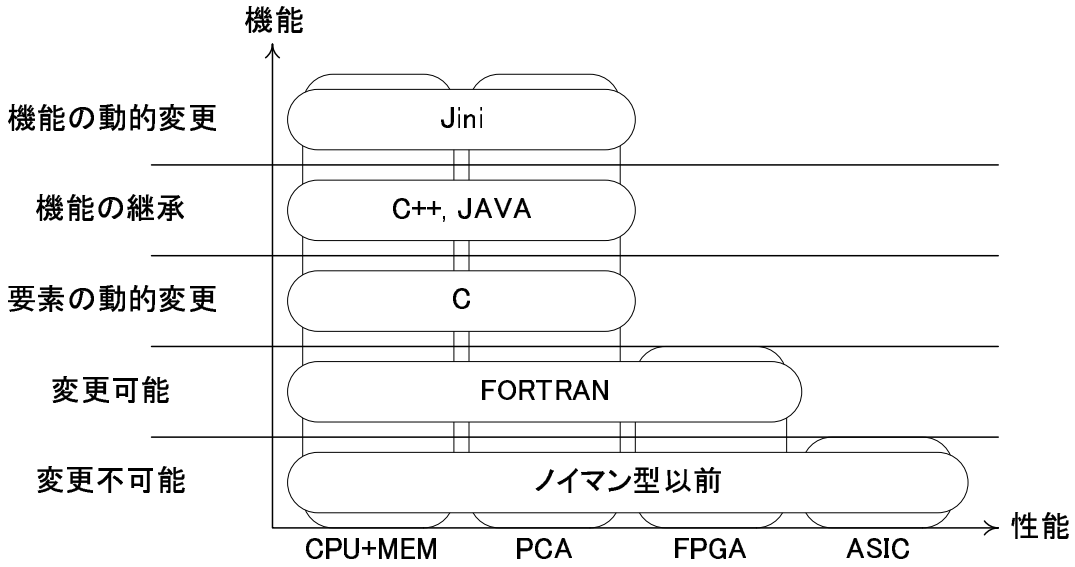


図 2.6 決定を後回しにできる能力

機能を動作時に追加できるようにさえなってきている．これらは機能や動作状況をあらかじめ正しく見積もることが如何に困難であるかを示しており，この決定を後回しに出来る機能は今後ますます重要になっていくものと考えられる．

布線論理の汎用性

以上説明した汎用性は，現在のところハードウェアとしては CPU+メモリという構造が担っている．しかしながら，CPU+メモリを前提としたプログラム (program logic, プログラム論理) は機能を実現する一手段に過ぎない．異なる手段として，論理回路 (wired logic, 布線論理，配線論理ともいう) によっても任意の機能を実現することができる．プログラム論理は機能を時間方向に展開したものであり，布線論理は機能を空間方向に展開したものである．図 2.6 に示したように，ASIC(Application Specific Integrated Circuit) では，機能は製造時に決められてしまうため汎用性は全くないが，ハードウェアの本質としての高速性を持っている．FPGA は製造後に布線論理をプログラムすることが出来るという点で画期的であるが，その汎用性は FORTRAN レベルであり，CPU+メモリの汎用性には及ばない．CPU+メモリはプログラム論理なので高い汎用性を持っているが，逐次処理であるため，布線論理の本質である並列性と高速性が活かされない．

PCA は，以上のことを踏まえ考案された新しいアーキテクチャである．

2.2.2 PCA の基本構造

PCA の基本構造は布線論理を汎用とするための必要条件と今後主流となると思われるメッシュ構造のネットワークを組み合わせることにより得られたものである．

図 2.7 に，組込部と可変部から成る PCA セルを示す．PCA の基本構造は，この PCA セルを 2 次元メッシュ状に並べた構造となっている (図 2.8).

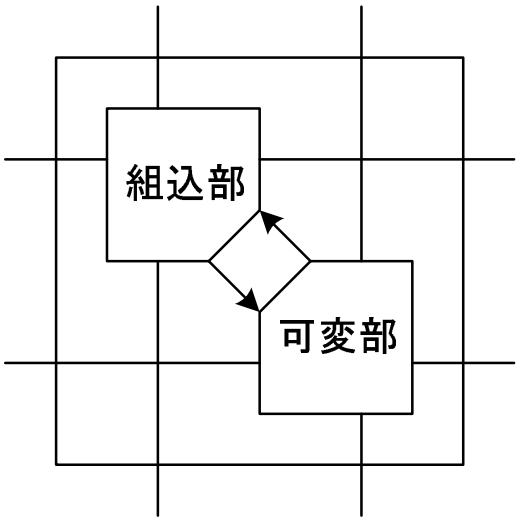


図 2.7 PCA セル

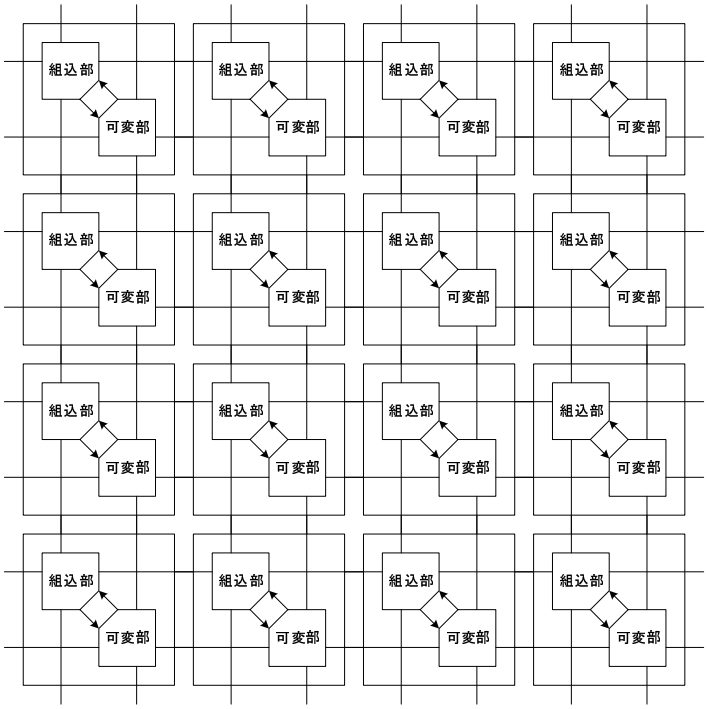


図 2.8 PCA の基本構造

2.2.3 動的再構成機能

PCA は動的再構成を可能にしている．その仕組みは，PCA では処理が可変部で行われることによる．可変部は組込部により構成が変更されるため，構成を変更する必要がある場合，動作中であろうと可変部の構成を変更することができる．

このようにして，動的再構成は，PCA においては機能として可能になっているが，一般的には，ソフトウェアにおけるオーバレイのように，大きな回路図の一部を FPGA や DRP(Dynamically Reconfigurable Processor)[8] にマッピングし，処理が終了したら切り替えて使用する，といった形で行われる．

動的再構成が可能になると，必要に応じて回路を構成することができ，様々な処理を行うことが可能になる．同期式の場合，動作中に回路構成を変化させるのは制御が難しいため，PCA はグローバルクロックを用いない非同期式を採用している．

2.2.4 PCA-1

PCA-1[13, 14] は NTT により開発され，PCA の概念の正当性を評価することを主目的として，その概念を最初に実現したものである．

PCA-1 では可変部に Sea of LUTs の構造を採用したほか，すべてに非同期式制御を採用するなど，将来性に優れたアイディアを盛り込みながら，PCA の概念を具体化することに成功した．

Sea of LUTs による可変部の構成

Sea of LUTs の構造とは，LUT(Look-up Table) すなわちメモリのみを敷き詰めた規則正しい構造である．PCA-1 における可変部の構成を図 2.9 に示す．

基本セルは 4 つの 4 入力 1 出力 16 ビット LUT で構成されている．各 LUT は任意の組合せ回路を実現するばかりでなく，配線としての論理も実現できる．また，隣接する基本セル内の 2LUT でフィードバック回路の論理を構成することにより，ラッチや Muller の C 素子も実現できる．この基本セルのアレイ構造を Sea of LUTs と呼び，これが論理的に任意の回路をマッピングできる可変部の構造となっている．

PCA-1 における 1PCA セルの可変部は 8×8 の基本セルのアレイによって構成される．つまり，1 個の組込部が管理する対象の可変部は 64 基本セル (=256LUT) で構成されており，$64 \times 64 = 4096$ ビットの記憶容量を持っていることになる．これらは，1 ワードを 4 ビットとして，1024 ワードの記憶容量を持つメモリとしてみなされ，組込部から書き込み・読み込みが行われる．

組込部の構成

PCA セル内の組込部は，独立に動作する 5 つのステートマシン (入力ポート) を持つ (図 2.10)．これらは PCA-1 上でルーティングスイッチを形成するとともに可変部の制御を行う．5 つの入力ポートは同一の構造で，それぞれ東西南北，あるいは可変部からの要求を

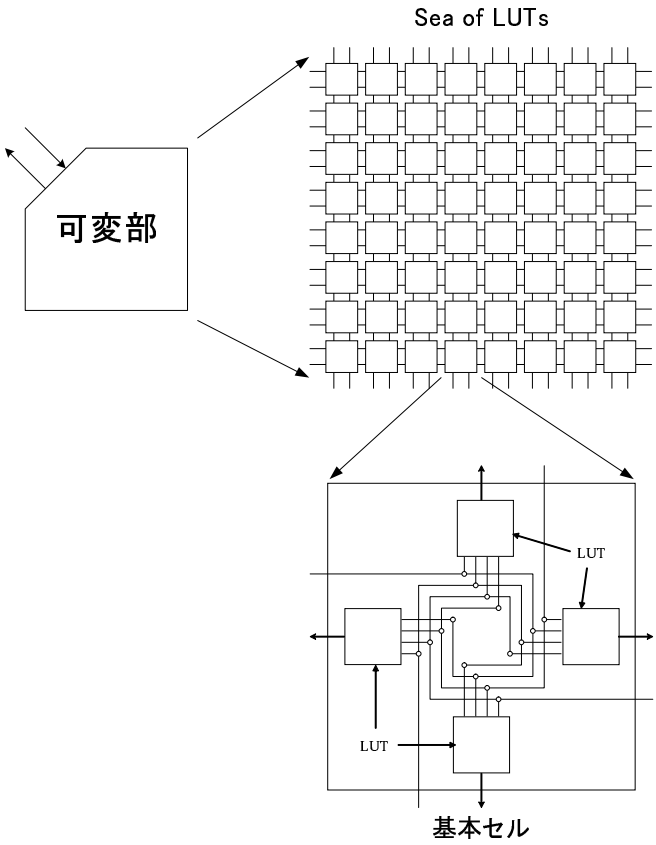


図 2.9 可変部の構成

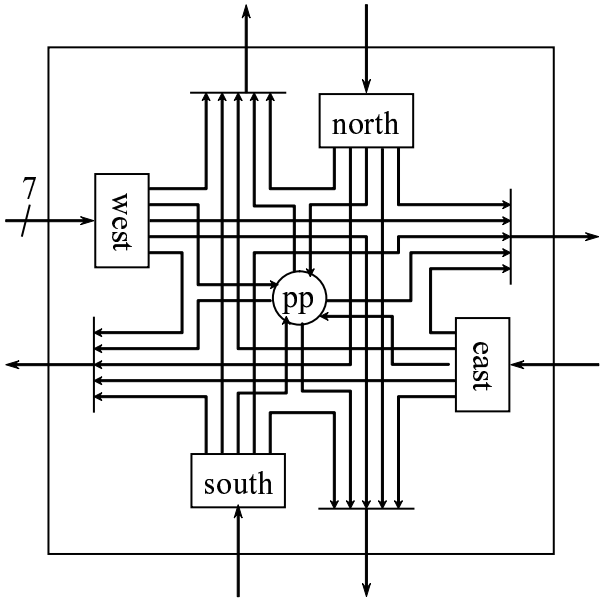


図 2.10 組込部の入出力ポートの構成

受け付け，入力ポートの状態に応じた処理を行う．要求は 11 種のコマンドによって行われ，これらは 5 ビットでコーディングされている (表 2.2).

