修　士　論　文

題　　目

# 動作中の回路追加・消去が可能な回路シミュレーション環境の開発

指導教官

小栗 清 教授

平成17年度

長崎大学大学院 生産科学研究科

電気情報工学専攻

髙野　智明（F04309）

## 論文要旨

フォンノイマン型コンピュータの発展の鍵はその汎用性と柔軟性にある．フォンノイマン型コンピュータはハードウェアとソフトウェアによって構成されており，ソフトウェアを変更する事によって汎用性を，メモリ上のデータ表現やオブジェクトインスタンスをmalloc や new 操作などによって動的に生成するヒープ管理によって柔軟性を得ている．その汎用性と柔軟性によってコンピュータは発展してきた．

PCA はその汎用性と柔軟性に着目して提案された新しいコンピューティングアーキテクチャである．フォンノイマン型コンピュータがプログラム理論をベースとするのに対し，PCA は布線論理をベースとし，動作中の回路が新たな回路を生成して協調動作する事が出来るという特徴がある．

現在，本研究室ではビットシリアルペトリネットシミュレータ QROQS を用いるなどして非同期ビット シリアル回路の研究は行われているが，PCA 最大の特徴である動的再構成機能を積極的に使ったアプリケーションの研究はあまり行われていない．その理由は，動的再構成機能を使うための malloc または new 操作の様な動的なインスタンスの生成方法が未だ提供されていないためであると考えられる．

このような問題を解決するため，本研究ではハードウェアにおける動的なインスタンスの生成を再帰表現を用いる事で実現する記述方法を考案し，記述に対する物理メカニズムを決めシミュレータの作成を行う．

動的に回路を生成して使用するという事を表現できる設計記述方式の検討に当たり，当初，空き領域を管理する回路が要求に応じて領域を払い出す malloc や new と同様の仕組みが使えるものと考えていた．しかし，払い出し要求がこの回路に集中してしまい布線論理の並列性を活かす事が出来ない，またハードウェアにはポインタ変数など存在しないので新しく配置したハードウェアと既存部分との接続関係を表現する為には新たな表現形式を考案しなければならないといった問題があった．

そこで，再帰表現が形式として領域の確保，インスタンスの配置，接続のすべてを持っている事に着目し，再帰表現をハードウェアの動的要素の表現として使えるのではないかと考えた．この再帰表現ではポインタを使用した接続関係の表現とは違い，新たに追加した回路をどのようにして既存部分に接続するかが明示されており，これにより必要に応じて回路を作り出して使用するという設計を直接的に表現できる様になった．

記述実験として FFT を再帰構造を用いて記述する実験を行い，再帰構造によって回路の生成，削除が行われる事を確認した．

今後は Bit Serial PCA へのマッピング，またより良いシミュレーション環境の構築を目指す．

# 目 次

# 図目次

# 表目次

# 第1章

# 緒 論

フォンノイマン型コンピュータの発展の鍵はその汎用性と柔軟性にある．フォンノイマン型コンピュータはハードウェアとソフトウェアによって構成されており，ソフトウェアを変更する事によって汎用性を，メモリ上のデータ表現やオブジェクトインスタンスを malloc や new 操作などによって動的に生成するヒープ管理によって柔軟性を得ている．その汎用性と柔軟性によってコンピュータは発展してきた．

PCA はその汎用性と柔軟性に着目して提案された新しいコンピューティングアーキテクチャである．フォンノイマン型コンピュータがプログラム理論をベースとするのに対し，PCA は布線論理をベースとし，動作中の回路が新たな回路を生成して協調動作する事が出来るという特徴がある．

現在，本研究室ではビットシリアルペトリネットシミュレータ QROQS を用いるなどして非同期ビットシリアル回路の研究は行われているが，PCA 最大の特徴である動的再構成機能を積極的に使ったアプリケーションの研究はあまり行われていない．その理由は，動的再構成機能を使うための malloc または new 操作の様な動的なインスタンスの生成方法が未だ提供されていないためであると考えられる．

このような問題を解決するため，本研究ではハードウェアにおける動的なインスタンスの生成を再帰表現を用いる事で実現する記述方法を考案し，記述に対する物理メカニズムを決めシミュレータの作成を行う．記述実験として FFT を取り上げ，実際に回路が増殖でき，動作する事を確認する．

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 章では現在使用されている動的要素の表現方法について，5 章では今回動的要素の表現方法として用いる事となった再帰表現について，6 章では回路増殖を表現するシミュレータの構成要素について，7 章では作成したシミュレータ tanoqs について，8 章では今回行った記述実験について，9 章では今後の課題について，最後に 10 章で本研究のまとめを述べる．

# 第2章

# 背 景

本章では本研究の背景となる PCA 及び非同期ビットシリアル回路について述べる．また，先に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 QROQS について述べ，関連研究についてのサーベイを行う．

## 2.1 PCA (Plastic Cell Architecture)

フォンノイマン型コンピュータの発展の鍵はその汎用性と柔軟性にある．フォンノイマン型コンピュータはハードウェアとソフトウェアによって構成されており，ソフトウェアを変更する事によって汎用性を，メモリ上のデータ表現やオブジェクトインスタンスを malloc や new 操作などによって動的に生成するヒープ管理によって柔軟性を得ている．その汎用性と柔軟性によってコンピュータは発展してきた．

PCA[1][2][3] はその汎用性と柔軟性に着目して提案された新しいコンピューティングアーキテクチャである．フォンノイマン型コンピュータがプログラム理論をベースとするのに対し，PCA は布線論理をベースとし，動作中の回路が新たな回路を生成して協調動作する事が出来るという特徴がある．

PCA 最初の LSI(PCA-1) は NTT により開発され，特徴的な機能のすべてが正しく動作する事が確認された．同グループは性能向上を目指した二つめの LSI である PCA-2 を開発した [4][5].

## 2.2 非同期ビットシリアル回路

非同期方式はグローバルクロックを用いず，ハンドシェイクを用いて局所的に通信を行い処理を進めていく．この方式は PCA の様に小さなシステムを組み込みながら大きなシステムを構成していく場合に向いている．同期方式では動作中のシステムに新しいシステムを追加する場合，全体のクロックが等長となる様にオブジェクトの位置を調整し，さらに動作開始時にオブジェクトのクロックをグローバルクロックに同期させる必要があるからである．

近年ビットパラレル方式の信号間スキューが問題となってきており，今後大きな性能制約要因になると言われている．ビットシリアル方式では単一の線でデータをやりとりするため一度に扱えるデータ量がパラレル方式に比べて少ないが，スキューの問題を考える必

要が無く，またパラレルに比べ配線が単純であるため回路を小さくできるという利点がある．さらに動的構成を行う際，ビットシリアル方式では接続が広がらないため接続部に設けられたスイッチによる性能低下が相対的に緩和される．

これらの理由により我々は非同期ビットシリアル回路が将来の動的再構成アーキテクチャの主流になると考えており，非同期ビットシリアル回路の研究を進めている．

## 2.3 非同期ビットシリアルシミュレータ

非同期パイプライン上でのビットシリアル演算回路の設計を行うためには信号遷移レベルで設計を行わなければならず，設計者の負担は大きかった．そのため，本研究室で非同期ビットシリアル回路をペトリネットでモデル化する方法が考えられた．

### 2.3.1 ペトリネットによる非同期回路のモデル化

非同期回路のモデリングにはペトリネットが用いられる事が多い．C 素子による非同期パイプラインをペトリネットで表現する場合，図 2.1 のように C 素子とトランジションを対応させる事が提案されている．これにより C 素子の入力線をそのトランジションの入力プレース，C 素子を反応させることの出来る入力値をトークンと対応付けることが出来る．入力プレースにトークンが揃ったときにトランジションは発火し，出力プレースにトークンを配置する．すると，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見ることが出来，これらの動作から非同期パイプラインの様子を観察することが出来る．

図 2.1 下部の様な非同期パイプラインを表現したペトリネットは，上部は処理されるべき情報を持ち，下部は処理するタイミングを制御していると見る事が出来る．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，発火可能時に出力プレースにトークンが配置されていないという条件を加える事で図 2.2 の様に簡略化する事が出来る．

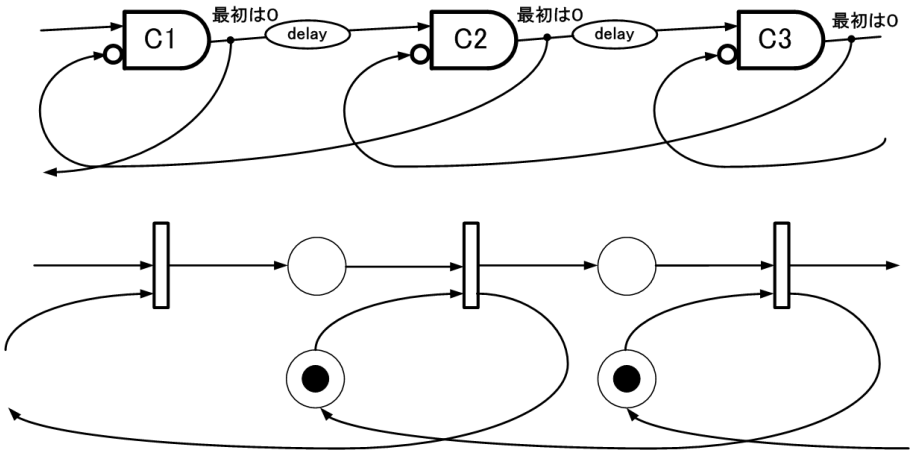



図 2.1: C 素子によるパイプラインをペトリネットでモデル化

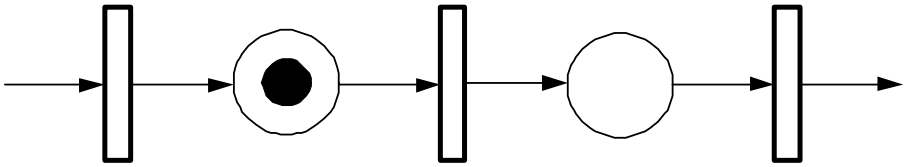


図 2.2: 簡略化したペトリネットによるパイプライン

### 2.3.2 ビットシリアルペトリネット

非同期ビットシリアル演算の設計とシミュレーションによる動作検証を効率よく行うためのモデルとしてペトリネットを用いたものがビットシリアルペトリネット (Bit Serial Petri nets) である．ビットシリアルペトリネットは黒トークンに値を設定しトランジションに関数を導入する事で，トランジションによる演算を黒トークンの値によって観察する事が出来る．この拡張はカラーペトリネットとして知られており，通常のペトリネットで表現できる．ビットシリアルペトリネットでは複数分岐，待ち合わせ合流，選択分岐，競合合流という要素を追加している．

1. 待合合流 (join)

   複数の入力を同時に必要とする演算はすべての入力が行われるまで演算を行うことが出来ない．このような場合において先に演算部に到着した処理要求が他の処理要求の到達を待ち合わせる (図 2.3(a))．

2. 競合合流 (conflict)

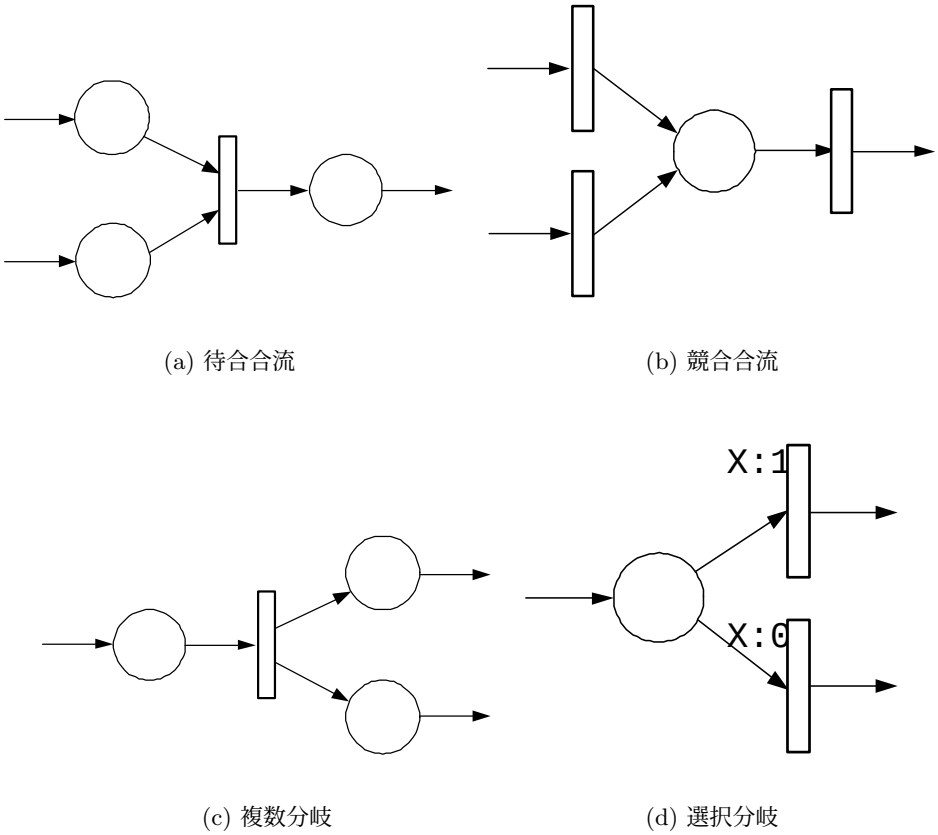
   一つの資源に対して複数の処理要求が出された場合に排他制御が必要となる．ここでガード式などにより処理要求に対していずれの処理要求を優先するか記述されていない場合，処理要求の到達の早いものが優先される (図 2.3(b))．

3. 複数分岐 (fork)

   あるパイプラインの要求信号を複数のパイプラインに伝達する，あるいは同じ情報から複数の演算結果を得る場合にパイプラインを分岐させる (図 2.3(c))．

4. 選択分岐 (select)

   パイプラインが複数に分岐し，そのいずれかに要求信号を伝達する場合が選択分岐である．通常のペトリネットではこのように共有する入力プレースに配置されたトークンをそのトランジションが先に発火するかということは規定されていない．そのためビットシリアルペトリネットではガード式を導入し，発火するトランジションを決定する．ガード式はトランジション関数同様入力プレースの保持する値を元に論理演算を行い，真であるトランジションが発火する．この際，ガード式は唯一つのトランジションが発火するよう記述されなければならない (図 2.3(d))．




(a) 待合合流 (b) 競合合流

(c) 複数分岐 (d) 選択分岐

図 2.3: ビットシリアルペトリネット

## 2.4 QROQS

これらビットシリアルペトリネットをドロー系 GUI によって各プリミティブの配置，接続，編集を行うツールが QROQS(図 2.4) である．QROQS はビットシリアルペトリネットシミュレーションにより，視覚的にビットシリアル演算の動作検証を行う事が出来る．他の機能として BSP から Verilog-HDL ファイルを出力することで遅延時間を含めたより詳細なシミュレーションを行うことが可能である．また，C 言語のソースファイルを出力することにより，より高速なシミュレーションを行うことが出来る [6].

## 2.5 関連研究

PCA 用のシミュレーションツールとしては NTT 未来ねっと研究所で開発された PCASIM[7] や京都大学中村研究室で開発された PCACAD がある．非同期ビットシリアル回路シミュレータには前述した QROQS がある．しかしながら QROQS には PCA の最大の特徴である動的再構成機能を持っていない．また，動的再構成機能を積極的に使ったアプリケーションの研究はあまり活発に行われていない [8].

PCA の研究として，NTT では PCA-2 が開発された．京都大学では PCA-Chip2 とい

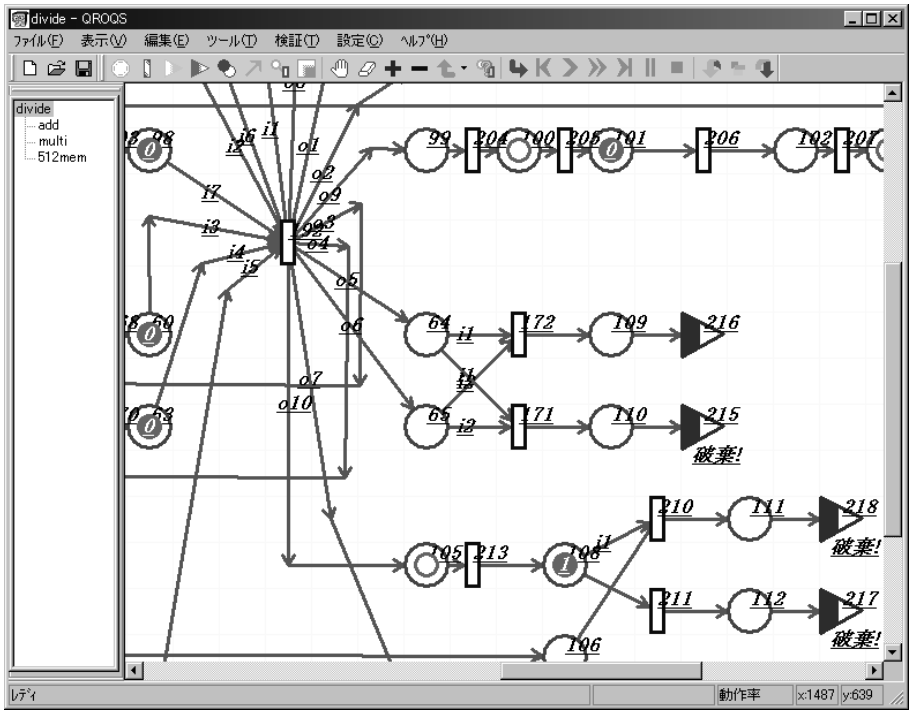



図 2.4: QROQS

う同期回路を用いた PCA の試作デバイスが開発された．他の再構成アーキテクチャとして NEC エレクトロニクスの DRP(Dynamically Reconfigurable Processor) や IPFlex の DAPDNA などがある．

# 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

現在，QROQS を用いるなどして非同期ビットシリアル回路の研究は行われているが，PCA 最大の特徴である動的再構成機能を積極的に使ったアプリケーションの研究はあまり行われていない．その理由は，動的再構成機能を使うための malloc または new 操作の様な動的なインスタンスの生成方法が未だ提供されていないためであると考えられる．

## 3.2 本研究の目的

このような問題を解決するため，本研究ではハードウェアにおける動的なインスタンスの生成を，再帰表現を用いる事で実現する記述方法を考案し，記述に対する物理メカニズムを決めシミュレータの作成を行う．

# 第4章

# 動的要素の表現形式

この章では現在使用されている動的要素の表現法について述べる．

## 4.1 ソフトウェア

ソフトウェアにおいてリソースを動的に確保して使用しようとする場合，幾つかの方法がある．

### 4.1.1 malloc や new

C 言語においては malloc，C++言語では new を使用する事によってプログラムの動作中に領域を確保する事が出来る．malloc は確保する領域のサイズを指定する事によりメモリ領域の確保を行い，new ではインスタンスを指定する事によりメモリ領域の確保，インスタンスの配置を行う．new，malloc を使用して作りだした構造体やクラスのインスタンスなどは，ポインタ変数にアドレスを代入する事によって既存部分と接続する．動的要素の使用に関しては，領域確保，配置，接続が基本要素であると思われるが，ポインタ変数に対する値の代入は一般の変数に対する値の代入と同じ扱いであり，動的要素の接続という形式が特別設けられているわけではない．

### 4.1.2 再帰表現

再帰表現を用いる事でも再帰呼び出しする関数のインスタンス (ローカル変数など) を動的に生成する事が出来る．再帰表現はスタックを用いる事で領域を確保し処理を行う．再帰表現では，再帰呼び出しを行う関数を単位として，その入出力関係，すなわち接続関係が記述されており，malloc，new に比べると動的要素の領域確保，配置，接続のすべてが形式として用意されていると考える事が出来る．ただしソフトウェアの再帰では処理中の関数は 1 つのみで，関数の処理が完了すればその関数のインスタンスは消されてしまうので再帰により作られた構造そのものを繰り返し使用するという事はない．

### 4.1.3　fork や thread 実行

fork によるプロセスの複製や thread の実行によっても動的に実行インスタンスを増やす事が出来る.

## 4.2　ハードウェア

上述の通り，ソフトウェアでの動的要素の表現は各種存在するが，ハードウェア記述言語でハードウェアを対象として動的要素の表現形式を用意したものは現在のところ存在しない.

# 第5章

# 再帰構造による回路増殖

動的に回路を生成して使用するという事を表現できる設計記述方式の検討に当たり，当初，空き領域を管理する回路が要求に応じて領域を払い出す malloc や new と同様の仕組みが使えるものと考えていた．しかし，払い出し要求がこの回路に集中してしまい布線論理の並列性を活かす事が出来ない，またハードウェアにはポインタ変数など存在しないので新しく配置したハードウェアと既存部分との接続関係を表現する為には新たな表現形式を考案しなければならないといった問題があった．

そこで，再帰表現が形式として領域の確保，インスタンスの配置，接続のすべてを持っている事に着目し，再帰表現をハードウェアの動的要素の表現として使えるのではないかと考える様になった．ただし，ソフトウェアの様な関数を単位とする再帰表現で，関数の呼び出しでインスタンスが生成され，関数の終了でインスタンスが削除されるというのではなく，回路の部品種を単位とする再帰表現で，特別な信号 new によってインスタンスが生成され，特別な信号 delete でインスタンスが削除されるという，再帰表現をベースとして new，delete の機能を加える事とした．例えば回路種 A の定義の中に部品として回路種 A が使われることを許し，回路種 A は特別な入力端子 new と delete を持つという様にする．この再帰表現ではポインタを使用した接続関係の表現とは違い，新たに追加した回路をどのようにして既存部分に接続するかが明示されており，これにより必要に応じて回路を作り出して使用するという設計を直接的に表現できる様になった．以下，この再帰表現による回路の動的割り付けを回路増殖と言う．

# 第 6 章

# 回路増殖を表現するシミュレータ

第 2 章で述べた様に

- 回路を動的に構成する場合のタイミング制御の簡単化
- 接続が広がらず，動的構成のために接続部に設けられたスイッチによる性能低下が相対的に緩和される
- 演算回路を小さくできるため，データをメモリと演算回路の間を何度も往復させるのではなく，処理のままに配置されたたくさんの演算回路の中を流す事によって性能向上を達成できる

といった理由のために我々は非同期ビットシリアル回路が将来の動的再構成アーキテクチャの主流になると考えている．このため再帰表現による回路増殖をどのように行うかも，この非同期ビットシリアル回路を前提に行う事とした．

これまで，非同期ビットシリアル回路の設計，動作検証を目的に非同期ビットシリアルシミュレータ QROQS を開発してきた．しかし，設計する非同期ビットシリアル回路が大規模化するにつれ，QROQS での設計，シミュレーションは時間が掛かるという問題が出てきた．そこで，より抽象度を上げることを目指した新しいシミュレータ tanoqs を卒業論文時に開発した．

今回，tanoqs に再帰表現による回路増殖機能を追加し，tanoqs 内で再帰表現を実現する．

この章では tanoqs の構成要素について述べる．

## 6.1 非同期ビットシリアル回路で用いられる要素

これまでに QROQS により非同期ビットシリアル方式で設計された音声の圧縮方式である IMBE 用のデコード回路などを調査したところ典型的な回路パターンが数多く出現する事が分かった [9][10]．それらの回路パターンとはステートマシン，シフトレジスタ，カウンタである．これらの事により，我々はステートマシンとシフトレジスタで構成される Bit Serial PCA の研究を行っている．

### 6.1.1　QROQS でのステートマシン表現

図 6.1 が QROQS でのステートマシンの例である．この時のトランジション関数は表 6.1 になる．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事が出来ず，状態ごとに違った処理を行おうとすると関数が複雑となってしまう．

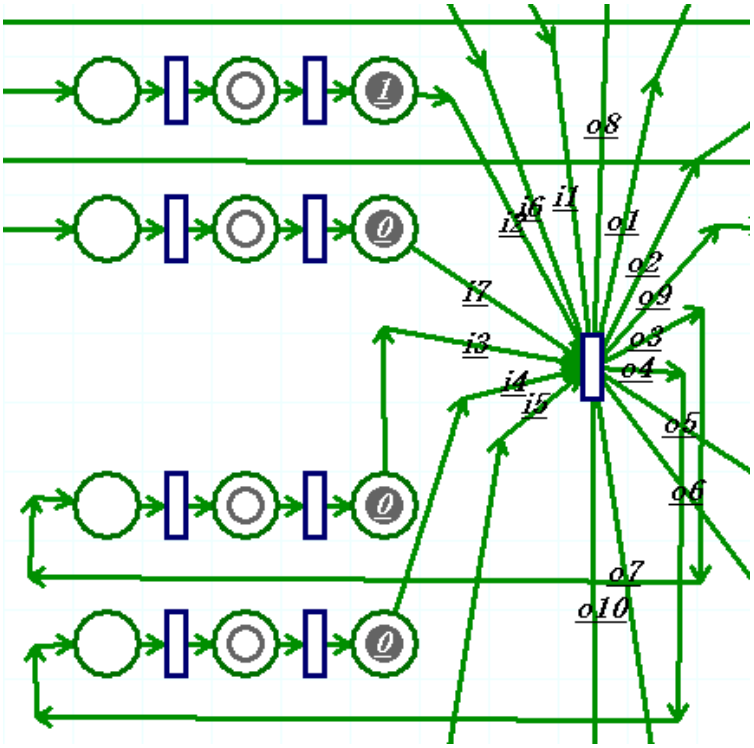



図 6.1: QROQS でのステートマシン

表 6.1: 図 6.1 での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;
```

### 6.1.2　QROQS でのシフトレジスタ表現

図 6.2 に QROQS でのシフトレジスタの例を挙げる．QROQS では入力をそのまま出力するパ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．

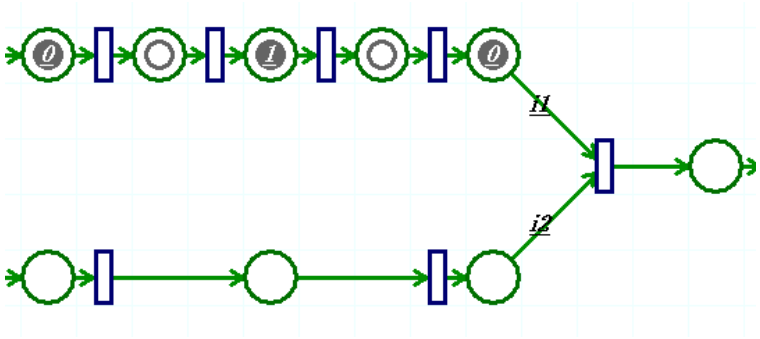



図 6.2: QROQS によるシフトレジスタの表現

### 6.1.3　QROQS でのカウンタ表現

図 6.3 に QROQS におけるカウンタ表現，表 6.2 にこの時のトランジション関数を示す．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 となった時に start に 1 を出力する．それ以外の時は 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，ビットの区切りを表すためにカウンタは必須であり多用される．

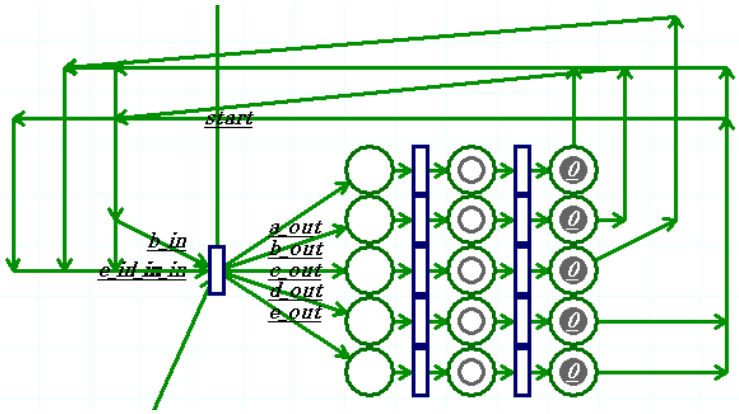



図 6.3: QROQS によるカウンタの表現

表 6.2: 図 6.3 での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^(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;
```

## 6.2　抽象度の向上

### 6.2.1　処理の合流，分岐

IMBE の様な複雑なアルゴリズムは処理の合流，分岐によって実現される．非同期回路における処理の合流，分岐は以下の 4 通りとなる．

1. 待合合流 (join)

   図 6.4(a) が待合合流である．入力が複数必要な処理を行う場合，すべての入力が揃ってから処理が行われる．幾つの入力を待ち合わせるかは状況によって変化する．

2. 競合合流 (conflict)

   図 6.4(b) が競合合流である．一つの資源に対して複数の処理要求が出された場合に調停が必要となる．

3. 複数分岐 (fork)

   図 6.4(c) が複数分岐である．複数の結果を出力する場合である．幾つ出力を行うかは状況による．

4. 選択分岐 (select)

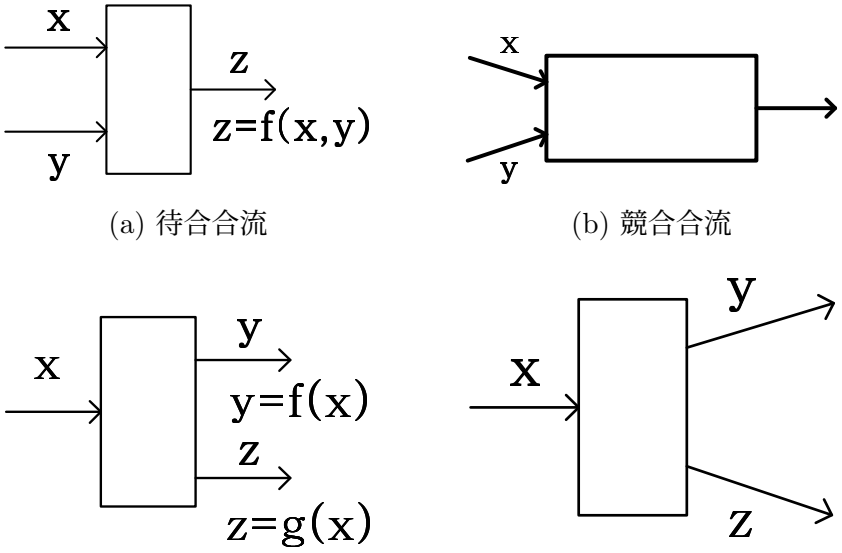
   図 6.4(d) が選択分岐である．いずれかに要求を出力する場合である．

### 6.2.2　プリミティブの定義

非同期ビットシリアル回路で使用される典型的なパターンを構成プリミティブとして用いる事で抽象度の向上を図る．用いるプリミティブはステートマシン，シフトレジスタ，カウンタである．これらと合流，分岐機能を結合する事により，tanoqs のプリミティブを以下の様に決定した．