表 2.2 PCA-1 のコマンド

コマンド	ビットパターン	動作
clear	0XXXX	経路解放
open	1000X	接続
close	1001X	切断
config_out	10100	読み出し
config_inout	10101	コピーメッセージの出力
config_in	1011X	書き込み
west	11000	経路設定 (西)
north	11001	経路設定 (北)
east	11010	経路設定 (東)
south	11011	経路設定 (南)
pp_out	111XX	経路設定 (可変部)

入力ポートでは同時に実行されるコマンドが競合することがあり，コマンドの終了を待たせる場合がある．このため，コマンドの受け付けはハンドシェイクで制御されている．隣接する PCA セルの入力ポート間はコマンドあるいはデータのための 5 ビットとハンドシェイクのための 2 ビットの計 7 ビットで接続される．PCA セルの入力ポートから各出力へ向かう配線は出力ポートでまとめられ，隣接する PCA セルの入力ポートに接続される．したがって，隣接する組込部間は入力ポート 7 ビット，出力ポート 7 ビットの合計 14 ビットで接続されている．

2.3 ビットシリアル PCA

ビットシリアル PCA とは，本研究室で研究している，ビットシリアル処理に特化した PCA のことである．

2.3.1 ビットシリアルの特徴

データの転送方式にはビットパラレル方式とビットシリアル方式がある．ビットパラレル方式は，従来のインタフェースに用いられている伝送方式であり，複数の信号線を用いてデータを送受信する方式である．一方，ビットシリアル方式は，USB や SATA(Serial ATA) などで用いられている伝送方式であり，単一の信号線を用いてデータを送受信する方式である．ビットパラレル方式は複数の信号線を用いるのでビットシリアル方式よりもデータを高速かつ大量に送受信できると思われがちだが，実際はそうとも限らない．なぜなら，ビットパラレル方式ではデータの到着の時間差 (スキュー) が発生するからである．動作速度を上げようとクロック周期を短くすると，このスキューの問題が無視できなくな

る．対して，ビットシリアル方式は一度に送れるデータ量は少ないものの，このスキューの問題を考慮する必要がない．この 2 つの転送方式における速度の差は用いる用途により変化するため，一概にビットシリアル方式の方が遅いと言うことはできない．

本研究室で研究しているビットシリアル PCA は，ビットシリアル方式を採用している．その理由として，ビットパラレル方式に比べ信号線間の干渉やノイズが抑えられ，単純な配線によってハードウェア量も抑えられる，といったことが挙げられる．

2.3.2 圧力による領域管理

PCA は回路が必要に応じて新しい回路を構成しこれと協調して動作することを可能にするアーキテクチャである．PCA-1 による動作実験により回路が自分の横に回路を作り出してこれと協調して動作するという設計が意図どおりに動作することを確認できた．しかしこの設計では回路の横に空き領域があることが前提となっており，どのように領域を管理するかについては考慮されていない．領域を管理する回路を設けるという方法はすぐに思いつくことができるが，この方法では同時動作しているたくさんの回路からの領域割り当て要求をどう処理するのかという問題が発生する．管理するためには情報の一元管理が必要でこれを行う回路の処理能力をいくらでも高められるはずはないからである．また必要な処理能力に応じて管理を分割するという改良を加えたとしても割り当て要求をどこに送ればよいのかという問題が発生する．

そこで管理機構によらず，回路がそこにあるかないか，そのことがすべてであると考えることにより，細胞分裂や生物の増殖を真似た，圧力による空き領域確保というメカニズムを思いつくことができる (図 2.11)．これは新しい回路の生成は隣接した空き領域に限ることとし，隣接部分のすべてに回路が存在する場合には，その隣接回路に圧力をかけて移動させ，空きを作るというアイディアである．この方法では複数の領域確保が同時に発生しても圧力をベクトルとして加算するだけでよく，また隣接した領域に新しい回路を作るため，協調動作を行うための接続が最短距離で行えることとなる．

領域管理は動的再構成可能アーキテクチャにとって大きな問題であったが，この圧力による領域管理は一つの解となっているものと思われる．また回路の移動に伴って回路間の接続も移動しなければならないので，圧力による領域管理は，回路の接続関係が複雑でないビットシリアル方式との相性が良いものと考えられる．

2.4 ビットシリアルペトリネットを用いた非同期回路のモデル化

2.4.1 ペトリネット

ペトリネット [20, 21] とは，1962 代にドイツ人の Carl Adam Petri によって考案された理論的計算モデルで，並列動作，分散状態，資源の概念を表現可能な有向グラフである．

設計者の負担となる非同期回路の複雑な構造を，ペトリネットに置き換えることで抽象度が上がり分かりやすくすることが出来る．ペトリネットは，離散事象システムをモデル化するものである．離散事象とは，ある時間で決まって起こることではなく，たまに起こる事象のことである．分かりやすい例を挙げると，電話やメールなどのことである．

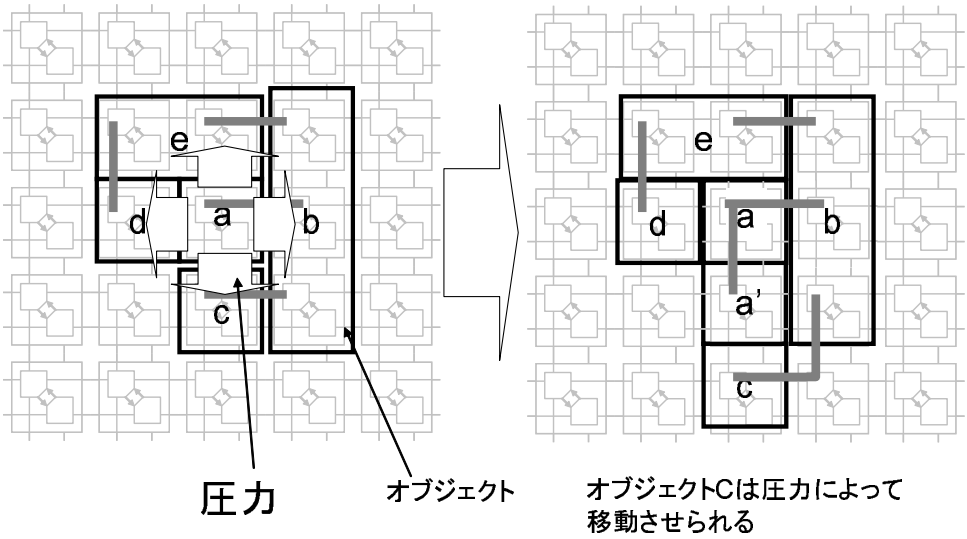


図 2.11 圧力による領域管理

ペトリネットは，プレース，トランジション，アーク，トークンと呼ばれるものからなっている．通常，プレースは白丸，トランジションは棒もしくは長方形で描かれ，アークは矢印，トークンは黒丸で描かれる．プレースはトークンを持ちトランジションの入力側に接続されているプレースがすべてトークンを持っているならば状態が変化する (トランジションが発火する)．アークはプレースとトランジションを繋ぐものである．図 2.12 にペトリネットで用いるプレース，トランジション，アークを，図 2.13 にトランジションの発火の様子を示す．

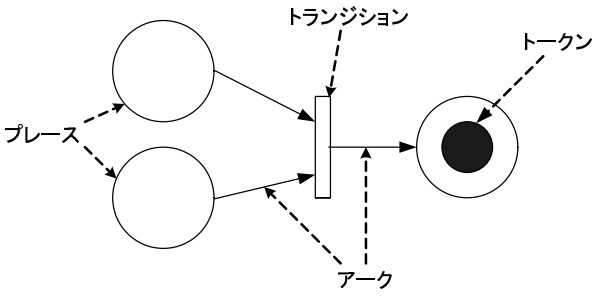


図 2.12 ペトリネットの要素

2.4.2 非同期回路のモデル化

前述した Muller の C 素子を用いて非同期回路の設計を行う場合，すべての信号間の通信は非同期的に行われるため，信号伝播を追尾しながらの設計は非常に困難である．そこで，本研究室において，非同期パイプラインをペトリネットで表現する手法が提案され，そのシミュレーションを行うための設計ツールが実装された．

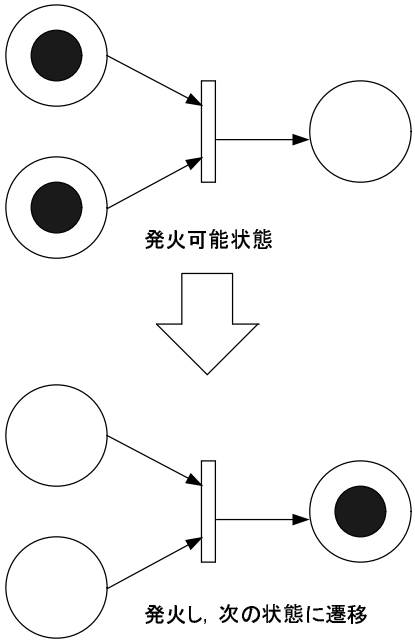


図 2.13 トランジションの発火の様子

非同期パイプラインの表現のペトリネット化

図 2.14 に示すように，C 素子をペトリネットのトランジション，C 素子の入力線を前段プレース，C 素子を反応させることが出来る入力値をペトリネットのトークンと対応づけて表現する．

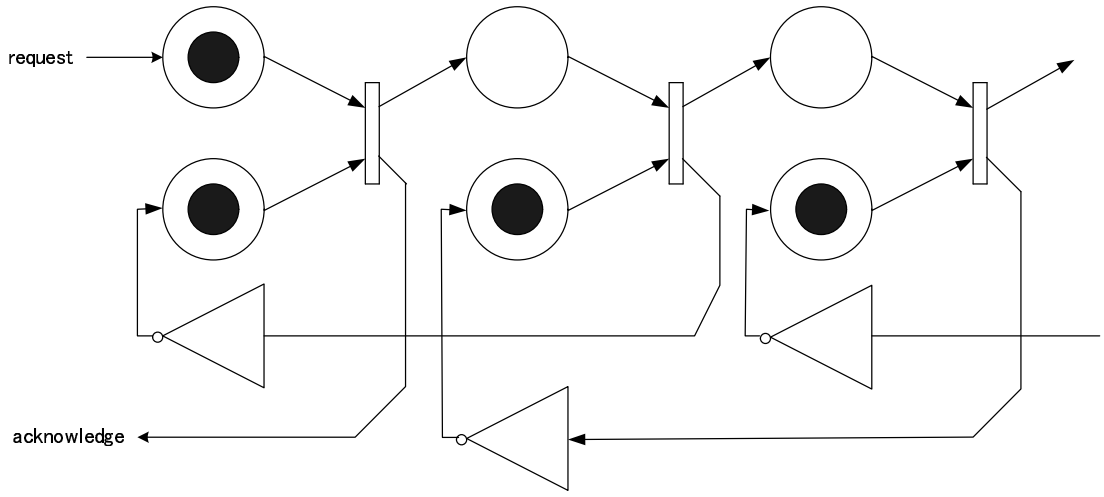


図 2.14 C 素子を用いた非同期パイプライン

こうすることで，C 素子の反応をトランジションの発火，信号の変化をトークンの移動とみなすことができ，これらにより非同期回路の動作を観察することができる．

図 2.14 において上段のプレース列は処理されるべき情報を保持し，下段のプレース列は処理を行うタイミングを制御していると考えることが出来る．パイプラインは必ずこの 2 段がセットになっているため，出力先のプレースにトークンがないことが発火の条件とすると，図 2.15 のように単純化することが出来る．

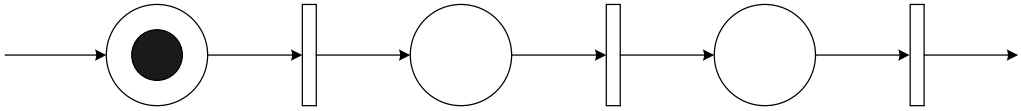


図 2.15 簡略化した表現

2.4.3 ビットシリアルペトリネット

C 素子による非同期パイプラインがペトリネットで表現できることはこれまでに述べた．そこで，さらにパイプラインによる多入力，多出力ビットシリアル演算のための要件を満たすため，次のような検討を経てビットシリアルペトリネットが考案された．

- C 素子の発火によって値を保持するラッチの導入
- 双方に分岐するパイプライン (fork)
- 待ち合わせによる合流 (join)
- 選択分岐
- 競合する合流

それぞれの拡張は，通常のペトリネットで示され，矛盾なく実際の回路を表現できるモデル化が行われた．

2.4.4 非同期ビットシリアルペトリネットシミュレータ QROQS

QROQS[22–25] は，非同期回路によるビットシリアル演算のための設計ツール・シミュレータである．QROQS は，前述したビットシリアルペトリネット (BSP) に基づいており，GUI を採用しているため，非同期回路の設計とシミュレーションをグラフィカルに行うことが可能になり，設計にかかる時間と手間を減少させることができる．QROQS の概観を図 2.16 に，その要素を図 2.17 に示す．

QROQS の要素

- プレース (図 2.17(a))

  プレースは，トークンを保持する．トークンには，黒トークン (2.17(b)) と白トークン (2.17(c)) がある．黒トークンは，1 ビットの値 (0 または 1) を持ち，信号の立上