#### ステートマシン

ステートマシンは複数の状態を保持する事が出来，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事によりステートマシン全体の機能を表現




(a) 待合合流　　(b) 競合合流

(c) 複数分岐　　(d) 選択分岐

図 6.4: 処理の合流，分岐

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図 6.4(a) 待合合流のどれだけの入力を待ち合わせるか，また図 6.4(c) の複数分岐あるいは図 6.4(d) の選択分岐を表現できる．ステートマシンは QROQS で記述形式として採用したペトリネットで表現すると，待ち合わせと複数分岐を表現するために図 6.5 のようになり，入出力ともペトリネットのトランジションからの信号となる．従って QROQS と同様の方法で非同期回路へマッピングする事を前提とすると，ペトリネットにおいてトランジションとトランジションは接続できないのでステートマシン同士の接続は禁止されることとなる．

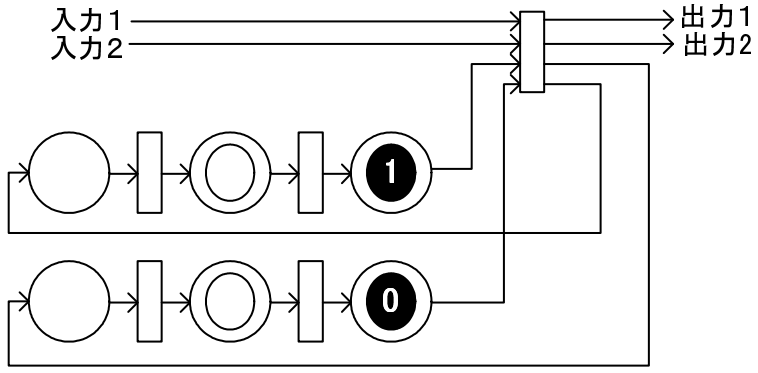



図 6.5: ステートマシンのペトリネット表現

#### シフトレジスタ

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事により記憶ビット数だけでなく遅れ時間数も定義する．また，シフトレジスタは通常 1 入力 1 出力であるが，複数の入力を繋ぐ事により競合合流を表現することとした．シフトレジスタはペトリネットでは競合合流を表現するために図 6.6 のようになり，入出力はペトリネットのプレースからのものとなる．従ってシフトレジスタ同士は接続できず，シフトレジスタはステートマシン，カウンタ，ソーストランジション，シンクトランジション，モジュールに接続される．また，シフトレジスタはあらかじめ値を入力しておく事も出来る様にした．

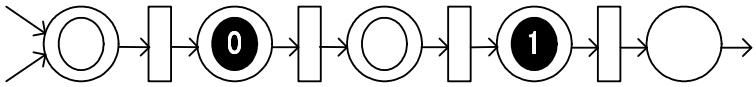



図 6.6: シフトレジスタのペトリネット表現

#### カウンタ

カウンタはカウント数を設定して使用するプリミティブである．カウント数は初期状態ではカウンタ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 となる時は 1 を出力すると共にカウント数をカウンタ変数にコピーする．カウンタはペトリネットでは入力 1，出力 1 のステートマシンと同様になる．従ってカウンタの入出力はシフトレジスタと接続される．カウンタはペトリネットでは図 6.5 のようにステートマシンと同様になる．カウンタをステートマシンに含めてしまう事も可能であったが，別にカウンタを用意する方が望ましいとここでは判断した．

### 6.2.3　シミュレータで用いられる要素

設計，シミュレーション時に用いられる要素として入出力とモジュールがある．

#### 入出力

シミュレーションを行う際のデータの入力プリミティブとしてソーストランジション，出力プリミティブとしてシンクトランジションを設けた．ソーストランジションからデータを入力し，シンクトランジションへデータを出力する．これらはシフトレジスタに接続される．また，これらは後述するモジュールの入出力としても使用される．

### 6.2.4　モジュールと階層構造

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事が多々ある．そのため，管理と構築の容易化，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を行える様にした．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事によってモジュール化と階層構造を表現できるようにした．また，この階層構造は後述する再帰構造にも使用される．下位モジュールでは，上位モジュールからの入力はソーストランジションからの入力に見え，上位モジュールへの出力はシンクトランジションへの出力に見える．従ってモジュールの入出力はシフトレジスタに接続されなければならない．再帰を行わない場合，モジュールとの接続を行うソーストランジション，シンクトランジション共に無視され，ソーストランジション，シンクトランジションに接続された各々のシフトレジスタは連結して扱われる．階層構造の様子を図 6.7 に示す．図 6.7 上部のモジュール

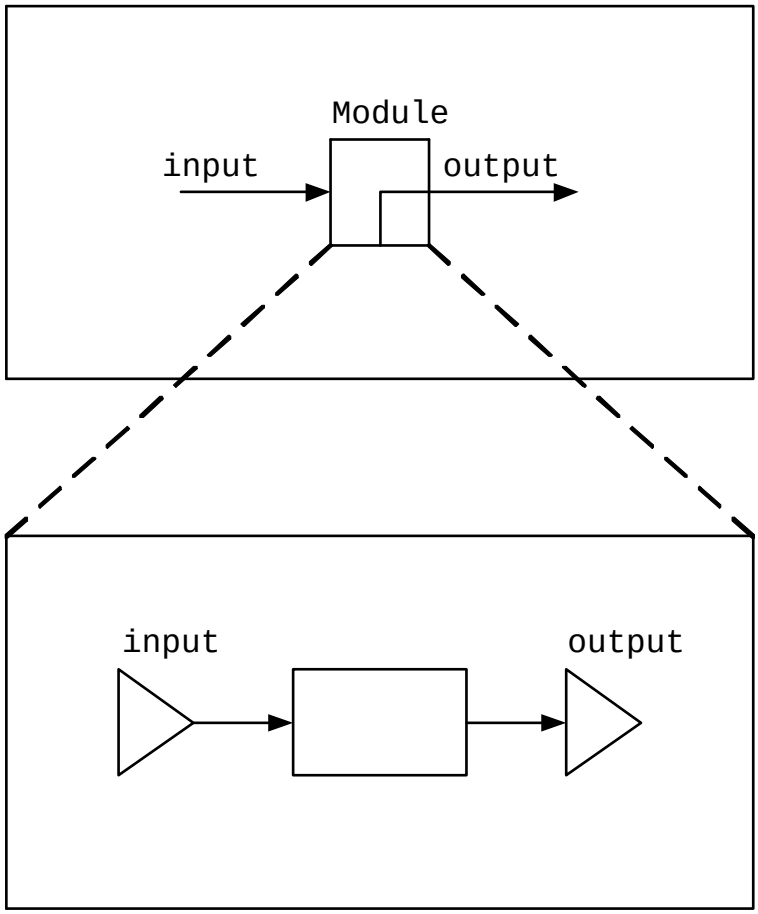



図 6.7: モジュールと階層構造

に入力が在った場合，図 6.7 下部の input からデータが入力される．図 6.7 下部の output にデータが入った場合，図 6.7 上部の output からデータが出力される．この時の様子を

ペトリネットで表現すると図 6.8 の様になる．

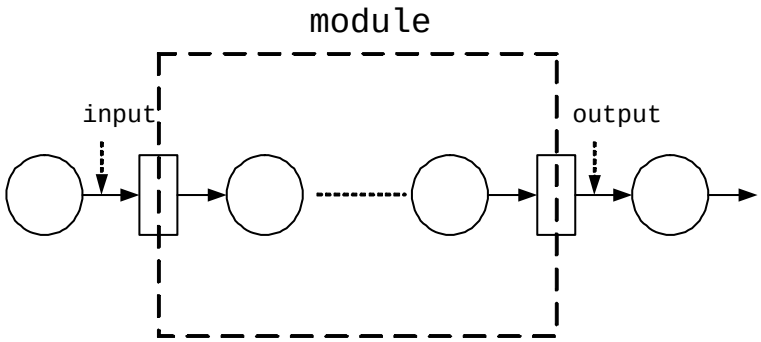



図 6.8: 階層構造のペトリネット表現

## 6.3　再帰構造

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事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事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事により任意個のモジュールを作成，削除できる．new や delete という端子の性質を設けただけでは任意個のモジュールを作り出し削除する事は出来ない事に注意されたい．再帰表現と組み合わせる事によってはじめて任意個のモジュールの作成，削除が可能となる．

図 6.9(a) に再帰表現を持つ回路の例を示す．図 6.9(a) 上段は TOP モジュール，中央は A モジュール．下段は B モジュールであり，A モジュールは new 端子と delete 端子を持っている．この回路が動作した時の回路構成の変化を図 6.9(b) に示す．初期状態では TOP モジュールのみの構成となる．TOP モジュール内の A モジュールの new 端子に要求が到着した時に回路の作成が行われ，回路構成は図 6.9(b) 中央の様になる．B モジュールは new 端子を持たないためそのまま展開される．図 6.9(b) 中央の B モジュール内の A モジュールは new 端子を持つため展開されない．図 6.9(b) 中央の B モジュール内の A モジュールの new 端子に要求が到着した場合さらに回路が作成され，図 6.9(b) の右の様になる．図 6.9(b) の右に示す回路構成で，上から 3 番目の B モジュールに含まれる A モジュールの delete 端子に要求が到着するところの A モジュールが削除され図 6.9(b) の中央の様になる．さらに TOP に含まれる A モジュールの delete 端子に要求が到着するとこの A モジュールが削除され図 6.9(b) 左のトップモジュールだけの回路構成となる．もし図 6.9(b) の右の回路構成で TOP 直下の A モジュールの delete 端子に要求が到着すると，この A モジュールは削除されるがその中に含まれていた A モジュールはどこにも接続されない状態で残ってしまう．new と delete は本来独立であるが delete は new を持った場合に限って持つ事が出来る事とした．

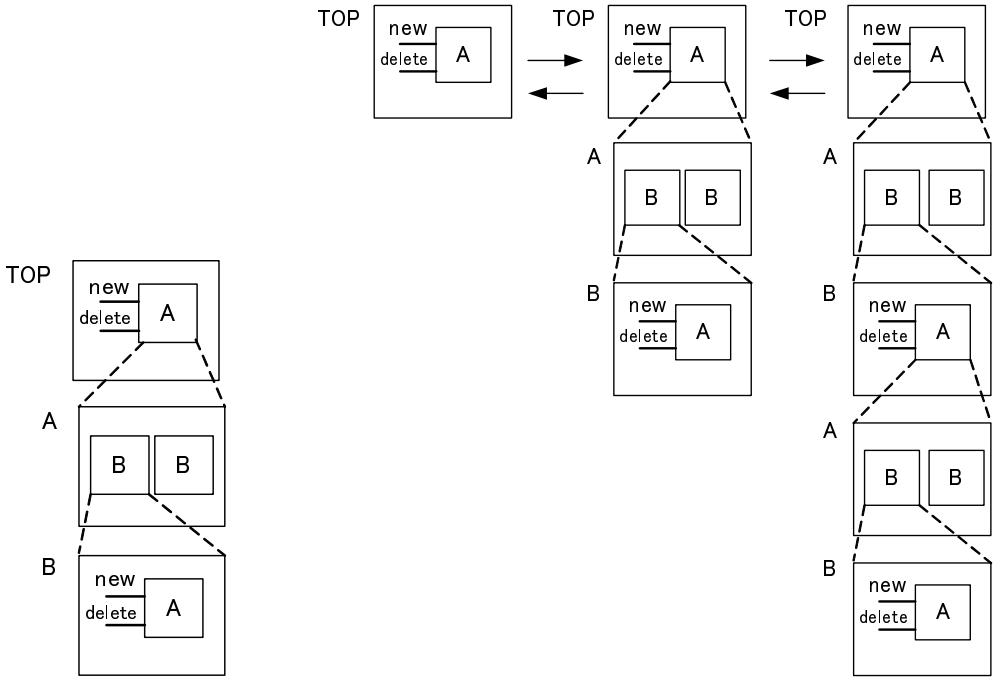



図 6.9: 再帰表現の例

### 6.3.1　再帰構造の回路構造

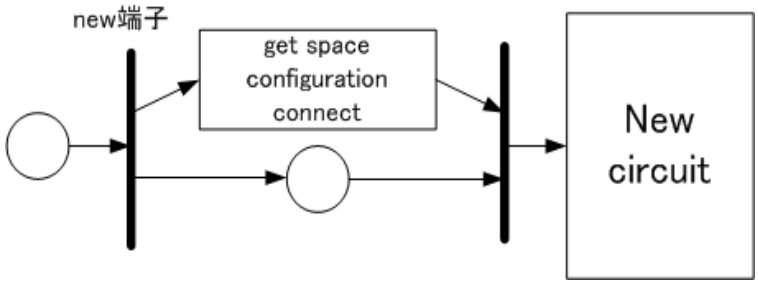



図 6.10: new 端子

new 端子を持つモジュールを直下に含むモジュールは new 端子を持つモジュールの回路情報を持ち，その回路情報を元に組み込み部に対して必要な空きスペースを作るためのコマンド，回路を構成するためのコマンド，構成した回路の端子を自モジュールに接続するためのコマンドを送出する回路を持つ．この回路は new 端子に要求が来た時に動作し，要求そのものは図 6.10 の様に new 端子を持つモジュールが完成し無事に接続されるのを待って，このモジュール内に入っていく．

delete 端子を持つモジュール，従って new 端子も持つモジュールを直下に含むモジュールは new の動作を行う回路に加えて delete を行う回路を持っており，new の時の情報を

元に回路の削除と接続の解放を行う.

### 6.3.2　アーキテクチャに要求される機能

任意の回路を構成できる可変部が，コマンドを送りつける事によって通信や構成などのあらかじめ決まったより上位の機能を持つ組み込み部を使う事が出来るという点が PCA の本質である．従って再帰構造と new 端子や delete 端子による回路増殖や縮小を可能とするには組み込み部の機能として

- 空き領域の確保
- 回路の構成
- 回路の接続
- 回路の削除
- 接続の解放

があれば良いといえる.

# 第7章

# シミュレータtanoqs

第 6 章で述べたプリミティブを用い，非同期ビットシリアル回路の設計，シミュレーションを行うシミュレータが tanoqs である．この章では tanoqs について述べる．

## 7.1 tanoqs

tanoqs はステートマシンとシフトレジスタ，カウンタを要素として非同期ビットシリアル回路の設計，シミュレーションを行うシミュレータである．図 7.1 に tanoqs の画面を示す．

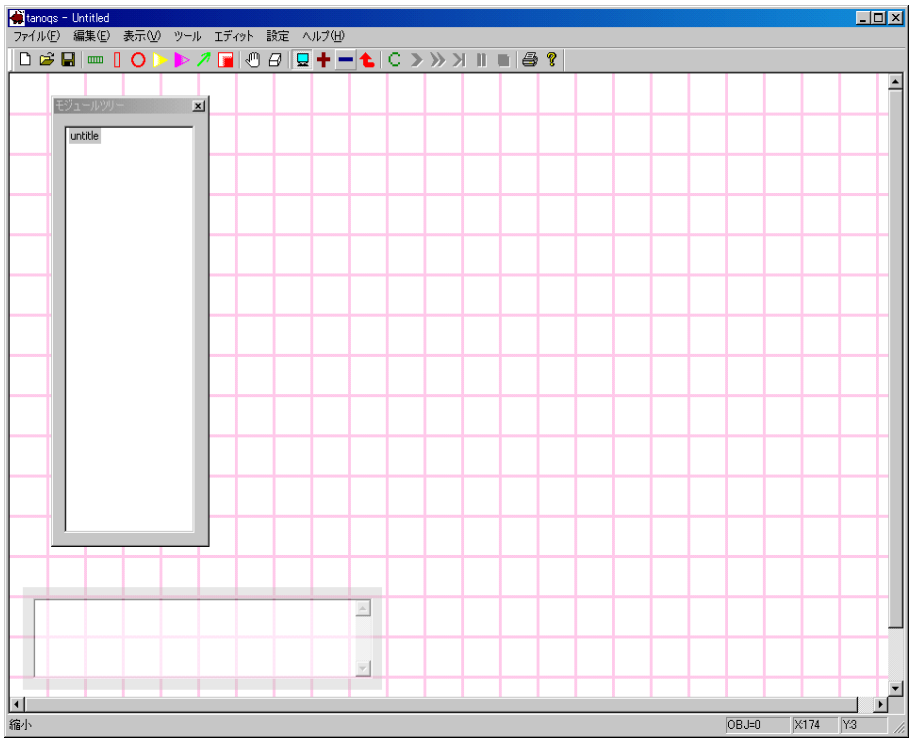



図 7.1: tanoqs

tanoqs は Microsoft Visual C++を用いて制作されており，マウスによりプリミティブ

の配置，接続，編集が行える．また，処理の流れを視覚的に見る事が出来る．

画面左に見えているダイアログはツリーダイアログである．モジュールによる階層構造を表示する．また，シミュレーション時には動作中に作成されたモジュールも同時に表示する．ツリーダイアログのモジュールをクリックする事でそのモジュールに移動する事が出来る．

画面左下に表示されているダイアログはアウトプットダイアログである．シミュレーションの開始，終了，エラー情報などが表示される．

### 7.1.1　ファイルツール

シミュレーションで使用するデータファイルを扱うメニューである．データはテキスト形式で，拡張子.dtn で保存される．

- 新規作成 

  今までのデータを破棄し，新しく設計を始める．現在表示されているモジュールだけではなく，すべてのモジュールデータを破棄する．

- ファイルを開く 

  保存していたシミュレーションデータを読み込む．すべてのモジュールデータを破棄し，保存していたデータの状態を復元する．

- ファイルに保存 

  作成したシミュレーションデータをファイルに保存する．新規作成した場合は，新たなファイルにデータを保存するが，ファイルを開いた状態からの保存はファイルを上書きして保存する．

### 7.1.2　プリミティブツール

メニューよりステートマシン，シフトレジスタ，カウンタ，ソーストランジション，シンクトランジション，モジュール，アーク，7 つのプリミティブを選択してフィールドに配置する．

- ステートマシン 

  ステートマシンを配置する．配置後，ダブルクリックすることでエディットウインドウを開きステートマシンの処理を入力することができる．

- シフトレジスタ 

シフトレジスタを配置する．配置後，ダブルクリックすることでエディットウインドウを開き保持する値，ビット数を入力することができる．

- カウンタ 

  カウンタを配置する．配置後，ダブルクリックすることでエディットウインドウが開きカウンタの設定を行うことができる．

- ソーストランジション 

  ソーストランジションを配置する．配置後，ダブルクリックすることでエディットウインドウを開きビット列を入力することができる．

- シンクトランジション 

  シンクトランジションを配置する．配置後，ダブルクリックすることでエディットボックスを開き，シンクトランジションの設定を行うことができる．

- モジュール 

  モジュールを配置する．配置後，ダブルクリックする事でモジュール名を記入し，新たに回路を構築する事が出来る．また，モジュール読み込みを行う事で既に作成したモジュールを読み込む事が出来る．また，名前設定後はダブルクリックする事でそのモジュールに移動する．

- アーク 

  プリミティブ同士をつなぐアークを配置する．マウスで接続元，接続先の順にクリックすることでプリミティブ同士をつなぐ．また，接続先までのアークは，フィールドをクリックすることで中継点を設定することができる．接続中右クリックすることでアークの接続を中止することができる．

### 7.1.3　エディットツール

配置したプリミティブの移動，消去を行うためのツールである．

- ハンド 

  配置されているプリミティブをドラッグ&ドロップで移動させる．また，複数のオブジェクトの選択も行う．

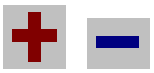
- 削除 

配置されたプリミティブを削除する．削除されたプリミティブにつながっているアークも同時に削除する．モジュールを削除した場合は，そのモジュール以下のモジュールも削除する．

### 7.1.4　その他ツール

- アウトプット 

  アウトプットダイアログの表示非表示を切り換える．

- スケーリング 

  画面の拡大，縮小を行う．

- 上位モジュールへ移動 

  自モジュールよりも上位にモジュールがある場合，そのモジュールに移動する．

### 7.1.5　シミュレーションツール

シミュレーションを行うためには，コンパイルによってシミュレーションデータのチェックを行わなければならない．コンパイルが成功すればシミュレーションモードになりシミュレーションを行う．シミュレーションモードではプリミティブの編集は行えない．

- コンパイル 

  コンパイルを行う．エラーがあった場合，エラーはメッセージダイアログに表示され，エラーを起こしたプリミティブは赤く囲まれる．

- 連続実行 

  シミュレーションを実行する．ソーストランジションに設定された値がペトリネットに入力され，出力をシンクトランジションが記録する．実行の際にはステートマシンの発火などアニメーションを行う．すべてのオブジェクトの動作が停止するまでシミュレーションは実行される．

- アニメーション無し実行 

  アニメーションを行わずシミュレーションを実行する．無限ループに陥らない限りすべてのオブジェクトの動作が停止した時点でシミュレーションは終了する．

- ステップ実行 

  すべてのオブジェクトの処理を 1 ステップだけ行う．

- 一時停止 

  連続実行中のシミュレーションを停止する．一時停止中のシミュレーションは連続実行，アニメーション無し実行，ステップ実行のいずれかで再開することができる．

- シミュレーションの停止 

  シミュレーションを停止し，エディットモードに戻る．

## 7.2　tanoqs 設定

使用者が自由に設定できる項目は以下の通りである．なお，設定ファイルは tanoqs.ini というファイル名で実行ファイルと同じフォルダに作られる．

- エディット幅

  フィールド上に表示されるグリッド幅の変更．デフォルトでは 40.

- フォント

  フィールド上で表示される文字のフォントの設定．デフォルトでは MS P ゴシック

- 各プリミティブの表示色

  各プリミティブ及び，背景，グリッドの表示色の変更を行える．

- アニメーション実行時のシミュレーション速度

  シミュレーションを行う際のアニメーションの実行速度を変更できる．デフォルトでは 50ms．マシンの性能によって左右されるため，必ずしも設定した速度でシミュレーションが行われるわけではない．

- シフトレジスタのデータ表示方向

  シフトレジスタのデータ表示方向を設定できる．デフォルトでは自動設定になっており，シフトレジスタから出力されるアークの方向によってデータの表示方向が決定される．

## 7.3　その他シミュレータの機能

- フィールドの拡大縮小，移動

  マウスホイールによってフィールドの拡大，縮小，移動が行える．ホイールクリックをした状態でドラッグする事によってフィールドの表示位置が変更される．

- Undo 機能

  Ctrl+Z で Undo が行える．

- 整列

  オブジェクトをグリッドに合わせて整列する．また Ctrl を押した状態オブジェクトを配置する事でグリッドに合わせてオブジェクトが配置される．

- コピー，カット，ペースト

  オブジェクトのコピー，カット，ペーストが行える．コピー，カットしたオブジェクトは他のモジュールで貼り付ける事も可能である．

## 7.4　各プリミティブの設定

各プリミティブの設計方法について述べる．

### 7.4.1　ステートマシンの設定

ステートマシンプリミティブをダブルクリックした時に表示されるダイアログが図 7.2 である．

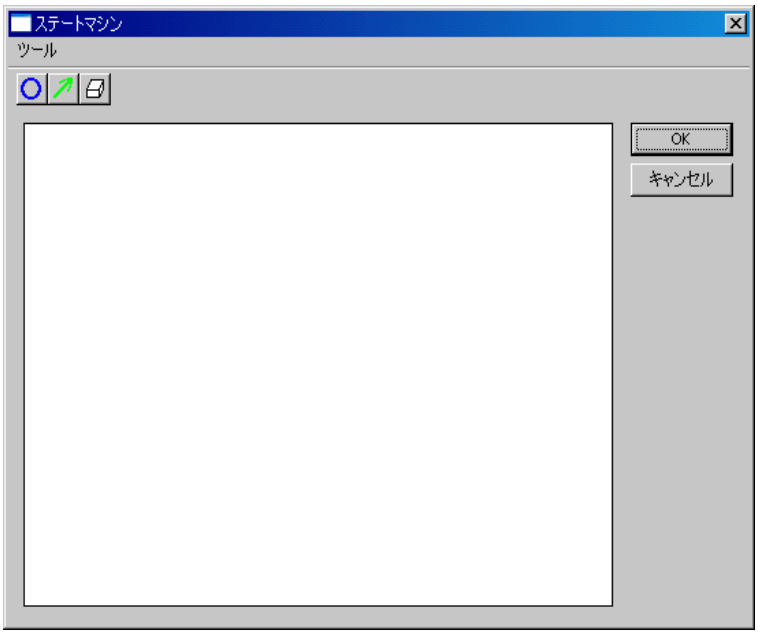



図 7.2: ステートマシンの設定 1

このダイアログに各状態での処理，状態遷移情報を記述する事によりステートマシンとしての処理を行う．状態の処理を記述した例が図 7.3 である．図 7.3 上部に入出力関係を表す関数を記述し，下部に状態遷移を行う関数を記述する．状態遷移は式右辺が真の場合に遷移を行う．main にチェックを入れた状態が初期状態となる．

各状態と状態遷移を記述した例が図 7.4 である．状態の移動先は矢印によって示される．


図 7.3: ステートマシンの設定 2

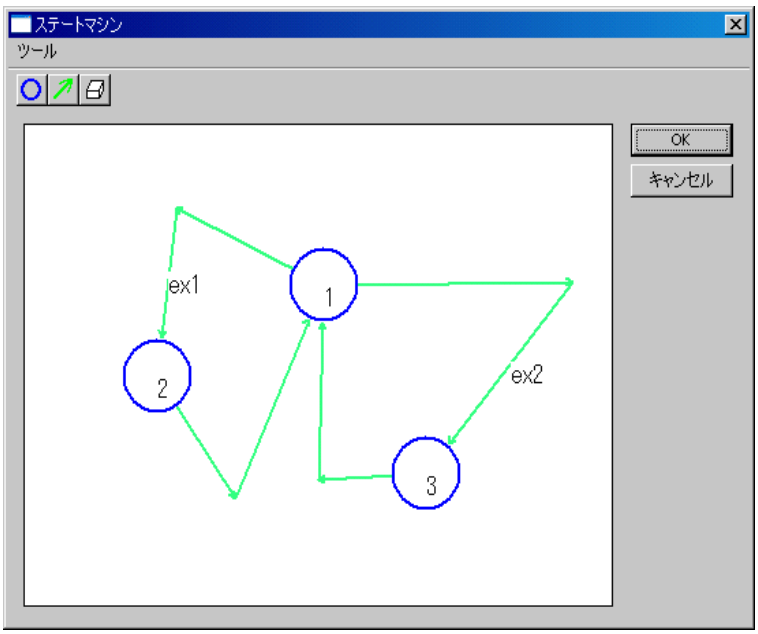



図 7.4: ステートマシンの設定 3

### 7.4.2　シフトレジスタの設定

シフトレジスタプリミティブをダブルクリックした時に表示されるダイアログが図 7.5 である．

図 7.5 左部のテキストボックスに必要ならば初期値を入力する．必要な場合は保持する


図 7.5: シフトレジスタの設定

データ量を指定する．保持データ量を設定しない場合は初期データ量に 1 を足した分のデータが保持される．

### 7.4.3　カウンタの設定

カウンタプリミティブをダブルクリックした時に表示されるダイアログが図 7.6 である．

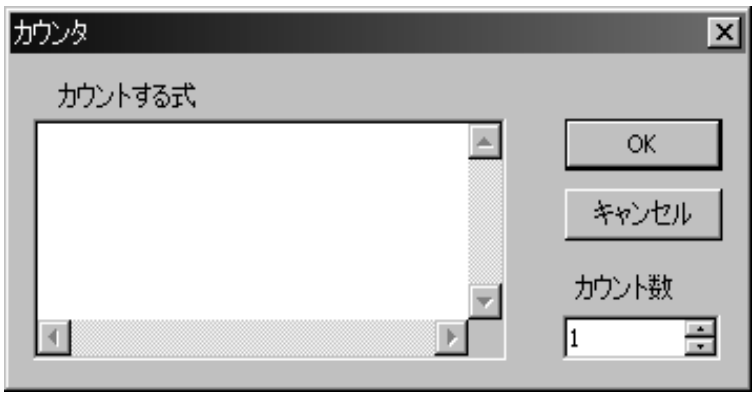



図 7.6: カウンタの設定

図 7.6 左部のテキストボックスにカウントする関数を記入する．カウント数にはカウントする数を設定する．

### 7.4.4　入力の設定

ソーストランジションをダブルクリックした時に表示されるダイアログが図 7.7 である．


図 7.7: ソーストランジションの設定

図 7.7 下部のテキストボックスに入力されるデータを記述する．16 進にチェックをつける事で 16 進数で記述する事も可能である．必要ならば Input name に入力名を記述する．この入力名は他のモジュールと接続する時に使用される．この入力名に new，delete と入力する事によって，モジュールが new 端子によって作成，delete 端子によって削除されると判断される．

### 7.4.5　出力の設定

シンクトランジションをダブルクリックした時に表示されるダイアログが図 7.8 である．

図 7.8 中央のテキストボックスにデータを記述するとシミュレーション時のデータと照合され，合否が診断される．16 進にチェックをつける事で 16 進数で記述する事も可能である．必要ならば Output name に出力名を記述する．この出力名は他のモジュールと接続する時に使用される．