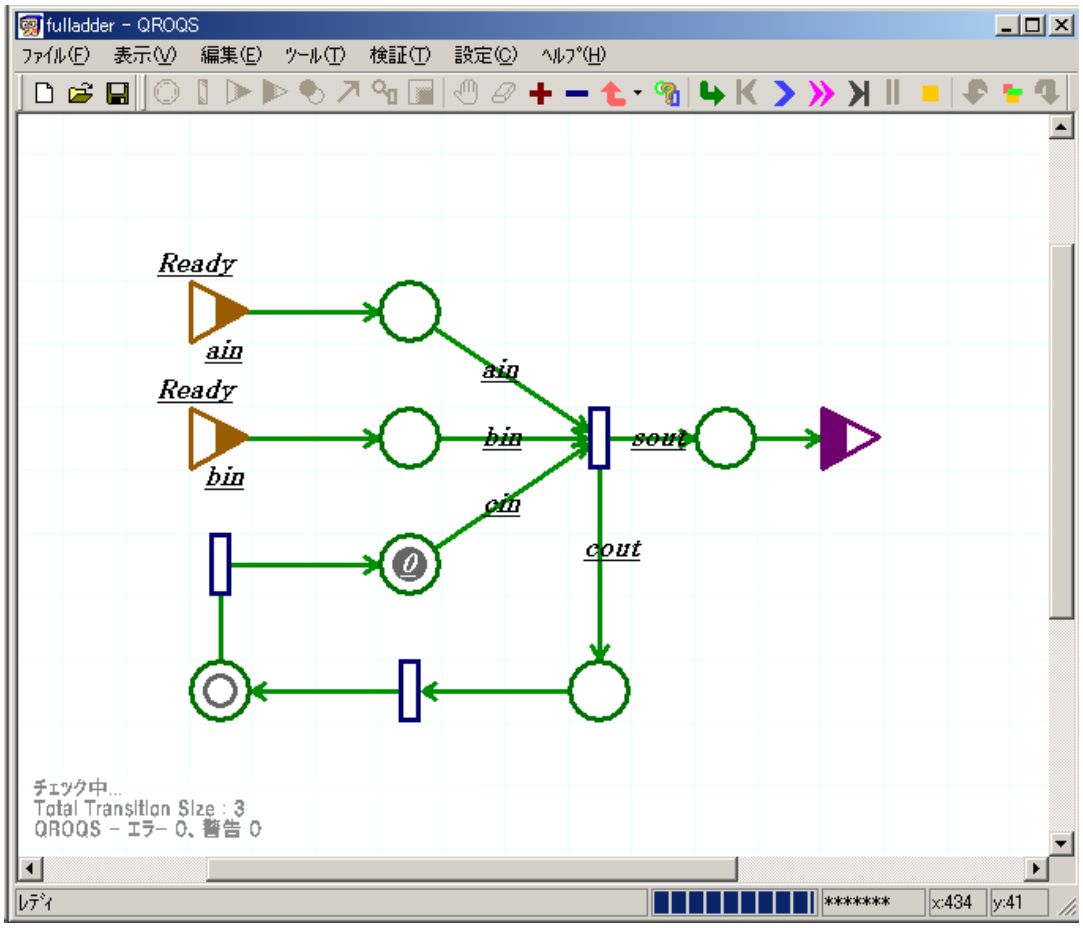


図 2.16 QROQS

(a) プレース (b) 黒トークン (c) 白トークン

(d) トランジション (e) ウージ (f) ケムマキ (g) モジュール

図 2.17 QROQS の要素

りを表す．トランジションの発火が起こると値が書き換えられる．白トークンは，値を持たないトークンであり，信号の立下りを表す．トランジションの発火が起こっても，評価は行われない．黒トークンと白トークンを用いることで，4 サイクルハンドシェイクプロトコルを表現している．

- トランジション (図 2.17(d))

  トランジションは，入力プレースのすべてにトークンがそろったら発火する．発火が起こると，入力プレースにある黒トークンが持つ値を入力として，トランジションが持っている 1 ビットの論理演算を行う関数が評価され，その結果がすべての出力プレースに渡される．渡されるトークンは，入力プレースと同じ色のトークンである，ただし，白トークンの場合は関数の評価は行われず，白トークンをすべての出力トークンに配置するだけである．トランジションは，回路の組み合わせ回路つまり演算などの処理に相当する．トランジションで用いることが出来る演算子を表 2.3 に示す．演算子の優先順位は，表の上へ行けば行くほど強くなる．また，トランジション関数には，ガード式と呼ばれるものも記述できる．ガード式は，トランジション関数の最初に記述され，“?” を用いて表される．ガード式を用いると，ガード式が負のときはトランジションの発火を防ぐことができる．また，1 つの入力プレースから複数のトランジションに接続する場合も，トランジションの同時発火が起こらないようにガード式を記述する必要がある．ガード式の例を図 2.18 に示す．

表 2.3　トランジションで用いられる演算子

演算子	機能
()	小括弧のみ
~	否定 (NOT)
&	論理積 (AND)
^	排他的論理和 (XOR)
|	論理和 (OR)

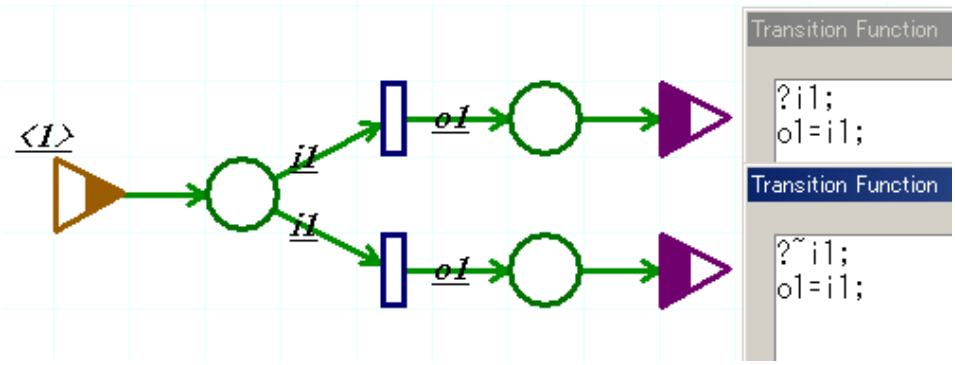


図 2.18　ガード式の例

- ウージ (図 2.17(e))

  ウージは，プレースへ値を与えるトークンを生産する．つまり，ペトリネットの入力として用いる．ウージには，一定値 (0 または 1) またはビット列を設定することが出来る．一定値を設定した場合は，ウージは回路生成時から不変のレジスタを意味し，ウージから生成される黒トークンの値は常に一定となる．対して，ビット列が設定された場合は，回路へのデータ入力を意味し，ビット列から 1 ビットずつ取り出し黒トークンの値とする．シミュレーション時には，ウージが保持していたトークンが消費される度に黒トークンと白トークンを交互に生産する．

- ケムマキ (図 2.17(f))

  ケムマキは，トークンを消費する．つまり，ペトリネット回路からの出力として用いる．ケムマキへの入力プレースに黒トークンが配置されると黒トークンの値を設定された出力先に出力する．このとき黒トークンは消費され，プレースは空になる．黒トークンと白トークンは交互に消費されなければならない．

- モジュール (図 2.17(g))

  モジュールを用いてビットシリアルペトリネットを分割して設計することで，分かりやすく階層的な実装を行うことが出来る．また，同じ回路を繰り返し使うときも，繰り返し使用する回路をモジュールとして用意しておけば，設計の手間が省ける．ただし，全加算器などをモジュールとして使うとき，一つのモジュールを二度使う (再利用する) ときに注意が必要となる．全加算器では桁上げを用いて計算が行われるが，初期化などを行わなければ，一度目の使用時の最上位ビットの桁上げが 1 の場合，二度目の使用時の最下位ビットの桁上げとして 1 が入力されるからである．

QROQS による回路の実装例

QROQS で全加算器を実装したものを図 2.19 に示す．

QROQS のその他の機能

QROQS は，設計した回路を，QROQS 依存の C 言語コードに変換し出力したり，論理合成可能な Verilog HDL を出力することが出来る．

2.4.5 回路増殖を表現できるシミュレータ tanoqs

tanoqs[26, 27] は，QROQS の問題点についての改善を行い作られたシミュレータである．tanoqs の概観を図 2.20 に，その要素を図 2.21 に示す．

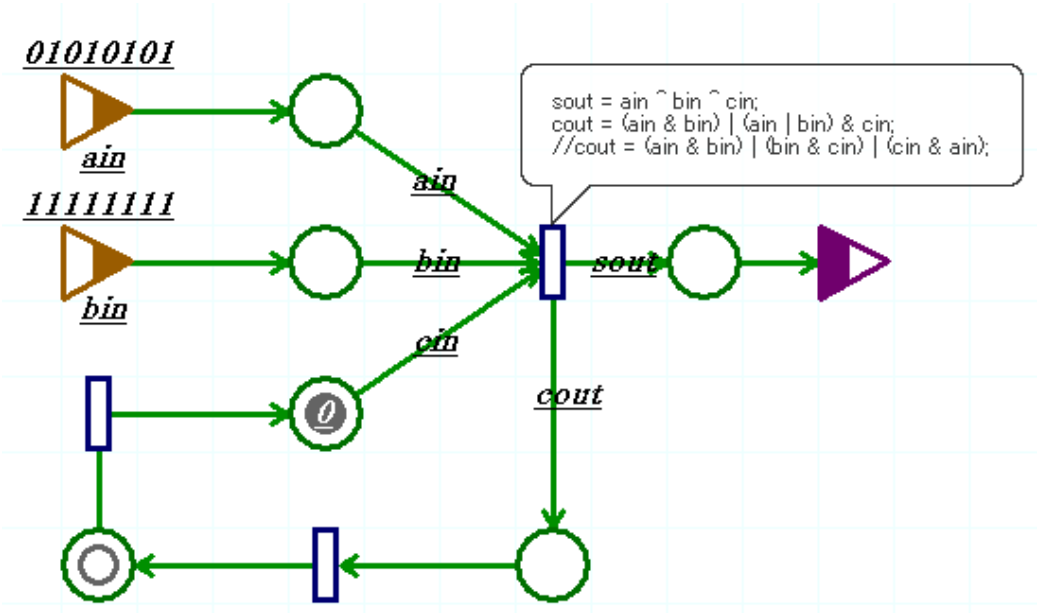


図 2.19 QROQS において実装した全加算器

tanoqs の要素

- シフトレジスタ (図 2.21(a))

  シフトレジスタはステートマシンの入出力，バッファなどとしてデータを保持するためあるいは時間遅れを実現するために使用する．シフトレジスタの保持ビット数を設定することにより記憶ビット数だけでなく遅れ時間数を定義する．また，シフトレジスタは通常 1 入力 1 出力であるが，複数の入力を繋ぐことにより競合合流を表現できる．シフトレジスタにはあらかじめ値を入力しておくこともできる．

  シフトレジスタを QROQS で表現した例を図 2.22 に示す．QROQS では入力をそのまま出力するバストランジションを用い，黒トークン，白トークンを交互に配置することによりシフトレジスタを表現する．

- ステートマシン (図 2.21(b))

  ステートマシンは複数の状態を保持することができ，状態ごとに処理式，状態遷移式を記述し，どの状態が初期状態であるかを指定することによりステートマシン全体の機能を表現する．ステートマシンは現状態の入力がすべて揃うまで待ち合わせを行う．また複数の入出力に対し，状態ごとに異なった部分を処理式，状態遷移式の中で指定することにより待合合流のどれだけの入力を待ち合わせるか，また複数分岐あるいは選択分岐を表現できる．

  ステートマシンを QROQS で表現した例を図 2.23 に，この時のトランジション関数を表 2.4 に示す．トランジションへの入力 $i4$ と $i5$ は現在のステートマシンの状態を表している．QROQS ではトランジションは一つの関数しか持つことができず，状態ごとに違った処理を行おうとすると関数が複雑になってしまう．

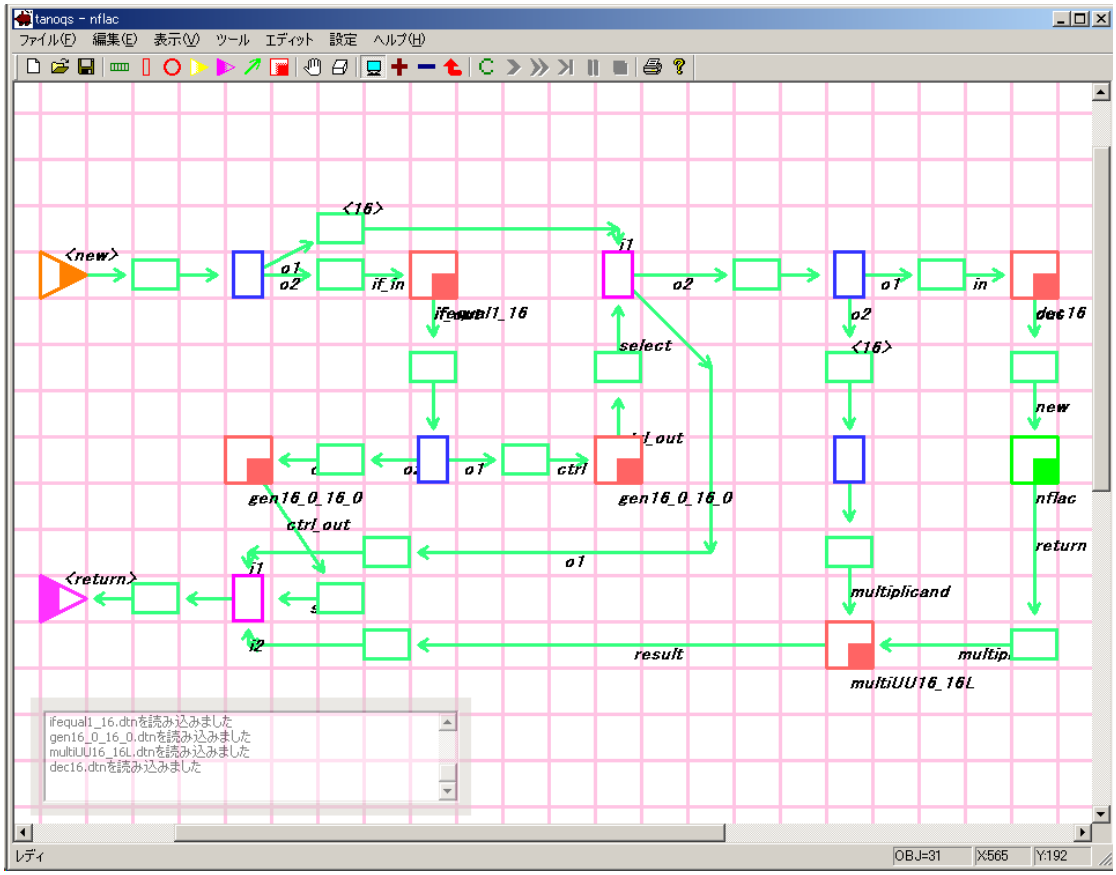
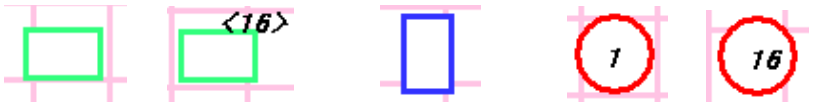