### 7.4.6　モジュールの設定

モジュールプリミティブをダブルクリックした時に表示されるダイアログが図 7.9 である．

接続を行う新たなモジュールを作成する時にモジュール名を設定する．

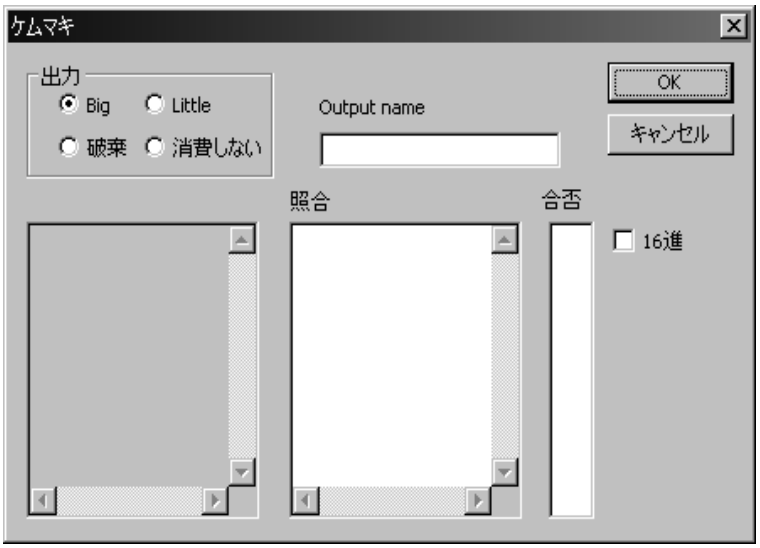



図 7.8: シンクトランジションの設定

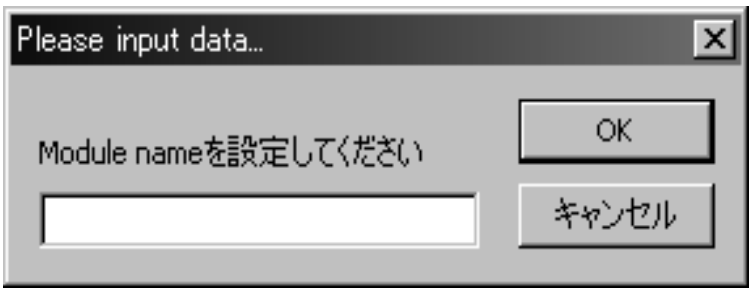



図 7.9: モジュールの設定

## 7.5　各プリミティブの動作

各プリミティブのシミュレーション時の動作について述べる．

### 7.5.1　ステートマシンとシフトレジスタの動作

図 7.10 の動作を例として説明する．この時のステートマシンの各状態は表 7.1 とし，初期状態を st1 とする．

$st1$ は $i1, i2, i3$ の入力を必要とし，$o1, o2, o3$ に出力を行う．すべての入力が揃っており，すべての出力先が空いているためステートマシンは与えられた関数処理と状態遷移を行う．処理後は図 7.11 の様になる．ステートマシンの状態は $st2$ へと移る．

$st2$ は $i1, i2$ の入力を必要とし，$o1, o3$ に出力を行う．シフトレジスタは値をシフトし，入力が揃い，出力先がすべて空いた後にステートマシンは処理を行う．処理後は図 7.12 の様になる．ステートマシンの状態は $st3$ へと移る．

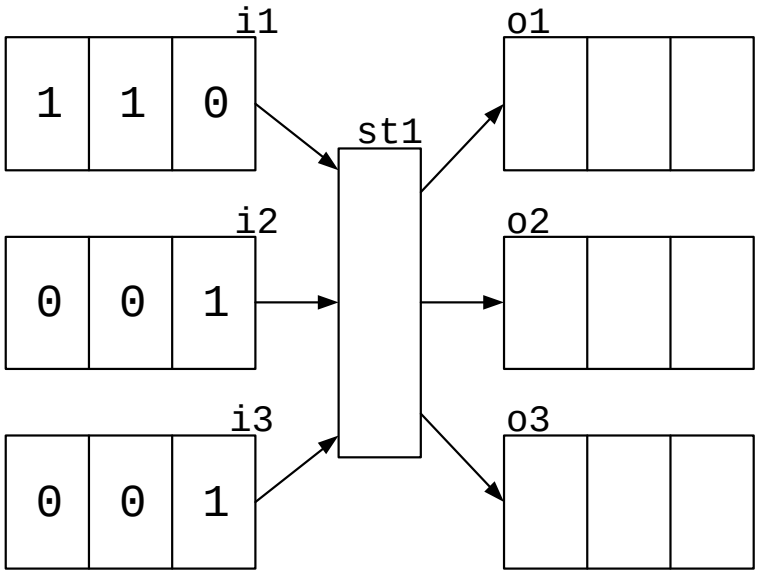



図 7.10: ステートマシンとシフトレジスタの動作 1

表 7.1: 各状態の式

| | st1 | st2 | st3 |
|---|---|---|---|
| 入出力関数 | $o1 = i1;$<br>$o2 = i2;$<br>$o3 = i1\&i2;$ | $o1 = i2;$<br>$o3 = i1;$ | $o2 = i2;$<br>$o3 = \tilde{}i2;$ |
| 遷移関数 | $st2 = i3;$<br>$st3 = \tilde{}i3;$ | $st1 = \tilde{}i1;$<br>$st3 = i1;$ | $st1 = i3 \mid \tilde{}i3;$ |

$st3$ は $i2, i3$ の入力を必要とし，$o2, o3$ に出力を行う．処理後は図 7.13 の様になる．状態は $st1$ へと移る．$st1$ に必要な入力，また出力先が空いていないため，ステートマシンは必要な入力がすべて揃うまで待ち合わせを行う．

トランジションではすべての入力プレースにトークンが配置されるまで発火する事が出来ない．QROQS でも待ち合わせが発生し，入力として使用されるプレースすべてにトークンが配置されるまでトランジションは発火しない．しかし，ステートマシンでは状態ごとに必要な入力が異なる場合もあり，その時は必要な入力のみを待ち合わせ処理を行う．図 7.12，7.13 の様に必要としない入力データは消費しない．また，記述されていない出力先には出力しない．

### 7.5.2 カウンタの動作

カウンタはカウント数を数え，0 または 1 を出力するプリミティブである．カウンタはカウントする関数が真ならばカウント数を減らし，偽ならばカウント数を減らさない．カ

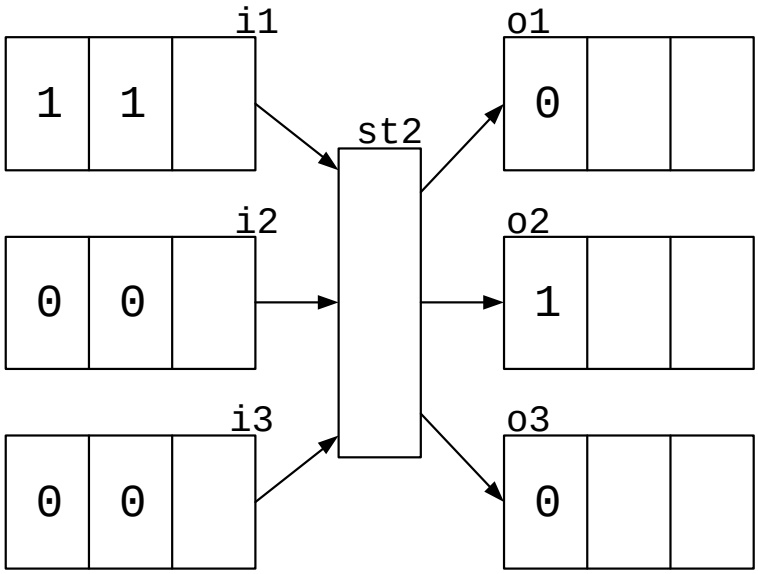



図 7.11: ステートマシンとシフトレジスタの動作 2

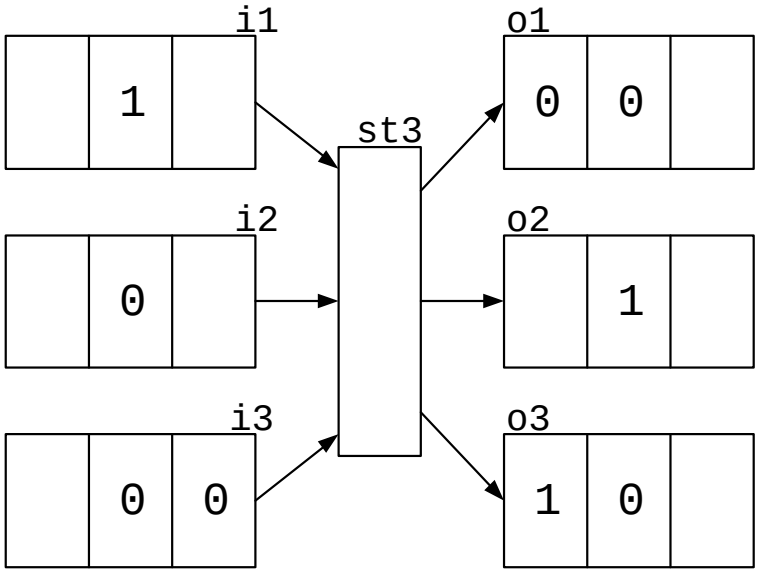



図 7.12: ステートマシンとシフトレジスタの動作 3

ウント数が与えられた値に達した時に 1 を出力する．それ以外の時は 0 を出力する．

カウンタを入力を持たず，出力のみで使用する事も出来る．これはペトリネットで表すと図 7.14 の様になる．ここで用いられているトランジションは現在のカウント数の他に入力を持たず，出力先が空いている時にカウントを行う様になっている．tanoqs でも出力を行う事が出来る場合にカウントを行う事を可能とする．この場合もカウント数が与えられた値に達した時に 1 を出力し，それ以外の時は 0 を出力する．

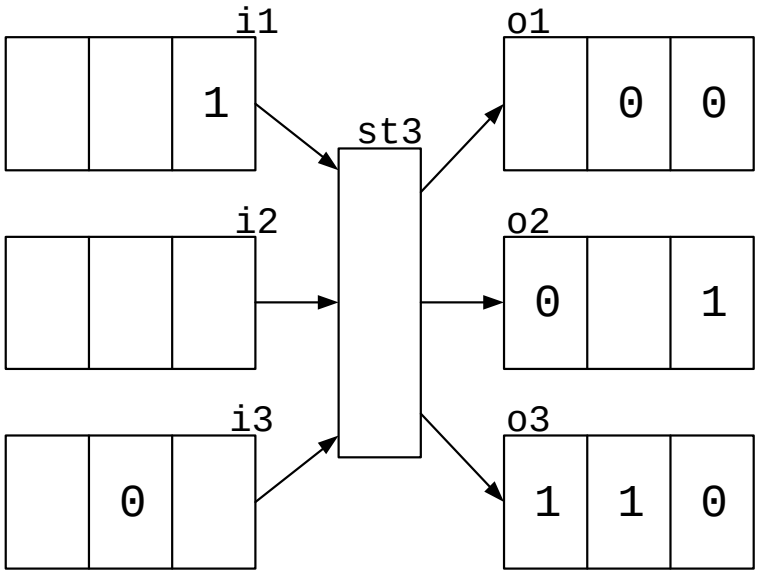



図 7.13: ステートマシンとシフトレジスタの動作 4

### 7.5.3 入出力とモジュールの動作

入力にはソーストランジション，出力にはシンクトランジションが使われる．ソーストランジションは与えられた入力を出力する，もしくはある値を出力し続けるという動作をする．シンクトランジションは値を取り込む動作をする．シンクトランジションによって出力結果を得る事が出来る．

階層構造を用いて設計を行った場合，他のモジュールと接続が行われているソーストランジションは，他のモジュールからデータが到着した場合，その値を自モジュールへの入力として使用する．他のモジュールと接続が行われているシンクトランジションは，そのシンクトランジションにデータが到着した場合，接続されているモジュールへとデータを出力する．

tanoqs 上ではモジュール間の入出力は，モジュールプリミティブからの入出力として見える．モジュールプリミティブへと入力されたデータは下位モジュールのソーストランジションから出され，下位モジュールのシンクトランジションから出力されるデータはモジュールプリミティブから出される．

### 7.5.4 スイッチ

ステートマシンは記述方法によりスイッチとして判断されて実行される．その記述法は図 7.15，表 7.2 の様になる．これは sel1，sel2 の値によって出力先を変化させる処理である．main 状態では入出力処理を行わず，遷移後の状態では必ず初期状態に戻る．このような場合にスイッチとして使用されると判断される．通常のステートマシンでは実行に状態遷移，データ処理と 2 ステップ必要であるが，スイッチでは 1 ステップでそれらを行う．

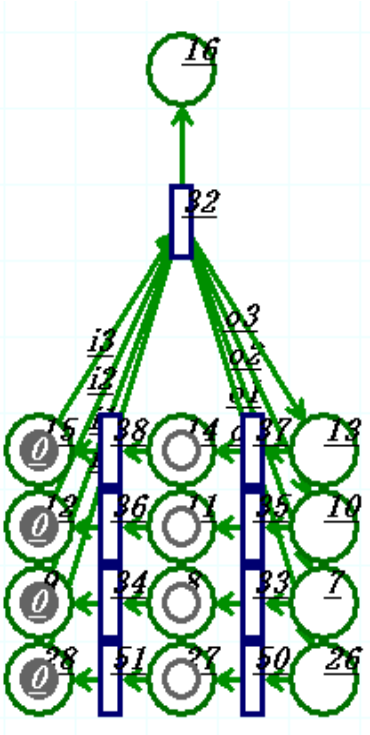


図 7.14: 出力しか持たないカウンタ例

表 7.2: スイッチとして判断されるステートマシン

| main | st1 | st2 | st3 | st4 |
|---|---|---|---|---|
| | $o1 = i1;$ | $o2 = i1;$ | $o3 = i1;$ | $o4 = i1;$ |
| $st1 = \tilde{}sel1 \& \tilde{}sel2$<br>$st2 = \tilde{}sel1 \& sel2$<br>$st3 = sel1 \& \tilde{}sel2$<br>$st4 = sel1 \& sel2$ | $main = i1 \mid \tilde{}i1;$ | $main = i1 \mid \tilde{}i1;$ | $main = i1 \mid \tilde{}i1;$ | $main = i1 \mid \tilde{}i1;$ |

## 7.6　再帰

tanoqs では再帰構造を用いる事によって回路増殖を表現する．ここでは再帰構造の表現法について述べる．

### 7.6.1　回路増殖の判断

tanoqs 上で回路増殖を行うか行わないかは new 端子によって判断される．トップモジュールでないモジュールが new 端子を持つ場合には，そのモジュールは回路増殖を行う

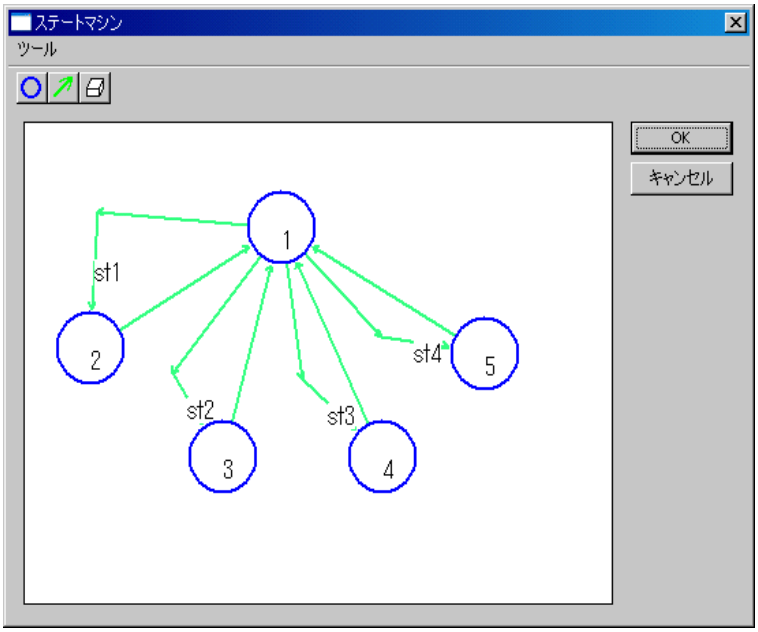



図 7.15: スイッチとして判断されるステートマシン

と判断される．new 端子を持たないモジュールは通常の階層構造と判断される．

### 7.6.2　new 端子

new 端子を持つモジュールはシミュレーション実行時に動的に回路増殖が行われる．回路増殖が行われるタイミングは new 端子にデータが到着した時である．new 端子に到着したデータは回路増殖後，その回路の入力として使用できる．new 端子にデータが到着する前に他の入力が到着した場合はエラーとなりシミュレーションは停止する．

new 端子によって作成されたモジュールはさらに new 端子によって回路を作成する事が可能である．このことにより回路を必要なだけ作り出す事が可能となる．

### 7.6.3　delete 端子

new 端子と共に delete 端子をモジュールは持つ事が出来る．delete 端子は new 端子によって作成されたモジュールを削除する端子である．未だ作成されていないモジュールの delete 端子に入力があった場合にはエラーとなりシミュレーションは停止する．

new 端子と delete 端子は同時に使用されるべきであるが，シミュレータ上では new 端子のみの使用も許される．その場合，作成されたモジュールはシミュレーション停止時に削除される．

### 7.6.4　回路の再帰使用

tanoqs 上では回路が再帰構造かどうかはチェックしておらず，あくまで new 端子と delete 端子だけの処理を行っている．回路を再帰的に使用するためには再帰を行うかどうかを判断するモジュール，再帰により新たに作られたモジュールに接続を切り換えるスイッチが必要となる．

## 7.7　シミュレーション

### 7.7.1　チェック

シミュレーションを行う前に回路が正しく構築されているかどうかのチェックを行う必要がある．チェックは

- ステートマシン，カウンタの関数チェック
- その他プリミティブの属性値のチェック
- 各オブジェクトは正しく接続が行われているか
- モジュール間の接続関係は正しく行われているか

という点について行われる．階層構造を持つ場合には下位モジュールの内容を展開し接続する．new 端子を持ち，回路増殖を行うモジュールは開始時点では展開しない．これらが正しく行われていた場合にチェックは完了し，シミュレーションが行われる．

### 7.7.2　実行

チェックを終了した回路はシミュレーションを行う．この時，

1. シンクトランジションによるデータの取り込み
2. モジュールのデータ取り込み
3. 実行可能と判断されたステートマシンの実行
4. 実行可能と判断されたカウンタの実行
5. シフトレジスタによるデータのシフト
6. ソーストランジションによるデータの出力
7. ステートマシンが実行可能かどうかのチェック
8. カウントが実行可能かどうかのチェック

を順番に繰り返し行う．ステートマシン，カウンタは実行可能かチェックされた後，可能ならば次のステップで実行される．すべてのオブジェクトで処理を行う事が出来なくなった時，シミュレーションを停止する．結果はシンクトランジションで確認できる．回路増殖，削除処理はモジュールのデータ取り込み時に行われる．

　これらの処理は全モジュールで並列的に行われる．

# 第8章

# 記述実験

再帰構造により動作中に回路の追加，消去が可能なシミュレーション環境を作成した．今回，FFT を再帰構造を用いて記述する実験を行った．

## 8.1 FFT

図 8.1 に FFT 演算の様子を示す．FFT はこのようにバタフライ演算を行う事で処理を行う．

図 8.2 に今回記述実験を行ったビットシリアル FFT の処理の流れを示す．入力としてデータと点数が入力されてくる．この点数を変更する事により任意個のデータの処理を行う事が出来る．入力された点数は二分の一され，必要ならば再帰を行い FFT モジュールを動的に作成する．データは点数によって待ち合わせを行いバタフライ演算を行い動的に作成された FFT モジュールに送られ再帰的に処理が行われる．

今回は実際に FFT の演算は行っておらず，再帰的に回路の作成，消去が行われる事を確認する事を目的としている．今回作成した回路に演算部を加える事により FFT の処理を行う事が出来る．

## 8.2 作成した回路

今回再帰により回路の増殖を行い処理を行う回路，処理が終わると再帰により作成された回路の削除を行う回路の 2 つの回路作成を行った．

### 8.2.1 必要な回路構成

必要な回路構成は以下の通りである．

1. 点数の入力 (new 端子)
2. データの入力
3. 演算結果の出力
4. 点数によって再帰を行うかどうかを決定する回路

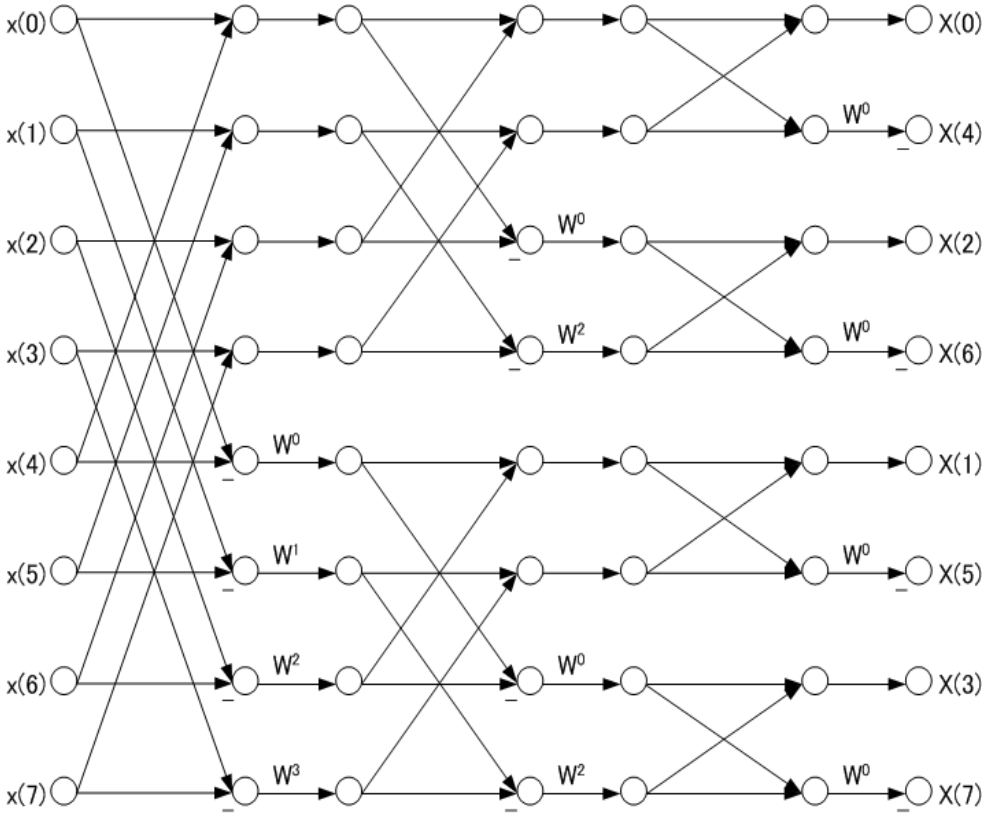



図 8.1: 通常の FFT

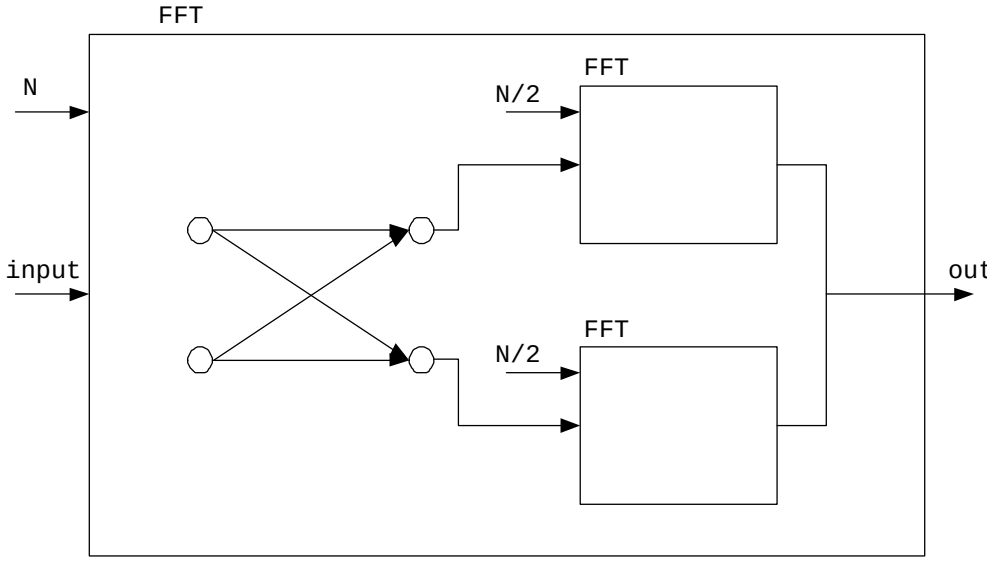



図 8.2: 再帰により処理を行うビットシリアル FFT

5. 再帰を行うかどうかを切り換えるスイッチ
6. ビットシリアルでデータが入力されるため，対となって演算が行われるデータが入力されるまでデータを蓄えておく回路
7. 再帰により新たに作られる回路
8. 出力データを並び替える回路
9. 点数を二分の一にする回路
10. delete 端子
11. delete 信号を上位モジュールへ伝搬する回路
12. 処理の終了を認識し，回路の削除信号を出す回路

### 8.2.2 使用したモジュール

使用したモジュールは以下の通りである．

- FFT

  トップモジュール．以下に示すモジュールなどにより構成される．new 端子を持ち，このモジュールが再帰的に作成，使用される事により処理を行う．

- dec16

  16 ビットのデータをデクリメントするモジュール．

- gen16_0_16_0

  入力された 1 ビットのデータを 16 個作成するモジュール．

- ifequal1_16

  入力された 16 ビットのデータが 1(0000000000000001) で在るかどうかを判断し，1 であれば 1，そうでなければ 0 を出力するモジュール．

- queue16v

  入力されたデータによって動的に領域が確保されキューとして使用されるモジュール．このモジュールも new 端子を持ち再帰的に使用する．

- r_shift16

  16 ビットのデータを 1 ビット右シフトし，二分の一にするモジュール．

### 8.2.3 回路作成のみを行う回路

回路作成のみを行い処理を行う FFT を図 8.3 に示す．図中の番号と回路構成は 8.2.1 節の番号と同じである．

### 8.2.4 処理の流れ

処理の流れを示す.

- 図 8.3 の 1 から点数，2 から処理データが入力される．それぞれ LSB から入力される.
- 点数は図 8.3 の 4 で再帰を行うかどうか (入力値が 1 であるかどうか) 判定される．1 ならばそれ以上再帰を行わない様，1 で無いなら再帰を行う様図 8.3 の 5 に信号を送る．点数は二分の一するため図 8.3 の 9 に送られる．二分の一された点数は，再帰を行う場合，図 8.3 の 7 に送られ，ここで回路が新たに作成される．新たに作成された回路の 1 に点数が入力され，再帰を行わなくなるまで処理を繰り返し必要な分の回路を作成する.
- 再帰を行う場合，処理データは図 8.3 の 6 に送られる．処理データはビットシリアルで入力されるため，演算を行う処理データが到着するまで処理データを保持する必要がある．ここでも再帰呼び出しを用い，データ保持に必要な領域の確保を行っている．幾つ領域を確保するかについては図 8.3 の 9 から送られてくる．演算を行った後は図 8.3 の 7 に送られ，先に作成された回路の 2 に入力され繰り返し処理が行われる.
- 再帰を行わない場合，処理データは図 8.3 の 3 に送られる．そこから出力されたデータは上位の回路で出力順に並び替えられる (図 8.3 の 8).

点数 8 の時の設計段階，実行初期状態，終了状態のモジュール構造は図 8.4 の様になる.

設計段階ではトップモジュールである FFT の他に ifequal_16，r_shift16，gen16_0_16_0，dec16，queue16v といったモジュールがある．queue16v モジュール内にはさらに ifequal_16，dec16 というモジュールがある.

実行初期段階ではこのうち queue16v モジュールが表示されていない．これは queue16v モジュールは new 端子を持つためである．トップモジュールを除き，new 端子を持つモジュールは初期状態では作成されておらず，new 端子に入力が入った時点ではじめて作成される.

終了状態では多くの回路が作成されているのが分かる．展開していないが queue16v モジュールも複数個作成されている.

### 8.2.5 削除処理を行う回路

回路作成を行い，処理が終了した後作成した回路の削除を行う FFT を図 8.5 に示す．図中の番号と回路構成は 8.2.1 節の番号と同じである.

### 8.2.6 処理の流れ

処理の流れを示す.