図 2.20 tanoqs の概観



(a) シフトレジスタ (b) ステートマシン (c) カウンタ

(d) ウージ (e) ケムマキ (f) モジュール

図 2.21 tanoqs のプリミティブ

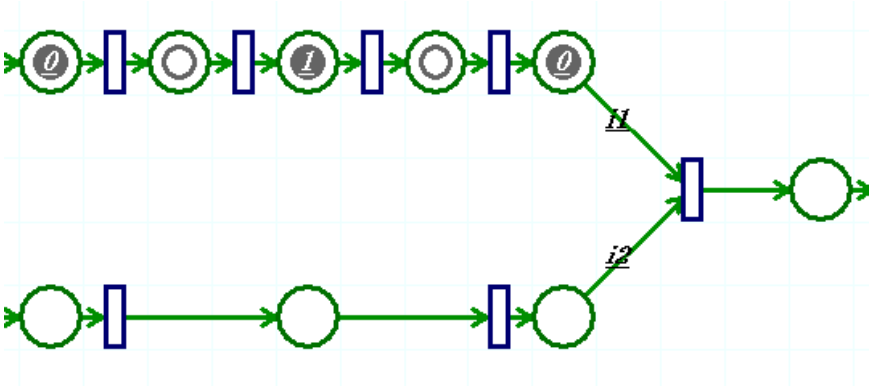


図 2.22　QROQS におけるシフトレジスタ

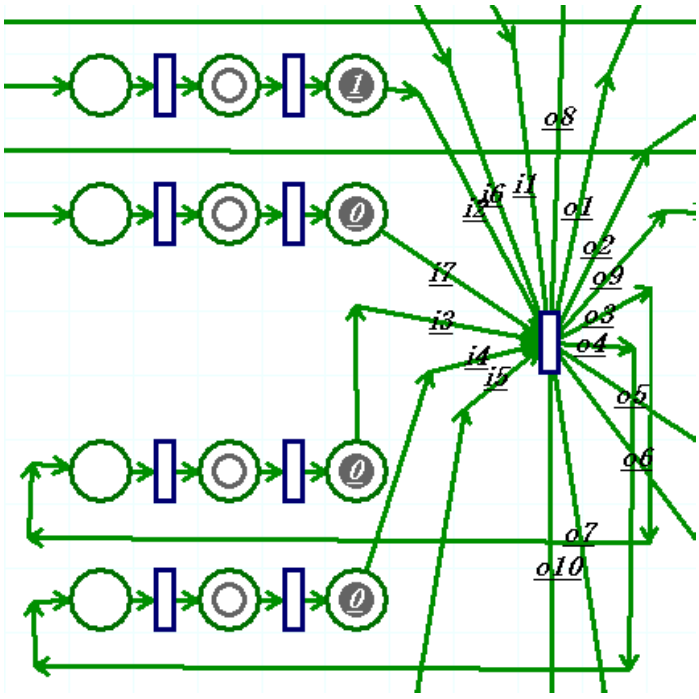


図 2.23　QROQS におけるステートマシン

表 2.4 図 2.23 におけるトランジション関数

```
o1=(i5&(i1^i2^i3))|~i5&i1;
o2=i2;
o3=i5&(~(i1^i4)&(i2|i3)|i2&i3)|~i5&i3;
o4=i5&i4|(~i5&(i2&i3|~i2&i4));
o5=~(i4^i3);
o6=~i5&i2;
o7=i5;
o8=i6;
o9=i7;
o10=i6&~i7&i5;
```

- カウンタ (図 2.21(c))

  カウンタはカウント数を設定して使用する入力 1 つ出力 1 つのプリミティブである．カウント数は初期状態ではカウント変数にコピーされている．カウンタは入力が 0 であると 0 を出力し，入力が 1 であるとカウンタ変数を-1 し 0 を出力する．ただし，-1 した結果，カウンタ変数が 0 となる時には 1 を出力するとともにカウンタ数をカウンタ変数にコピーする．

  カウンタを QROQS で表現した例を図 2.24 に，この時のトランジション関数を表 2.5 に示す．このカウンタは 5 ビットカウンタであり，5 つの入力すべてが 1 となったときに *start* に 1 を出力する．それ以外のときは 0 を出力し続ける．このカウンタは 32 をカウントしている．扱うデータがビットシリアルであるため，ビットの区切りをあらわすためにカウンタは必須であり多用される．

- ウージ (図 2.21(d))

  ウージは，データの入力として用いられる．

- ケムマキ (図 2.21(e))

  ケムマキは，データの出力として用いられる．

- モジュール (図 2.21(f))

  回路の規模の増大につれ，ステートマシンやシフトレジスタなどの構成プリミティブ数が膨大になり，接続関係が複雑化してしまう．また，各種演算などの構成が様々な場所で複数使用されることが多々ある．そのため，管理と構築の容易化，また回路の再使用を考え，同じ構成を持ち，複数存在する部分をモジュールとして階層化して設計が行える．

  tanoqs 上ではモジュールプリミティブを追加し，上位モジュールから接続することによってモジュール化と階層構造を表現できる．また，この階層構造は後述する再帰表現にも使用される．

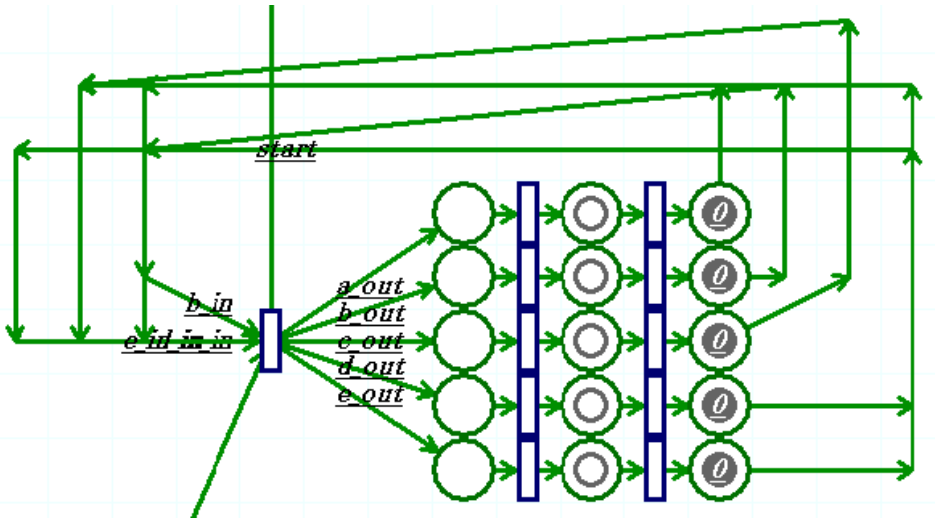


図 2.24 QROQS におけるカウンタ

表 2.5 図 2.24 におけるトランジション関数

```
a_out=a_in^(b_in&c_in&d_in&e_in);
b_out=b_in^(c_in&d_in&e_in);
c_out=c_in^(d_in&e_in);
d_out=d_in^e_in;
e_out=~e_in;
start=a_in&b_in&c_in&d_in&e_in;
```

再帰表現

tanoqs では再帰表現により実行時に回路を任意個追加することができる．設計時に作成したモジュールに new という端子が存在することにより動作時にモジュールの作成を行うモジュール，delete という端子が存在することにより動作中にモジュールの削除を行うモジュールと判断される．このような new あるいは delete 端子を持つモジュールの配下に同一種のモジュールを含ませることにより任意個のモジュールを作成，削除できる．new と delete は本来独立であるが，tanoqs では delete は new を持った場合に限り持つことができる．

2.5 高速フーリエ変換 (FFT)

ここでは，本研究で用いる高速フーリエ変換 (以下 FFT:Fast Fourier Transform)[17, 18] について述べる．

高速フーリエ変換 (FFT) が一般に知られるようになったのは，1965 年の J.W.Cooley と J.W.Tukey による短い論文 [19] からとされている．FFT があまり知られていかったこ

ろは，N 点の離散フーリエ変換 (DFT) を計算するためには N^2 回の計算が必要であると信じられていた．しかし，FFT を用いると $N \log N$ に比例する計算で済む．この FFT の基本原理は，簡単な添字の変換で大きなサイズの DFT を計算が楽な小さな DFT に分解するという考えに基づく．

まず，N 点の DFT

$$X_k = \sum_{j=0}^{N-1} x_j W_N^{jk} \tag{2.1}$$

$$W_N = e^{-2\pi i/N} \tag{2.2}$$

を素直に計算する場合を考える．この場合，A_0 から A_{N-1} までの各項の計算に N 回の乗算が入るため，全体で N^2 回の乗算が必要となる．しかし，もし N が 2 で割り切れるならば，添字 k を偶数と奇数に分けることで N 点の DFT は二つの $N/2$ 点の DFT

$$X_{2k} = \sum_{j=0}^{N/2-1} (a_j + a_{N/2+j}) W_{N/2}^{jk} \tag{2.3}$$

$$X_{2k+1} = \sum_{j=0}^{N/2-1} (a_j - a_{N/2+j}) W_N^j W_{N/2}^{jk} \tag{2.4}$$

に容易に分解できる．$N/2$ 点の DFT は素直に計算して $N^2/4$ 回の乗算で実行できるので，この分解で計算量は約半分に減ることになる．さらに，この分解を 2 回 3 回と繰り返せば計算量は約 1/4, 1/8 と激減する．これが Cooley-Tukey FFT(正確には，基数 2，周波数間引き Cooley-Tukey FFT) の基本的な考え方になる．図 2.25 に，サンプリング数が 8 のときのこの FFT の計算の概略を示す.

この分解を $\log_2 N$ 回行い，1 点の自明な DFT になるまで行ったときの計算量を考える．この分解自体には各々の段で W_N^j を乗ずる $N/2$ 回の複素数乗算と N 回の複素数加算が必要で，複素数乗算回数は $(N/2)\log_2 N$ に減少する．したがって，浮動小数点演算の量は $N \log_2 N$ のオーダーとなる．これは，Cooley-Tukey FFT の典型的な演算量で，様々な FFT の演算量の削減アルゴリズムは，基本的にこのオーダーの比例定数と，$N \log_2 N$ より低い次数の項を小さくするものである.

2.6 関連研究のサーベイ

PCA に関連する研究として，NTT では PCA-1 に引き続き，PCA-2 が開発された．PCA-2 は，PCA のアーキテクチャおよびハードウェア実現方式の将来性の高精度な確認と，高性能なハードウェア実験環境の構築とを目的として開発されたものである．PCA-2 の設計に際しては，PCA-1 の評価結果に基づくアーキテクチャおよび設計手法の改良と先端プロセスの利用による性能向上が図られ，具体的には，集積度の向上，アーキテクチャの改善，非同期式回路設計の改良が行われた.

京都大学の中村研究室では，PCA-Chip2 という同期式を用いた PCA の試作デバイスが開発され，このデバイスを対象とし様々な検討が行われている [28].

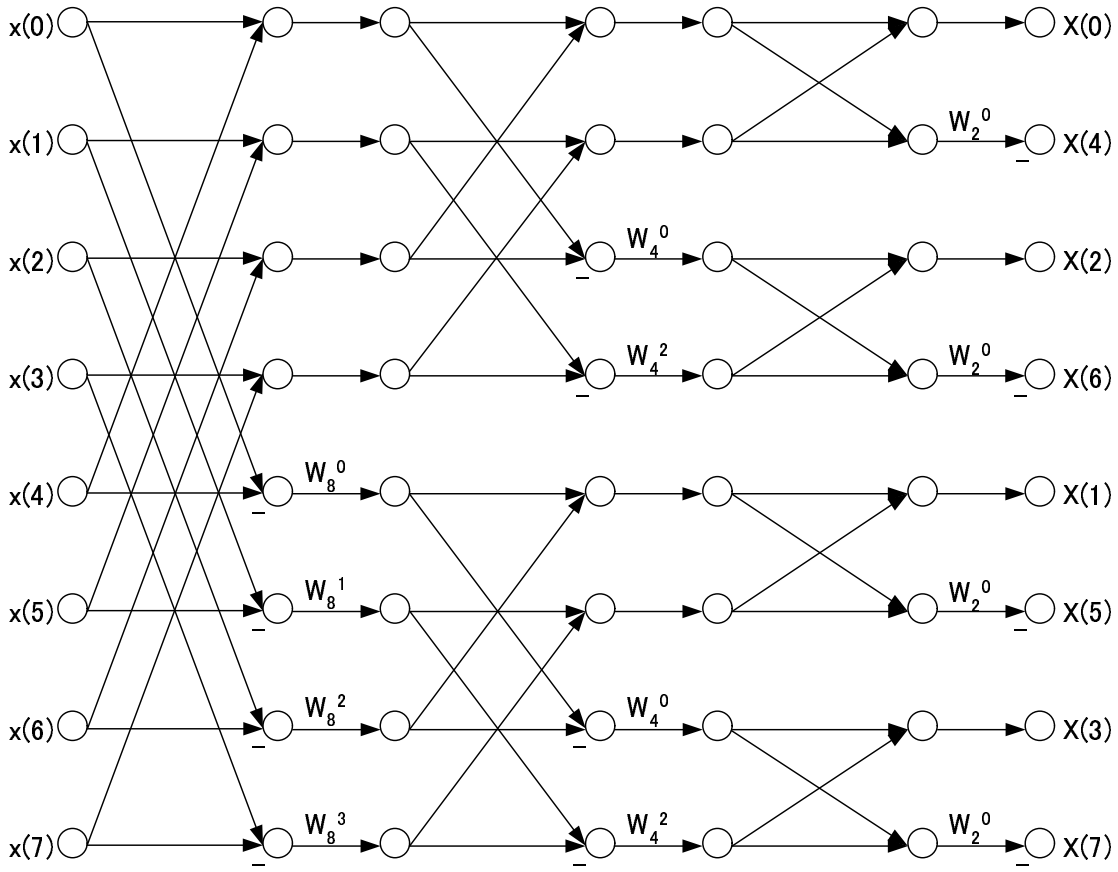


図 2.25 基数 2 周波数間引き FFT のデータフロー

東京大学の南谷研究室では，非同期式マイクロプロセッサである TITAC-2[29] が開発された．これにより，実用レベルマイクロプロセッサをクロックなしで実現できる設計技術があることが示された．

また，NTT 未来ねっと研究所で PCASE，PCASIM，PCASIM II[30]，京都大学の中村研究室で PCACAD[31] が開発されている．これらは，PCA での非同期回路を記述できるハードウェア記述言語 (HDL) が存在しないために開発された，レイアウトレベルでの設計を支援する設計ツールである．

第3章

研究の目的と意義

本章では前章で述べた従来の研究の問題点を指摘し本研究の目的と意義を明らかにする.

3.1 既存の研究の問題点

近年のハードウェア設計では，ハードウェア記述言語 (HDL) を用いるのが主流であるが，PCA における動的に回路を追加・削除する機能を持った非同期回路を設計することのできる HDL は今のところ存在しない.

本研究室では，HDL を用いて非同期ビットシリアル方式で，加算器，減算器，乗算器，除算器，FFT 回路，DCT 回路などの設計が行われたが [32–34]，HDL を用いて設計されているため動的再構成機能についての考慮がなされていない.

また，本研究室で開発された非同期ビットシリアルペトリネットシミュレータである QROQS においても様々な回路の設計が行われたが [35]，QROQS は動的に要素を追加・削除する機能を持っていないため，動的再構成機能を想定した設計を行うことができない.

3.2 本研究の目的

そこで，本研究では，PCA の持つ動作中に回路の構成が変化するという特性を活かした回路の設計手法を提案する.

また，その有用性を明らかにするために，高速フーリエ変換 (FFT) を，動的に要素を追加・削除できる非同期ビットシリアルシミュレータ tanoqs を用いて設計，シミュレーションを行い，動作中に構成が変化する特性を活かした回路を設計できることを明らかにする．ここで高速フーリエ変換 (FFT) を用いて検証するのは，5.1 節で後述するが，FFT はソフトウェアにおいて再帰構造を用いて表現することができ，サンプリング数に応じて動作が変わり，比較的構造の複雑なアルゴリズムであることから，動作中に回路の構成が変化するという本研究の要件を満たすためである.

第4章

提案手法

ここでは，ハードウェアにおいて動作中に構成が変化する構造をどのようにして表現するかについて検討する．

4.1 ソフトウェアにおける動的要素の表現方法

まず，ソフトウェアにおける動的要素の表現方法について述べる．

4.1.1 malloc, new

一般的なコンピュータでは資源管理者である OS がメモリを管理しており，C 言語における malloc や C++, JAVA 言語における new といった要求を受け取ると，OS がメモリのヒープ領域の空き部分をソフトウェアに必要なだけ割り当ててくれるようになっている．新たに割り付けられた領域と既存の部分との接続はポインタ変数により行われる．

4.1.2 再帰表現

繰り返し構造を用いるアルゴリズムなどでは，for や while などといったループ部を，関数を再帰的に用いて置き換えることで簡潔に書くことができるようになる場合がある．再帰関数はメモリのスタック領域を用いることで実現されている．新たに割り付けられた領域と既存の部分との接続は関数の引数と返り値により行われる．

4.2 ハードウェアにおける動的要素の表現方法

ここでは，前述したソフトウェアにおける動的要素の表現方法について，それぞれハードウェアにおいても適用できるかを検討する．

4.2.1　malloc, new

ハードウェアにはアドレスの概念がないためポインタ変数などあるはずもなく，そのため新たに確保した領域と既存の領域との接続関係を表現するには新たな表現形式を考案しなければならない．よって，malloc や new といった機能をハードウェアにおいて実現するのはそのままでは無理である．

4.2.2　再帰表現

ソフトウェアにおける再帰表現について考えてみると，malloc, new と違い接続関係が記述されており，領域確保，配置，接続のすべてが形式として用意されていることが分かる．

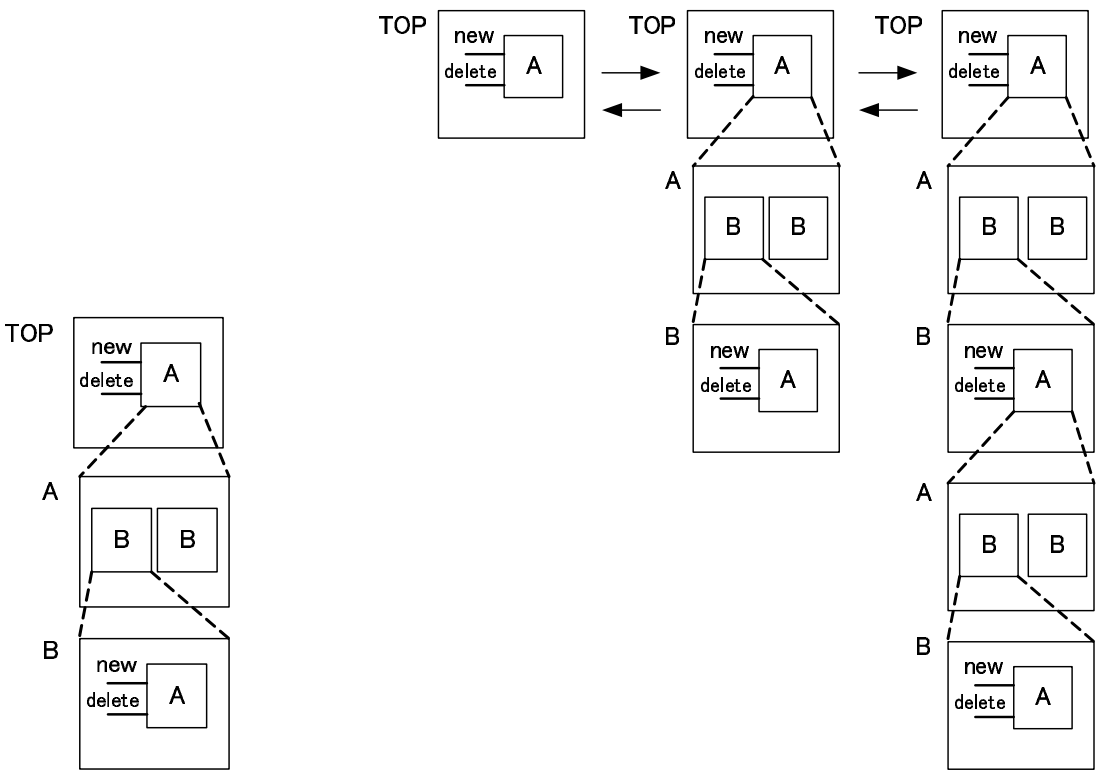
このことから，再帰構造をハードウェアの動的要素の表現方法として使うことができると考えられる．

4.3　再帰表現のハードウェアへの適用

ここではハードウェアにおいて再帰構造をどのようにして表現するかについて考える．

ソフトウェアにおいて再帰が実行されたとき，メモリ領域の確保，配置，入出力の接続が自動的に行われる．

ハードウェアにおいてこのような仕組みを実現する方法を考える．図 4.1 に，再帰構造を用いて表現した回路の例を示す．設計時は，図 4.1(a) のような回路表現となる．実際に動作を開始し，TOP 回路の new 端子に回路を生成する要求が到着すると領域が確保され，新たに回路 A が構成される．回路 A に含まれる回路 B の new 端子に要求が到着するとさらに領域が確保され，新たな回路 A が構成される．逆に，delete 端子に要求が到着すると，下位にある回路が削除される．



(a) 設計時　　(b) 動作時

図 4.1　再帰表現の例

第5章

設計と実装

本章では，非同期ビットシリアル方式において FFT 回路を再帰構造を用いて表現し，動的要素を記述できる設計ツール tanoqs を用いて実装，シミュレーションを行う．

注意点として，布線論理の並列性を活かすために回路のできるだけ多くの部分が同時に動作するような設計を行う必要がある．

5.1 FFT を用いる理由

FFT をソフトウェアにおいて実装すると，再帰構造を用いて表現することができ，サンプリング数に応じて動作が変わり，かつ構造が比較的複雑なアルゴリズムであることから，本研究では FFT を用いて動作を検証することにした．

従来のハードウェアにおける FFT

従来のハードウェアでは，サンプリング数が 4 の場合は 4 用の回路，8 の場合は 8 用の回路，とサンプリング数に応じて回路を複数用意する必要があった．

ここで，ハードウェアにおいて FFT を再帰構造を用いて表現できるとすると，回路図的には 1 つの回路で任意のサンプリング数の FFT に対応することが可能となる．

5.2 設計

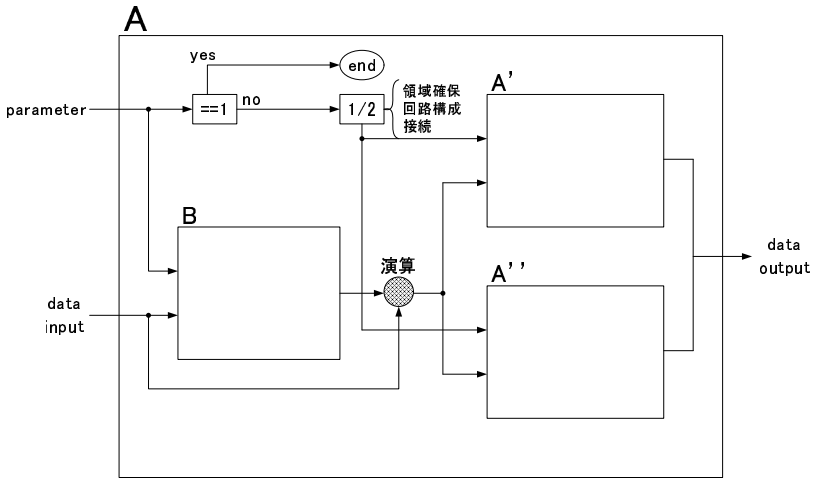
ビットシリアル方式で考えるので，データ入力は 1 つずつ順番に入ってくる．

FFT の計算を行う回路を回路 A と呼ぶことにする (図 5.1(a))．バタフライ演算を行う部分を考えると，1 番目に入ってきたデータと演算を行うのは，サンプリング数を n とすると，$(n/2+1)$ 番目のデータなので，1 番目〜$n/2$ 番目までのデータを記憶しておく必要がある．このデータ数は可変なので，これだけのデータを記憶しておく回路は動的 (再帰的) に構成しなければならない．この回路を回路 B と呼ぶことにする (図 5.1(b))．