- 図 8.5 の 1 から点数，2 から処理データが入力される．それぞれ LSB から入力される．図 8.5 の 10 の delete 端子からは何も入力されない．

- 点数は図 8.5 の 4 で再帰を行うかどうか (入力値が 1 であるかどうか) 判定される．1 ならばそれ以上再帰を行わない様，1 で無いなら再帰を行う様図 8.5 の 5 に信号を送る．点数は二分の一するため図 8.5 の 9 に送られる．二分の一された点数は，再帰を行う場合，図 8.5 の 7 に送られ，ここで回路が新たに作成される．新たに作成された回路の 1 に点数が入力され，再帰を行わなくなるまで処理を繰り返し必要な分の回路を作成する．

- 再帰を行う場合，処理データは図 8.5 の 6 に送られる．処理データはビットシリアルで入力されるため，演算を行う処理データが到着するまで処理データを保持する必要がある．ここでも再帰呼び出しを用い，データ保持に必要な領域の確保を行っている．幾つ領域を確保するかについては図 8.5 の 9 から送られてくる．演算を行った後は図 8.5 の 7 に送られ，先に作成された回路の 2 に入力され繰り返し処理が行われる．

- 再帰を行わない場合，処理データは図 8.5 の 3 に送られる．そこから出力されたデータは上位の回路で並び替えられる (図 8.5 の 8)．データの出力が終了した時点で上位モジュールに delete 信号が送られ自モジュールが削除される．上位モジュールでは 2 つの FFT モジュールから delete 信号が送られてくるまで待ち合わせを行い，2 つの信号が到着した時点でデータを蓄えておいた queue16v モジュールに delete 信号を送る (図 8.5 の 12)．queue16v モジュールに送られた delete 信号は queue16v モジュールの最下位に送られ，最下位から順に削除され，delete 信号は上位に伝搬されていく．delete 信号によって queue16v モジュールがすべて削除し終えると，delete 信号は同モジュール内の 2 つの FFT モジュールに削除信号を送る．delete 信号を送るタイミングは点数によって管理されており，下位モジュールから先に削除されていく．

点数 8 の時の設計段階，実行初期状態，終了状態のモジュール構造は図 8.6 の様になる．設計段階では図 8.4 よりも多くのモジュールが使用されている．これは delete 信号を送る際の制御に使用されているモジュールである．実行初期段階ではこのうち queue16v モジュールが作成されていない状態となる．終了状態では作成された回路が削除され，回路構造が初期状態に戻っている事が分かる．削除要求が行われる前には図 8.4 の最終状態の様に複数のモジュールが作成されていた．

## 8.3 実験結果

今回再帰により回路の増殖を行い処理を行う回路，処理が終わると再帰により作成された回路の削除を行う回路の 2 つの回路作成を行った．実行結果は図 8.4，図 8.6 で示した様になった．このことにより，再帰構造によって回路の生成，また削除が行われた事が分かる．

点数を 8 として実験を行ったが，今回作成した回路では点数とデータを自由に変更する事が可能である．

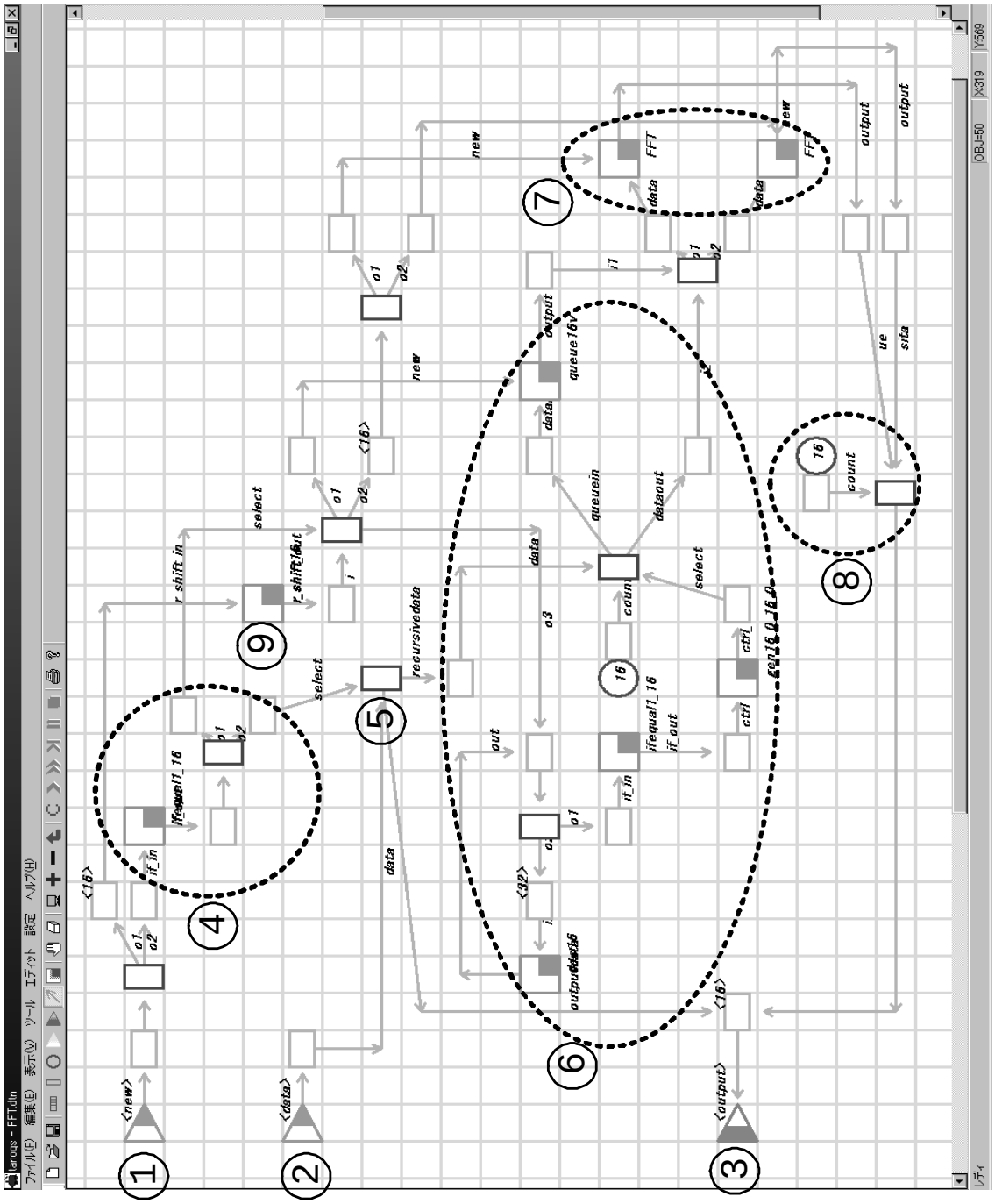
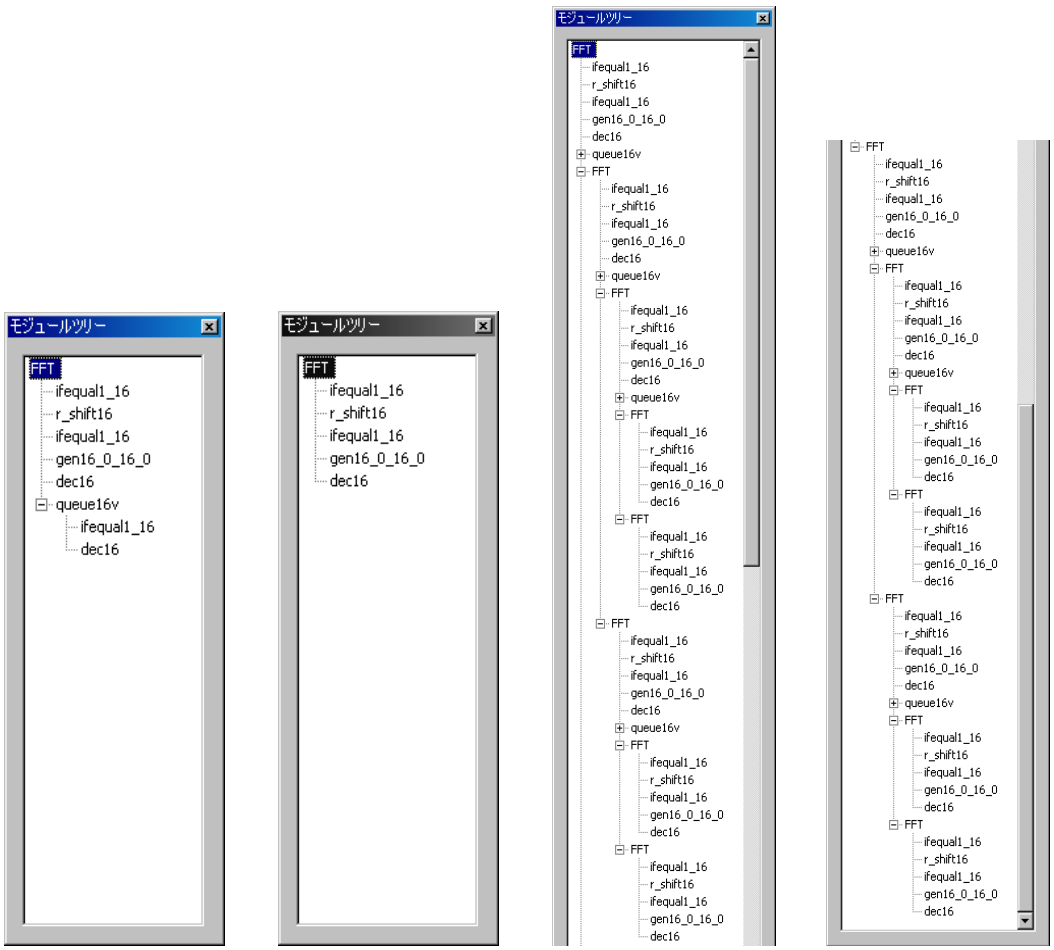



図 8.3: 回路作成のみを行い処理を行う FFT




(a) 設計段階　(b) 実行初期状態　(b) 終了状態 (1)　(b) 終了状態 (2)

図 8.4: 削除処理を行わない場合の回路構造の変化

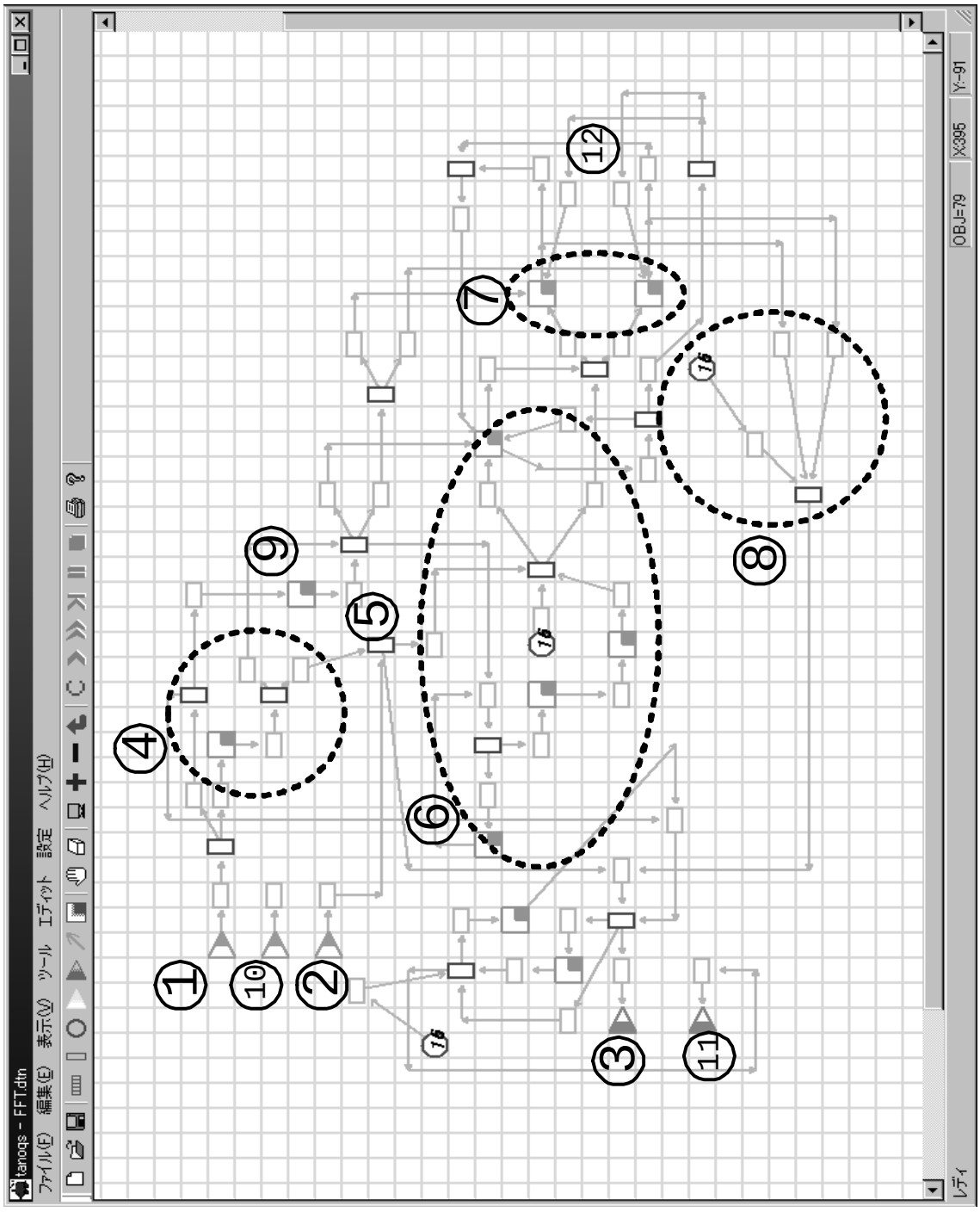
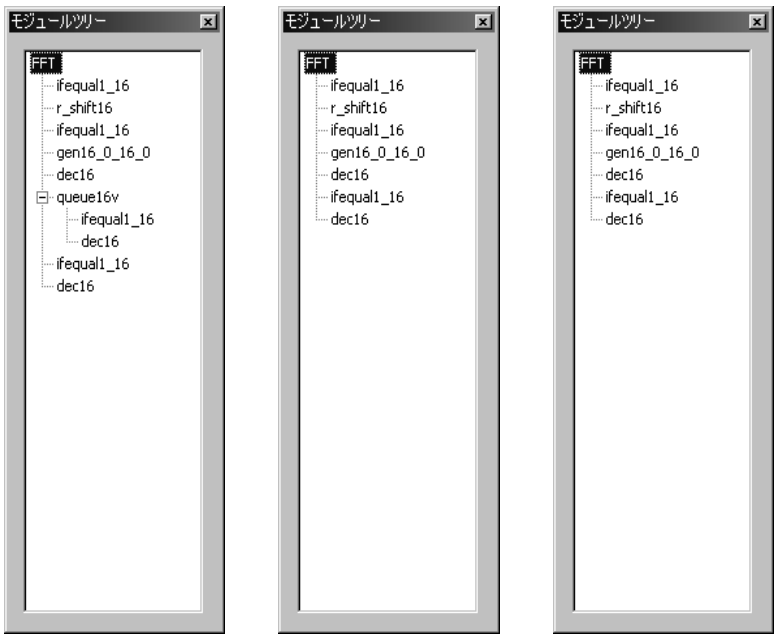



図 8.5: 回路の削除処理を行う FFT




(a) 設計段階 (b) 実行初期状態 (b) 終了状態

図 8.6: 削除処理を行う場合の回路構造の変化

# 第9章

# 今後の課題

今後の課題を以下に示す.

- Bit Serial PCA へのマッピング

  tanoqs は当研究室で研究を行っている Bit Serial PCA をターゲットとし作成を行っている．Bit Serial PCA は可変部の要素をステートマシンとシフトレジスタとしており，tanoqs はその要素をプリミティブとしている．tanoqs で作成された回路の Bit Serial PCA へのマッピングを目指す.

- シミュレーション速度の向上

  tanoqs は当初 QROQS よりも回路設計，シミュレーション時間の短縮を目標として開発してきた．ペトリネットレベルからステートマシン，シフトレジスタへと抽象度を上げる事によって速度向上を図ったが，今回回路が動的に追加，削除される事となり，そのタイミングでのシミュレーション回路の追加，ツリーの構築などに時間が掛かる様になり，シミュレーション速度の低下を招いた．今後はシミュレーション速度の向上を行う必要がある.

- 新たな構成プリミティブの設定

  tanoqs はステートマシン，シフトレジスタ，カウンタを構成プリミティブとしている．スイッチなどはステートマシンの記述を工夫する事によって表現しているが，スイッチそのものを構成プリミティブとして回路設計を行う方がより効率的である可能性がある.

- 他の回路増殖法の模索

  今回再帰表現による回路追加の例として FFT を作成した．また，データの記憶領域の確保も再帰表現を用いて行った．しかし再帰表現では回路追加ができない例もあると思われる．今回は new，delete 端子を用い，再帰的にモジュールを用いる事で回路増殖を表現したが，より良い方法があるかもしれない．場合によっては再帰表現ではなく他の方法を模索する必要があると思われる.