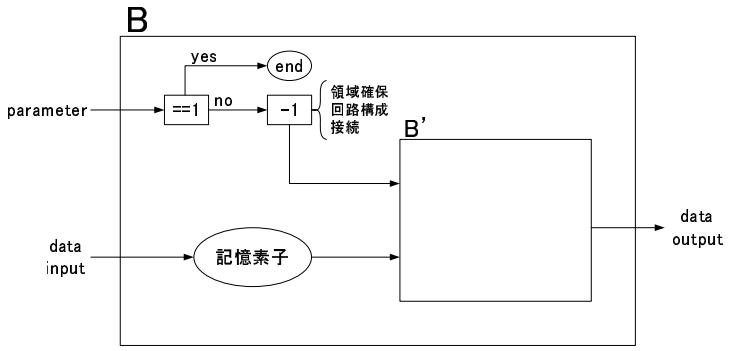
まず，回路 A について説明する．回路 A は，演算を行う回路である．演算部にデータが 2 つ揃うと，バタフライ演算を行う．演算は，入力データ 2 つに対して加算を行ったデータ，回転と減算を行ったデータの 2 つが出力される．ここで，再帰的に演算を行うために，加算を行ったデータを回路 A’，回転と減算を行ったデータを回路 A” に入力する．

回路 A' と A" は，回路 A のパラメータ値が 1 でないなら領域確保・構成・接続が行われて動的に追加される回路である．回路 A' と A" の構成は回路 A と全く同じである．回路 A' と A" のパラメータ入力には回路 A のパラメータ入力を 1/2(1 ビット右シフト) したものが接続される．

次に，回路 B について説明する．保持しておくデータは $n/2$ 個なので，$n/2$ とデータを入力として渡せばよい．この回路 B を再帰構造を用いてリストのように表現する．回路 B のデータ入力は記憶素子により保持する．入ってきたパラメータ値が 1 より大きかったら新たに回路 B' の領域確保と構成を行い，その回路 B' の入力にはパラメータ値から 1 を引いたものとデータ (記憶素子) を接続する．パラメータ値が 1 だったら，新たな回路の領域確保・構成・接続は行わず，入ってきたデータを記憶素子に繋ぎ，その記憶素子をそのまま出力に接続する．



(a) 回路 A



(b) 回路 B

図 5.1　FFT の再帰表現

動作説明

この図 5.1 について，サンプリング数が 8 のときを例にして説明する (図 5.2, 図 5.3, 図 5.4)．サンプリング数は 8 であるため，最上位モジュールの回路 A のパラメータの入力に 4，データの入力が 8 個入る．

まず，動作開始時には図 5.2 の回路 A のみが存在する．パラメータ値に 4 が入力されると，パラメータ値が 1 ではないため，回路 A', A" の領域確保・構成を行い，パラメータ値 2 をそれぞれに接続する．また，回路 B の領域確保・構成を行い，パラメータ値 4 とデータを接続する．このとき回路 B に入力されるパラメータ値は 4 であるため，回路 B はデータを 4 つ記憶する回路となる．回路 B でデータが 4 つ記憶された後，5 つ目以降のデータは回路 B でなく斜線をかけた丸で示した演算部に送られる．そして，1 つ目のデータと 5 つ目のデータ，2 つ目のデータと 6 つ目のデータ，$\cdots$ のそれぞれについてバタフライ演算が行われる．加算を行ったデータを y，回転させ減算を行ったデータを y' とすると，データ y は回路 A' に，データ y' は回路 A" のデータ入力に接続される．

このようにして，図 5.2, 図 5.3, 図 5.4 のような階層構造が構成され，動作を行う．必要なくなった回路は使用後に削除してもよいが，繰り返し計算を行う場合は再利用することも可能である．

また，この構造ではできるだけ回路のすべての部分が同時に動作するようになっているので，布線論理の並列性を活かした動作が期待できる．

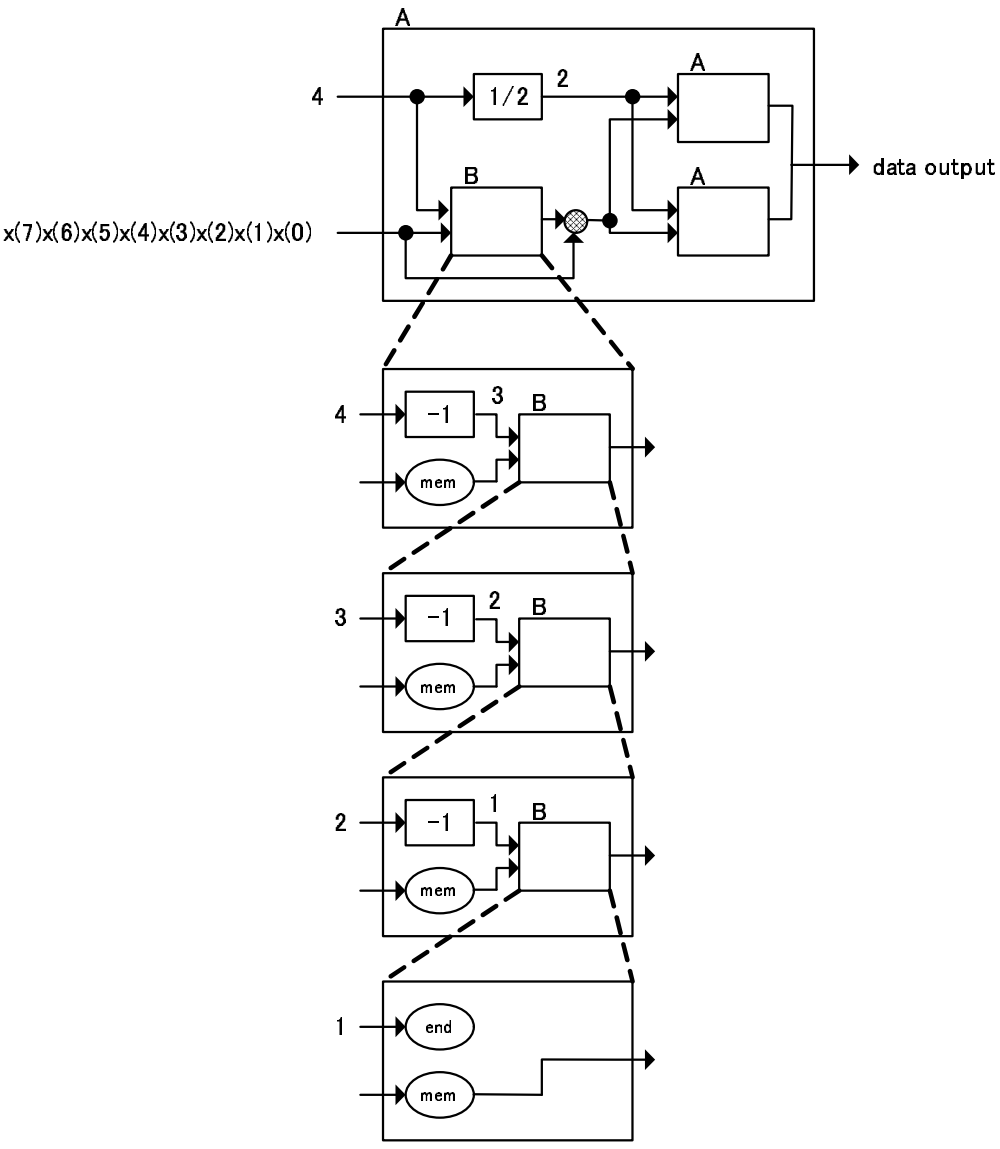


図 5.2　FFT の再帰表現における実行図 (最上位モジュール A)

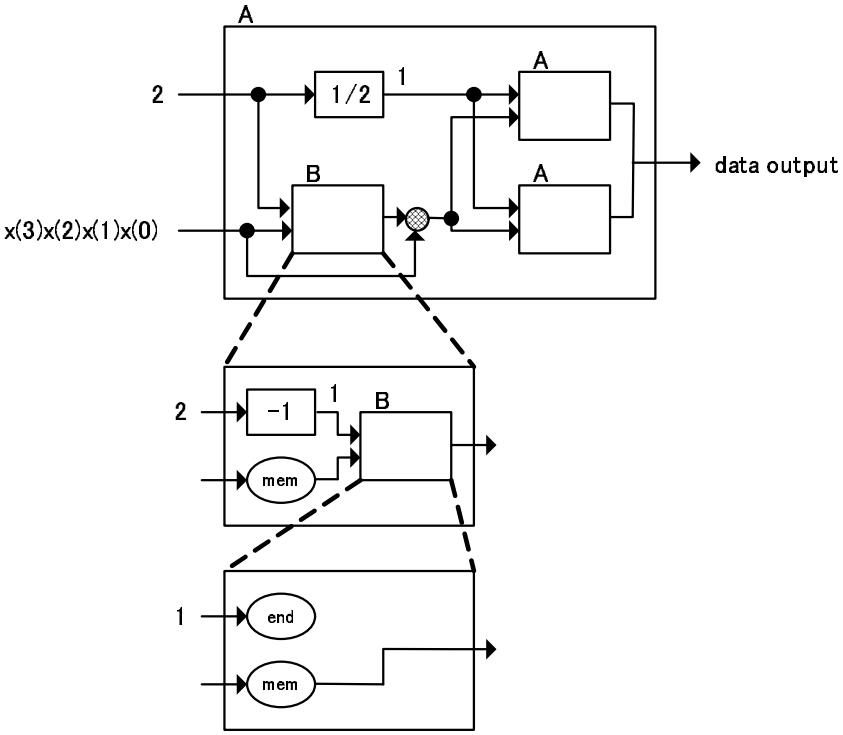


図 5.3　FFT の再帰表現における実行図 (深さ 2 のモジュール A)

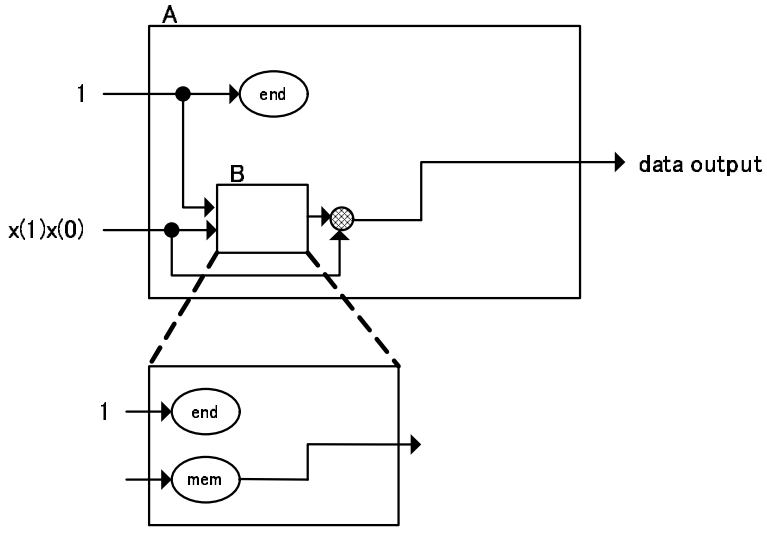
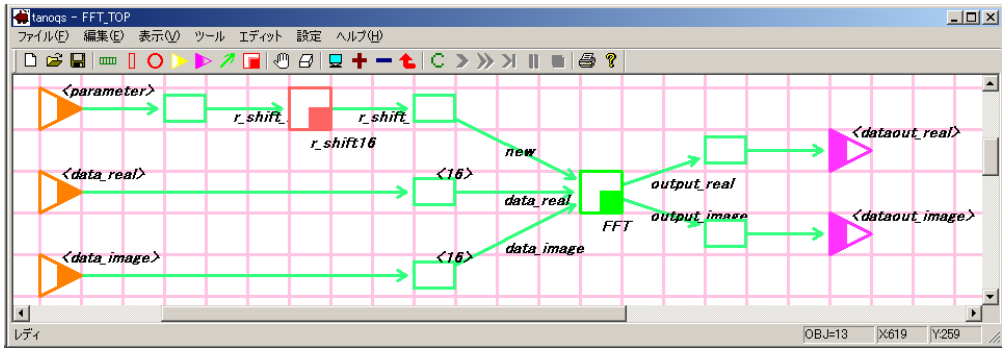