- 非同期ビットシリアル回路設計の言語化

  tanoqs を始め，現在までに研究室で作成された非同期ビットシリアル回路の開発環境はマウスを用い GUI によって回路設計を行っている．GUI による設計，シミュ

レーションはデータの流れを目で追う事が出来，直感的に動作を確認する事が出来るという利点がある．しかし，これから先さらに大規模な回路を設計する事になるとGUIでの作成には限界があると思われる．そこでGUIによらず言語によって非同期ビットシリアル回路の設計を行える環境を考える必要があると思われる．

# 第10章

# 結 論

今回，増殖する非同期ビットシリアル回路の表現法として再帰表現を取り上げ，再帰表現によって回路の増殖を設計できるシミュレータを作成した．回路を増殖し処理を行う例として FFT の作成を行い，new，delete 端子を用い，再帰表現によって回路の作成，処理の実行，削除が行える事を確認した．動的に回路が作成され，それぞれの回路が並列的に動作する事によってより高速な処理が行えるものと考えられる．

今後はさらに記述実験を進め，再帰表現による回路作成の更なる可能性の模索，ハードウェアの並列性と柔軟性をどのように活かすかなどの検討を行う必要がある．最終的には tanoqs で作成した回路の Bit Serial PCA へのマッピングを目指す．

# 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，優しく，時には厳しく親身になって見守って頂いた本研究室博士 2 年の永本太一さんに深く感謝します.

また，共に研究を行った本研究室修士 2 年の上田真里江さんに厚く敬意を表します.

また，私の様な者と共に研究してくれた研究室のメンバー全員に感謝します.

# 参考文献


[1] 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(January 2000)

[2] 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(March 2000)

[3] 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(July 2000)

[4] 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.

[5] H. Ito, R. Konishi, H. Nakada, H.Tsuboi and A.Nagoya, “Dynamically Reconfigurable Logic LSI designed as Fully Asynchronous System - PCA-2,” Proc. of COOL Chips VI, pp.84, Apr. 2003.

[6] 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, pp155-162, Feb.2005.

[7] 小西 隆介，伊藤 秀之，中田 広，名古屋 彰: “動的再構成可能アーキテクチャの視覚的シミュレータの開発”，第 1 回リコンフィギャラブルシステム研究会，pp.127-131，(September 2003).

[8] 神山知己，倉田圭吾，池畑陽介，北道淳司，黒田研一，“動的再構成デバイス PCA 上での自己複製アプリケーション設計容易化手法の提案と実装，” 情報処理学会研究報告 IPSJ SIG Technical Reports, vol.2005, No.8, pp.141-146, 2005.

[9] 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, pp344-352, Feb.2005.

[10] 永本太一，坂本博和，小佐々武志，柴田裕一郎，小栗清，“アプリケーションからの PCA 基本要素の抽出，” 第 4 回リコンフィギャラブルシステム研究会，pp.87-94, Sep.2004.

# 付録 A

# tanoqs について

ここでは tanoqs の歴史，プログラムの構造などについて述べる．tanoqs のソースファイルなどを変更する際に参照して頂ければ幸いである．

## A.1 tanoqs の歴史

tanoqs の開発は学部 4 年から始まる．院試終了後の 10 月から開発を始め，1 月に完成．卒業論文として発表．この時点では C 言語で書かれていた．機能としてもビューの拡大縮小が出来ないなど問題があった．

M1 になり C++言語を用いて tanoqs を一から作り直す事にする．ここでの変更事項は

- ビューの拡大縮小
- ステートマシンの設定の見直し
- tanoqs から qroqs へのファイル出力
- シミュレーション速度の向上

である．シミュレーション速度はこの時点で qroqs よりも速くなった．2004 年 9 月に行われたリコンフィギャラブルシステム研究発表会において発表．

就職活動を終えた後，再帰構造による回路増殖機能を tanoqs に追加する事になる．当初は new 端子などはなく，再帰を行うモジュールにデータが到着した時点で回路増殖を行う方式であった．しかしこの方式では自由度が高すぎるなど問題があり，new，delete 端子を用いる事になる．この変更に際して必要のないメンバ変数などが存在しているため注意が必要である．

ここでの変更事項は

- 回路の階層構造
- 再帰構造による回路増殖

である．tanoqs から qroqs へのファイル出力機能はこの時点で無くなる．今回記述実験で作成した FFT 回路を EDSFair2006 で展示する．

## A.2　tanoqs のデータ構造

tanoqs で使用されている主なクラスは以下の通りである．

- dtntop

  tanoqs 内で唯一つ存在するクラス．objdata クラス，moduletree クラスなどを持つ．

- objdata

  オブジェクトを管理するクラス．このクラスはモジュール一つの中身のオブジェクトを管理している．

- moduletree

  モジュールのツリー構造を管理しているクラス．

- state

  ステートマシンの情報を保持するクラス．

- t_count

  カウンタの情報を保持するクラス

- substate

  ステートマシンの各状態のデータを保持するクラス．このクラスは state クラス内に存在する．

- shift

  シフトレジスタの情報を保持するクラス

- uhzi

  ソーストランジションの情報を保持するクラス

- kemmakix

  シンクトランジションの情報を保持するクラス

- module

  オブジェクトとして配置されたモジュールの情報を保持するクラス．

tanoqs は dtntop クラスによって管理されている．dtntop クラスは唯一つのみ存在するクラスである．dtntop クラスはすべてのモジュールの情報を持っている．階層構造については dtntop クラス内にある moduletree クラスによって管理される．

作成された回路構造は objdata クラスで管理される．objdata クラスはモジュールごとに作られ，ステートマシン，シフトレジスタ，カウンタ，モジュール，アーク，ウージ，ケムマキの情報を保持している．これらは std::map で保存されている．

ツリー構造を管理するクラスは moduletree クラスである．シミュレーション時にはモジュールツリークラスに objdata クラスがコピーされ処理が行われる．

## A.3　表示部とデータの関連

表示部にオブジェクトが配置された時，tanoqsView クラスで情報が管理される．表示部が表示された際，表示部の大きさ分の配列が作成され，その配列にオブジェクトの ID が詰め込まれる．マウスでダブルクリックされた際，そのポイントに配置されているオブジェクト ID を元に objdata クラスによってオブジェクトが何であるか判断される．

画面へのオブジェクトの表示は CtanoqsView::PrintDraw(CDC*, CtanoqsDoc*) によって行われる．これは CtanoqsView::OnDraw(CDC*) から呼ばれ，画面の書き換えが必要な時に実行される．書き換え時の画面のちらつきを押さえるために，画面は一度ビットマップを作成し，そのビットマップに書き込んだ後ビューに転送している．

## A.4　ファイル保存，読み込み

tanoqs のファイルの保存，読み込みを行うクラスは tanoqsFile クラスである．

上書き保存を行う場合，CtanoqsView::OnFileSave() が呼ばれる．ここで tanoqsFile::FileSave(FILE*,objdata*) によってファイルの書き出しが行われる．名前を付けて保存する場合は CtanoqsView::OnFileSaveAs() が呼ばれる．ここで名前を付け，tanoqsFile::FileSave() によって保存が行われる．上書き保存，名前を付けて保存の両方とも現在開かれているモジュール一つのみの保存を行う．一括保存を行う場合 CtanoqsView::OnAllsave() が呼ばれる．ここから dtntop クラスが保持しているすべてのモジュールの保存が行われる．

読み込みを行う場合，CtanoqsView::OnFileOpen() が呼ばれる．ここで tanoqsFile::FileLoad(FILE*, objdata*) によってファイルの読み込みが行われる．ファイル内にモジュールデータがあり階層構造が使われている場合にはここから CtanoqsView::RecursiveReadFile(objdata*, moduletree*) が呼ばれる．この中でも tanoqsFile::FileLoad(FILE*, objdata*) によってファイルが読み込まれる．さらにモジュールデータがあった場合には再帰的に CtanoqsView::RecursiveReadFile(objdata*, moduletree*) を使用する．

## A.5　tanoqs のファイル形式

tanoqs のファイルは.dtn という拡張子で保存される．そのファイル形式は表 A.1 の様になっている．ID はオブジェクトの ID，X はオブジェクトの X 座標，Y はオブジェクトの Y 座標，DATA はオブジェクトの保持する値を表す．

表 A.1 の shift 内の SDATA1 は領域の値を保持するかどうか，SDATA2 は領域の値，SDATA3 はデータの表示方向を表す．

同表 state の MAIN はどの状態が初期状態かを表す．sub はステートマシンが保持する状態ごとの設定であり，SUBID は状態の ID，SIKI は関数，SENI は遷移関数となる．smarc はどの状態からどの状態へ移動するかを表す矢印の設定であり，START は遷移元，END は遷移再帰を表す．() 内の x,y は矢印の中継座標であり中継座標がない場合は記述されない．ARCNAME は関数で使用された遷移先の名前である．

表 A.1: tanoqs のファイル形式

```
//tanoqs file
shift[ID,X,Y,SDATA1,SDATA2,SDATA3]{
DATA}
…
state[ID,X,Y,MAIN]{
sub[SUBID,X,Y]{
{SIKI}
{SENI}}
smarc[START,END,(x,y)]{ARCNAME}
…
}
…
count[ID,X,Y,NUM]{
COUNTSIKI}
…
uhzi[ID,X,Y,HOZI,SAYUU,HEX,UHZINAME]{
DATA}
…
kemmakix[ID,X,Y,KEMS,HEX,KEMNAME]{
DATA}
…
module[ID,X,Y]{
MODULENAME}
…
arc[START,END,(x,y)]{ARCNAME}
…
```

同表 count の NUM はカウント数を表す．COUNTSIKI はカウントを行うかどうかを判定する式を表す.

同表 uhzi の HOZI は同じ値を出力し続けるかどうか，SAYUU は MSB，LSB どちらから値を出力するか，HEX は DATA が 16 進であるかどうか，UHZINAME は階層構造で使用されるモジュールの入力名を，DATA はシミュレーション時に出力するデータを表す.

同表 kemmakix の KEMS はシミュレーション時に入力される値を MSB，LSB どちらとするか，また値を破棄するかどうかを，HEX は 16 進でデータを表すか，KEMNAME は階層構造で使用されるモジュールの出力名を，DATA はシミュレーション時に比較するデータを表す.

同表 module の MODULENAME はモジュール名を表す.

同表 arc の START は接続元の ID を，END は接続先の ID を，() 内の x,y は矢印の中

継座標を，ARCNAME は接続名を表す.

## A.6　情報表示

アウトプットウィンドウに文字を表示する関数は trace(const char *, ...) である．C 言語の printf 関数と同じように使用できる.

## A.7　モジュールツリーダイアログ

階層構造を表示しているのが ModuleTreeDlg である.
ModuleTreeDlg::ChangePrintData() で全モジュールツリーの書き換え,
ModuleTreeDlg::AddModuleData(const string &modulename, HTREEITEM) で
HTREEITEM で指定したモジュールの下位に modulename のモジュールを追加する．AddModuleData は HTREEITEM を返す．これは追加したモジュールのモジュールツリーダイアログでの位置情報である.

モジュール名がクリックされた時は ModuleTreeDlg::OnTvnSelchangedTree2(NMHDR *pNMHDR, LRESULT *pResult) が呼ばれる．ここで選択された HTREEITEM をメンバ変数として保持し，メッセージを送り選択された事を知らせる．メッセージを受け取ったビューは HTREEITEM によってビューを変更する.

## A.8　コンパイル時の動作

CtanoqsView::OnCompile() においてシミュレーションを実行する前にチェックを行う．チェックを行う前に，まず moduletree に各モジュールの objdata をコピーする．その後に moduletree の objdata をチェックする．この際，new 端子を持つモジュールは objdata のコピーを行わない.

実行を行うモジュールは dtntop::vector m_CompileModuleTreePointerVect に格納される．また，m_CompileModuleTreePointerVect の begin() と end() のイテレータを m_CompStartitr，m_CompEnditr に保存する．このイテレータの範囲でシミュレーションを実行する.

## A.9　実行

シミュレーション実行時には CtanoqsView::OnStep() を実行する．ここから CtanoqsView::StepOne(bool &) が呼ばれ，実行される．StepOne() からは dtntop::AllStepOne(bool &) が呼び出される．ここからすべてのモジュールに対し実行命令が出される.

アニメーションなし実行の場合は CtanoqsView::OnNoAnime() を実行する．ここから CtanoqsView::NoAnimeStep() が呼び出され処理が行われ，処理が終了した時点で画面を書き換える.

アニメーション実行の際には CtanoqsView::OnTimer(UINT nIDEvent) によって SetTimer される．このことにより CtanoqsView::OnTimer(UINT nIDEvent) 内で CtanoqsView::OnStep() が連続して呼び出される．

アニメーション実行時に一時停止ボタンが押された時，CtanoqsView::f_pause を true にする．この f_pause は CtanoqsView::OnTimer(UINT nIDEvent) によって毎ステップ監視され，ture ならば KillTimer で実行が中断される．

回路が増殖した場合には，moduletree にモジュールを追加し，objdata をコピーし，上位，及び下位モジュールとの接続を行い，その moduletree のポインタを m_CompileModuleTreePointerVect に追加し，イテレータを変更する．

## A.10　シミュレーションの終了

シミュレーション停止の際には CtanoqsView::OnCompStop() を実行する．ここから CtanoqsView::AllCompStop() が呼ばれ全体の処理を終了する．処理の終了の際には先ずすべてのモジュールで作成したシミュレーション用の objdata の削除を行う．その次に実行中に作成された moduletree を削除する．最後に m_CompileModuleTreePointerVect の中身をクリアする．

## A.11　終了処理

tanoqs を終了する際に CtanoqsView::OnDestroy() によって設定が tanoqs.ini ファイルに保存される．保存される内容は

- 各オブジェクトの表示色
- シフトレジスタのデータの表示方向の設定
- アウトプットウィンドウの表示非表示
- フォント設定

である．

また，ここで作成した各オブジェクトの pen や brush，表示に使用するビットマップが破棄される．

メモリリークを防止するため tanoqs で new によって作成された moduletree は dtntop のデストラクタで削除される．

## A.12　各メニュー実行の際に呼び出される関数など

- シフトレジスタ

  CtanoqsView::OnShift()

- ステートマシン

  CtanoqsView::OnState()

- カウンタ

  CtanoqsView::OnCount()

- ウージ

  CtanoqsView::OnUhzi()

- ケムマキ

  CtanoqsView::OnKemmakix()

- アーク

  CtanoqsView::OnArc()

- モジュール

  CtanoqsView::OnModule()

- ハンド

  CtanoqsView::OnTebkuro()

- 消しゴム

  CtanoqsView::Onkesigomu()

- アウトプット

  CtanoqsView::OnOutPutButton()

- モジュールアップ

  CtanoqsView::OnModuleUp()

- コンパイル

  CtanoqsView::OnCompile()

- アニメーション実行

  CtanoqsView::OnAnime()

- アニメーションなし実行

  CtanoqsView::OnNoAnime()

- ステップ実行

  CtanoqsView::OnStep()

- 一時停止

  CtanoqsView::On_t_Pause()

- シミュレーション停止

  CtanoqsView::OnCompStop()