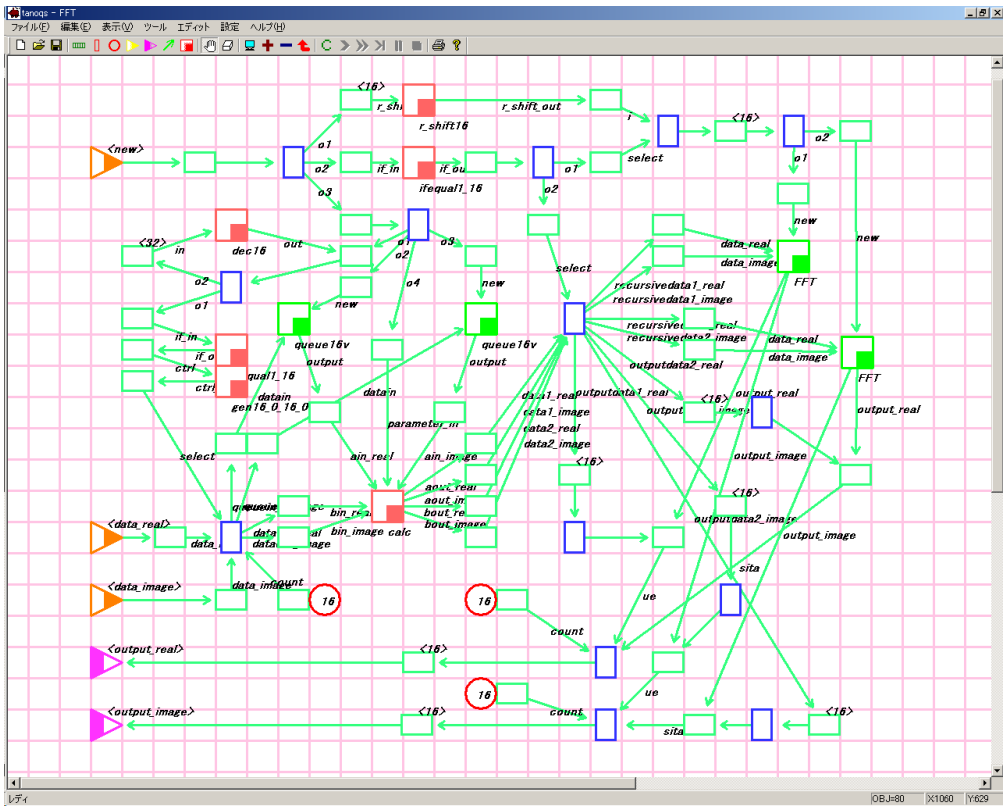
図 5.4　FFT の再帰表現における実行図 (深さ 3 のモジュール A)

5.3　実装

図 5.5 に，tanoqs において作成した FFT を示す．ただし，データは複素数なので，実部と虚部の二つを組として扱う．また，データは実数だが，オーバフローなどを考慮し簡単のために整数部を 15 ビット，小数部を 1 ビットの計 16 ビットで表現した．



(a) TOP_FFT



(b) FFT

図 5.5　tanoqs で作成した FFT

5.3.1　動作説明

設計した FFT の説明を行う．説明のために，サンプリング数は 8 の場合を考える．

まず，図 5.5(a) に示した FFT_TOP が最上位モジュールである．このモジュールに，サンプリング数 8 とデータが 8 個入力される．そして，サンプリング数 8 を 1/2(1 ビット右シフト) して，これを FFT モジュールの new 端子に繋ぐ．new 端子に入力が来ると，このとき FFT モジュールが構成される．それからデータを流す．

次に，FFT モジュールの説明をする．FFT モジュールでは，まず new の値が 1 かどうかをチェックする．1 だったら，演算部の出力を直接 output_real と output_image に出力する．1 でなかったら，1/2 したパラメータ値を下位の FFT モジュールの new 端子に接続し新たにモジュールを生成して，演算結果を生成した FFT モジュールの data 端子に繋ぐ．演算部では，まず queue の new 端子にパラメータ値を接続し，キューを生成する (図 5.6)．このキューは，再帰的に用いられ，new の値が 1 だったらそこで再帰は終了する．

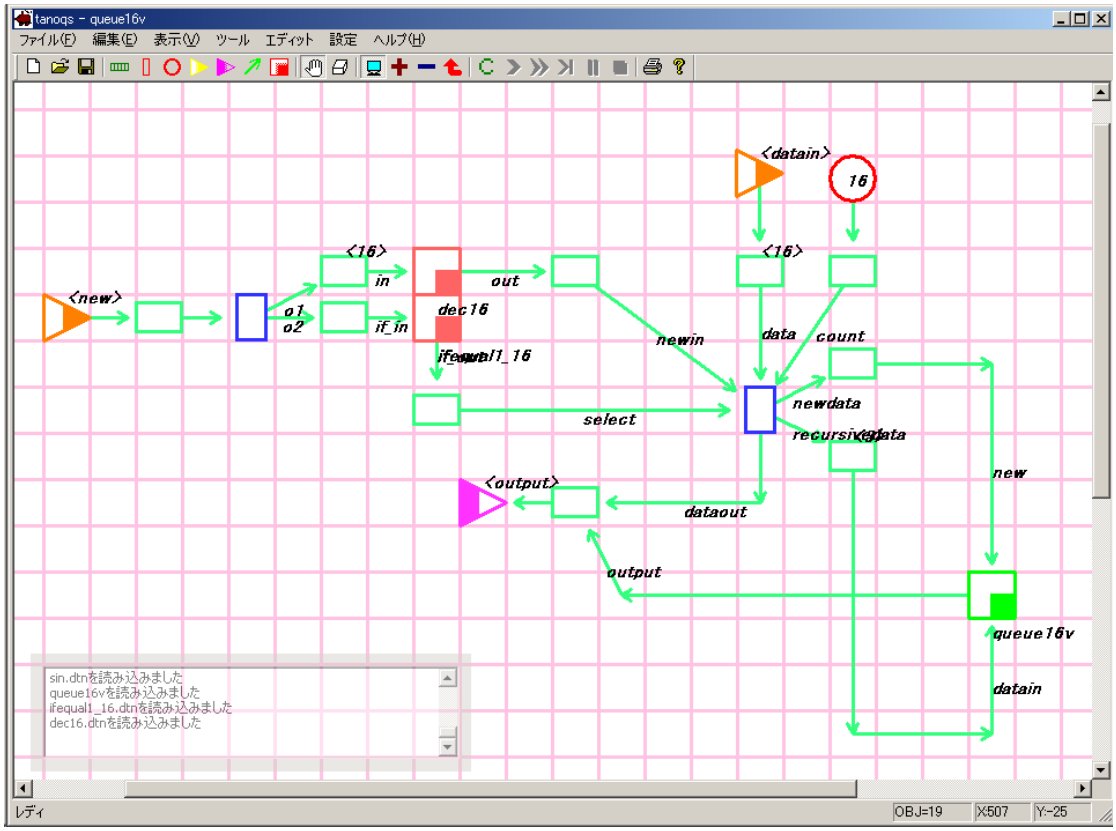


図 5.6　キュー (queue16v)

その後で，パラメータの値の個数分のデータをキューに流す．キューに入れ終わったら，その次のデータからは演算部 (図 5.7) に流す．演算部では，バタフライ演算が行われる．

実際には，下記の C 言語コード

```
wr = cos(theta * i);
```

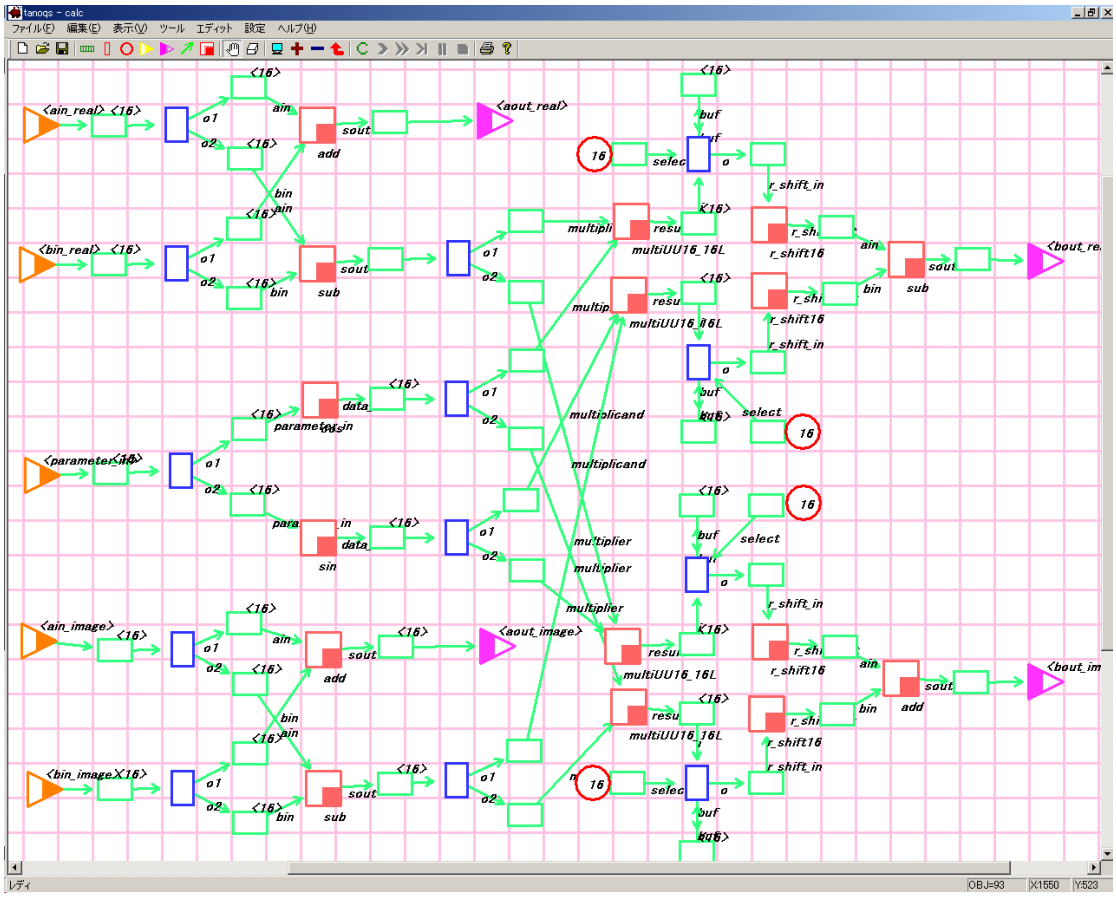
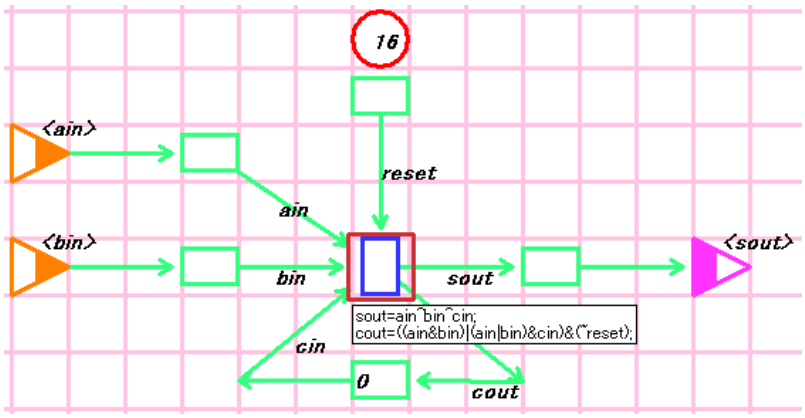


図 5.7　演算部 (calc)

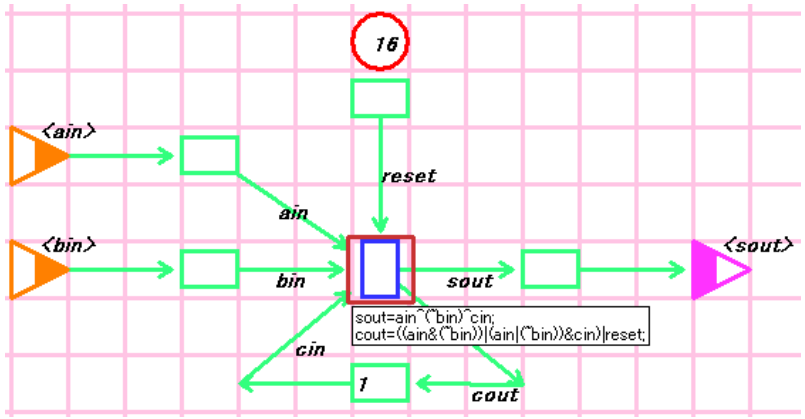
```
wi = sin(theta * i);
xr = ain_real  - bin_real;
xi = ain_image - bin_image;
ain_real  += bin_real;
ain_image += bin_image;
bin_real  = wr * xr - wi * xi;
bin_image = wr * xi + wi * xr;
```

と等価の演算を行う．cos や sin の計算を行うのは困難であるため，cos や sin の結果は参照用テーブルを配置し用いた．ここで用いたリセット機能付きの加算器，減算器と乗算器を図 5.8 に示す．

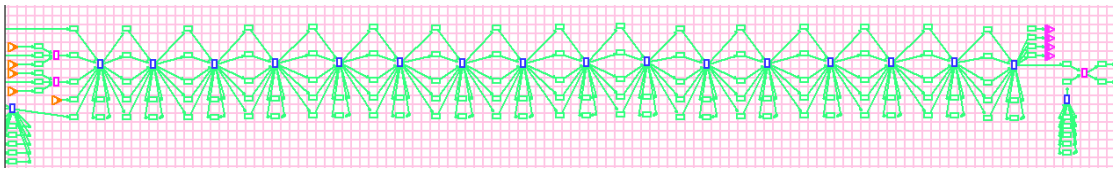
ifequal1_16 は，16 ビットのデータが 1，つまり 1000000000000000 であるとき 1 を，そうでないとき 0 を出力するモジュールである．r_shift16 は，16 ビットデータを 1 ビット右シフト (1/2 に) するモジュールである．最上位ビットには，入力データの 16 ビット目，つまり最上位ビット目が 0 なら 0，1 なら 1 を入れる．これは負の数を扱うためである．r_shift16 は FFT モジュールの中でも使用されているが，このときは入力の最上位ビットを 1 にしない限り，考慮する必要はない．gen16_0_16_0 は，入力された 1 ビットのデータ



(a) 16 ビット加算器 (add)



(b) 16 ビット減算器 (sub)

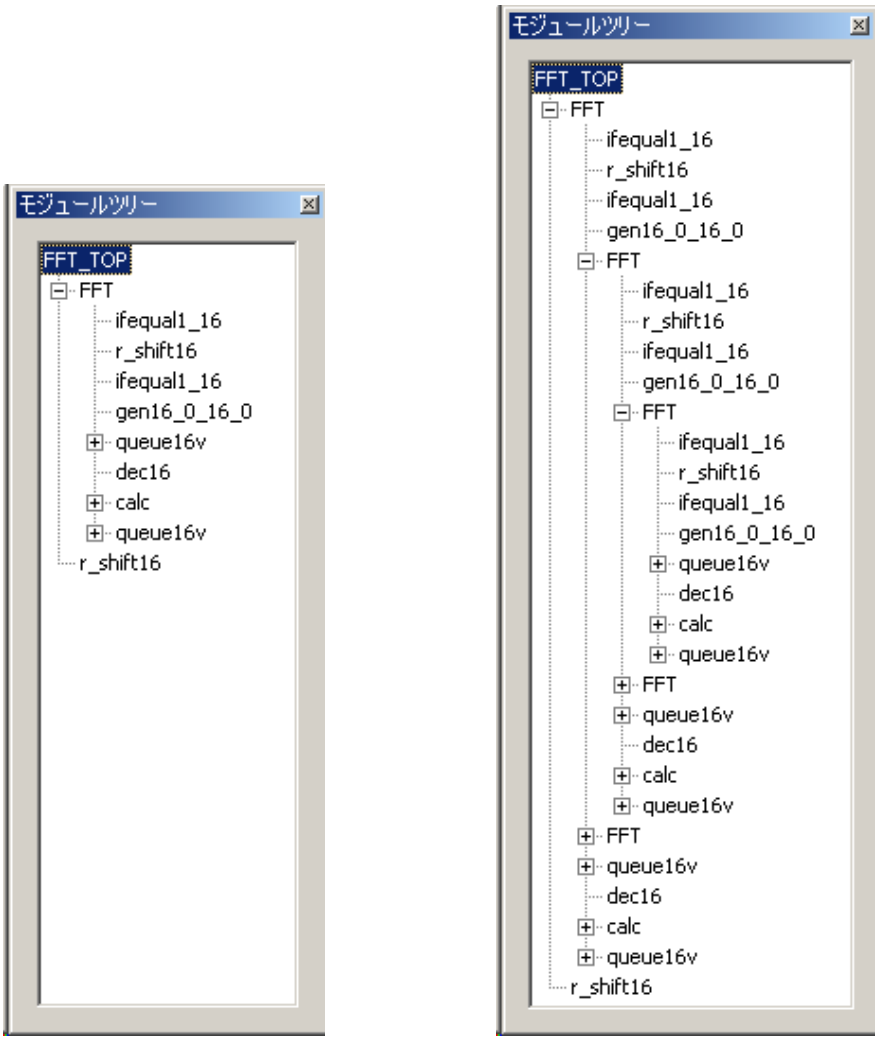


(c) 乗算器 (multiUU16_16L)

図 5.8　演算器

を 16 ビット作成し出力するモジュールである．dec16 は，16 ビットデータをデクリメントするモジュールである．

この FFT における動作時のモジュールは図 5.9(a) から図 5.9(b) のようになっており，動作時における再帰表現による回路の増殖が行われていることが確認できる．



(a) 設計時　　(b) 動作時

図 5.9　モジュール状態の変化 (サンプリング数 8 のとき)

5.3.2 動作確認

サンプリング数 8(000100000000000)，入力データが

```
data_real                    data_image
0000000000000000 ⇒ 0         0000000000000000 ⇒ 0
0100000000000000 ⇒ 1         0000000000000000 ⇒ 0
0010000000000000 ⇒ 2         0000000000000000 ⇒ 0
0110000000000000 ⇒ 3         0000000000000000 ⇒ 0
0001000000000000 ⇒ 4         0000000000000000 ⇒ 0
0101000000000000 ⇒ 5         0000000000000000 ⇒ 0
0011000000000000 ⇒ 6         0000000000000000 ⇒ 0
0111000000000000 ⇒ 7         0000000000000000 ⇒ 0
```

のとき，出力データは

```
dataout_real                 dataout_image
0001110000000000 ⇒ 28        0000000000000000 ⇒ 0
0001111111111111 ⇒ -4        0000111111111111 ⇒ -8
0001111111111111 ⇒ -4        0001111111111111 ⇒ -4
0001111111111111 ⇒ -4        0000000000000000 ⇒ 0
0001111111111111 ⇒ -4        0000000000000000 ⇒ 0
0001111111111111 ⇒ -4        0000000000000000 ⇒ 0
0001111111111111 ⇒ -4        0001000000000000 ⇒ 4
0001111111111111 ⇒ -4        0000100000000000 ⇒ 8
```

となった．今回作成した FFT は，実数部を 15 ビット，小数部を 1 ビットで表現しているため，少数部は 0.5 しか表現できない．そのため，cos と sin の値で $1/\sqrt{2}$ を用いるときはやむなく 0.5 に近似している．よって，出力データは本来の値とは異なるが，C 言語で記述した FFT プログラムにおいても同様の近似を行い計算した結果，tanoqs で作成した FFT と全く同じ結果が得られた．入力データをいくつか用意し，それぞれについて検証した結果も同様であった．

したがって，本研究において作成した FFT 回路は正しく動作するものであると確認できた．

第6章

結 論

本研究により，再帰構造を用いて回路を構成することにより，動作中に構成が変化する回路構造を有効に使えることを明らかにすることが出来た．これは，PCA の持つ動的再構成機能を有効に使えることを示しており，今後は他のアプリケーションへの適用について考えていく．

謝 辞

本研究の機会を与えてくださり，常にご指導いただいた長崎大学工学部情報システム工学科 小栗 清 教授に深く感謝します.

また，博士 2 年の永本 太一 さん，修士 2 年の 上田 真理江 さん，高野 智明 さん，修士 1 年の 内田 満 さん，休坂 慎也 さん，樋口 準人 さん，矢野 智史 さん，学部 4 年の 荒巻 徹 君，松林 秀典 君に厚く感謝します.

参考文献

[1] John von Neumann: “First Draft of a Report on the EDVAC(EDVAC に関する報告書第一稿)” (1945).

[2] John von Neumann: “電子計算機の論理設計の予備的討論” (1946).

[3] S. Trimberger: “A Reprogrammable Gate Array” Proceedings of the IEEE, Vol. 81, pp. 1030-1041(1993).

[4] FPGA Information Ltd. : “FPGA 入門” http://www.fpga.co.jp/nyumon2.html.

[5] S. B. Furber, P. Day: “Four-phase micropipeline latch control circuits” IEEE Transactions on VLSI Systems, 4(2): 247-253(June 1996).

[6] D. E. Muller, W. S. Bartky: “A theory of asynchronous circuits” in The Proceedings of an International Symposium on the Theory of Switching, Part1, pp. 204-243, Harvard University Press(April 1959).

[7] 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, pp181-186(1998).

[8] NEC Electronics Corporation: “動的再構成プロセッサ Dynamically Reconfigurable Processor (DRP)” http://www.necel.com/drp/ja/index.html.

[9] 小栗 清: “布線論理による新しい汎用情報処理アーキテクチャPCA1” bit, Vol. 32, No. 1, pp. 27-35 共立出版 (January 2000).

[10] 小栗 清: “布線論理による新しい汎用情報処理アーキテクチャPCA2” bit, Vol. 32, No. 3, pp. 54-62 共立出版 (March 2000).

[11] 小栗 清: “布線論理による新しい汎用情報処理アーキテクチャPCA 完” bit, Vol. 32, No. 7, pp. 51-59 共立出版 (July 2000).

[12] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi: “Plastic Cell Architecture: Towards Reconfigurable Computing for General-Purpose” Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp. 68–77 (1998).

[13] H. Ito, R. Konishi, H. Nakada, K. Oguri, A. Nagoya, N. Imlig, K. Nagami, T. Shiozawa, and M. Inamori: “Dynamically Reconfigurable Logic LSI — PCA-1” in Proc. 2001 Symposium on VLSI Circuits, pp. 103-106(Jun. 2001).

[14] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Shiozawa, M. Inamori, and K. Nagami: "PCA-1: A Fully Asynchronous, Self-Reconfigurable LSI" in Proc. ASYNC 2001, pp. 54-61(Mar. 2001).

[15] H. Ito, R. Konishi, H. Nakada, H. Tsuboi, Y. Okuyama, and A. Nagoya, "Dynamically Reconfigurable Logic LSI: PCA-2" IEICE Transactions on Information and Systems, Vol. E87-D, No. 8, pp. 2011-2020(Aug. 2004).

[16] H. Tsutsui, A. Tomita, S. Sugimoto, K. Sakai, T. Izumi, T. Onoe, and Y. Nakamura: "LUT-Array-Based PLD and Synthesis Approach Based on Sum of Generalized Complex Terms Expression" IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E84-A, No. 11, pp. 2681-2689(Nov. 2001).

[17] 三橋 渉: "信号処理" 培風館 (1999).

[18] 船越 満明: "キーポイント フーリエ解析" 岩波書店 (1997).

[19] J. Cooley, J. Tukey: "An Algirithm for the Machine Calculation of Complex Fourier Series" Mathematics of Computation, Vol. 19, pp. 297-301(1965).

[20] 太田 淳: "ペトリネット入門"
`http://www.aichi-pu.ac.jp/ist/~qua/intropn/intropn.html`(1995).

[21] "Prof. Dr. Carl Adam Petri"
http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri.html.

[22] 坂本 博和: "ビットシリアルペトリネットシミュレータに関する研究" 長崎大学工学部情報システム工学科卒業論文 (2002).

[23] 坂本 博和, 永本 太一, 柴田 裕一郎, 小栗 清: "非同期ビットシリアルペトリネットによるモデル化と設計ツールの実装" 第一回リコンフィギャラブルシステム研究会論文集 P285-292(2003).

[24] 坂本 博和, 永本 太一, 柴田 裕一郎, 小栗 清: "非同期ビットシリアル回路シミュレータ QROQS の開発" 信学論 (D-I), vol. J88-D-I, no. 2, pp. 155-162(2005).

[25] 坂本 博和: "QROQS Online Help"
http://crystal.freespace.jp/pgate1/QROQS/QROQS.html(2002-2003).

[26] 高野 智明: "ステートマシンとシフトレジスタを要素とする設計環境の検討" 長崎大学工学部情報システム工学科卒業論文 (2003).

[27] 高野 智明, 坂本 博和, 永本 太一, 柴田 裕一郎, 小栗 清: "ビットシリアル PCA シミュレータ tanoqs の開発" 第四回リコンフィギャラブルシステム研究会 論文集 P125-132(2004).

[28] 長川 大介: “動的自己再構成デバイスによる FFT の状況適応型構成手法の検討” 京都大学大学院情報学研究科修士論文 (2004).

[29] T. Nanya, A. Takamura, M. Kuwano, et al.: “TITAC-2: A 32-bit Scalable-Delay-Insensitive Microprocessor” Proceedings of HOT CHIPS IX, pp. 99–32, (1997).

[30] NTT Network Innovation Laboratories.: “PCA application design environment (PCASIM)”
http://www.onlab.ntt.co.jp/mn/pca/pcasim.html.

[31] 京都大学中村研究室: “PCACAD”
`http://easter.kuee.kyoto-u.ac.jp/~nemoto/pcacad/`.

[32] 上田 真理江: “MSB から演算を行うビットシリアル加算器・乗算器の研究” 長崎大学工学部情報システム工学科卒業論文 (2003).

[33] 川尻 健介: “非同期ビットシリアル除算器の開発” 長崎大学工学部情報システム工学科卒業論文 (2003).

[34] 中尾 宏一: “非同期ビットシリアル FFT の研究” 長崎大学工学部情報システム工学科卒業論文 (2002).

[35] 永本 太一, 坂本 博和, 柴田 裕一郎, 小栗 清: “低ビットレートボコーダ IMBE の固定小数点 DSP による実装” 信学論 (D-I), vol. J88-D-I, no. 2, pp. 344-352(Feb. 2005).