設 計 自 動 化 60−10
（1991．12．12）

プロセス記述による非同期式制御回路合成の一手法

籠谷裕人　　南谷 崇

東京工業大学 工学部

本稿ではまず、プロセッサの非同期的な構成における、効率的な動作の実現法を提案する。従来要求と応答による通信を主体にした非同期回路は、休止相のオーバーヘッドにより効率的な実行を阻まれたが、本稿で提案するモジュールの挿入により論理的な意味を変えずにオーバーヘッドをほとんど打ち消すことができる。さらに、プロセッサの各機能ブロックのプロセスとしての仕様記述から、基本プロセスへの分解による制御回路を合成する手法を提案し、より一般的なデータ信号を含む回路を容易に扱えるようにした。また、上の手法によって基本回路を実現することで、動作の効率的な実行が可能となる。

A Synthesis Method for Asynchronous Control Circuits based on Process Description

Hiroto Kagotani　　Takashi Nanya

Faculty of Engineering, Tokyo Institute of Technology
2–12–1 Ookayama Meguro-ku Tokyo 152, Japan

A synthesis method is presented for asynchronous control circuits based on an asynchronous process model. Conventional asynchronous circuits based on communication with request and acknowledge have suffered a significant overhead in performance due to idle phases required for 2-phase operation. We propose a circuit module called "auto-sweeping module" which effectively eliminates the overhead due to the idle phases. We also propose a method for decomposing a process description of a functional block to primitive processes so that the synthesis is made for more general modules including data in it. Implementing these primitive processes to hide idle phases allows circuits to run faster.

1 まえがき

今後予想されるチップ内配線遅延の相対的な増大を考慮すると、クロックを排した非同期式ディジタル回路は、回路設計上の重要な位置を占める [8]。非同期式制御回路は従来 Huffman の順序回路モデルを基本とする方法で設計されている。しかし最近、Muller の遅延モデルに基づいた設計法がいくつか提案された。これらは、信号遷移グラフ (STG) による仕様を元に制御回路を合成するもの [6, 2, 3] と、他のプロセスとの通信を基本としてプロセスの仕様を記述しこれを制御回路に合成するもの [5, 7] に大きく分けることができる。前者は信号線の記述による詳細な仕様を与え、後者は外部との信号のやりとりはすべてハンドシェークによる通信で表すという点が大きな相違である。後者は、このことから、通信にデータを用いることを記述することによって、データをひとまとめにして扱うことができる。それに対し前者ではデータ信号の一本一本を扱わなければならないため、データの通信を記述することは難しく、データはその制御信号によって間接的に表現することになる。

本稿では、後者のプロセス記述からの回路合成に関して、二相式非同期回路の効率的な動作を可能にする手法について述べる。さらにプロセス記述の分解を経て回路を合成する方式を提案し、これにその効率化手法を適用する。

2 非同期プロセス記述

2.1 非同期回路モデル

本稿で対象とする制御回路は、クロックを完全に排除して非同期的に動作する。そのため制御回路を構成する各機能モジュールは、他の機能モジュールや制御対象とのやりとりを、**要求信号**と**応答信号**によるハンドシェークによって実現する (図 1)。こうした個々の機能モジュールは、他と通信する独立したプロセスとみなすことができる。そこでその機能モジュールの動作仕様を、対応するプロセスの通信動作としてプロセス記述言語で記述する。

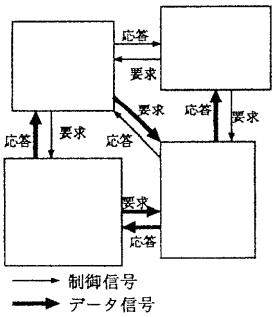


図 1: 機能モジュール間の通信

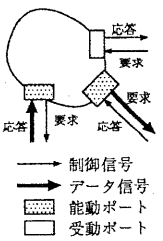


図 2: プロセスの外観

プロセスは通信路に対して、ポートを介して信号を送り、また信号を受けとる。ポートはそれぞれ名前を持ち、同名のポートは同一の通信路に接続される。ポートには**能動ポート**と**受動ポート**があり、能動ポートは要求信号を出して応答信号を受け、受動ポートは要求信号を受けて応答信号を出す。それぞれの通信を**能動通信**、**受動通信**と呼ぶ (図 2)。図ではデータを要求信号や応答信号として用いることを示しているが、データそれ自身にタイミング情報が含まれるように、データは自己同期符号によって符号化されていなければならない。また制御信号による要求や応答はふつうその 1/0 のレベルによって意味付けるので、データも二相符号化しておく方が整合がとれて都合がよい。本稿では二相符号の中でも、回路の設計が容易な**二線二相符号**で符号化されたデータを扱う。

制御信号 単線。1 は要求または応答があることを示す。

データ信号 二線。二線符号語の時は 0 または 1 の値を持つことを示し、スペーサ (全信号線が 0) はデータがないことを示す。

2.2 プロセス記述言語

Martin らは Hoare の CSP[1] を元にしたプロセス記述言語を提案した [4, 5]。さらに Berkel らは能動通信と受動通信を記述の上で明示的に区別するように改良した [7]。本方式では基本的に同様な記法を用いる。より正確な定義は後述するが、いくつかの記述例を以下に示す。

- $*(A' : B; C | D' : E \bullet F)$

  A から F はこのプロセスが持つ通信ポートの名前であり、プロセス記述中ではそのポートでの通信を表す。A' などの $'$ はそのポートが受動ポートであり、そこで受動通信を行うことを表す。; はそれらの動作の直列実行、$\bullet$ は並列実行を示す。: はその左の受動ポートに対する要求信号を受けて、右側の実行を行い、その完了後、元のポートから応答を返すという動作を表す。| は複数の受動ポートへの要求を受けて、それに対応する動作を排他的に実行する。要求は同時に来ることもあり得るので、アービタによって調停する必要がある。$*$ は全体の繰り返しを表すもので、すべてのプロセスに必要である。

  以上より、このプロセスは、ポート A に来た要求によって $B; C$ を実行し、D に来た要求によって $E \bullet F$ を実行するが、二つの要求が同時に来る場合は一方の実行が完了するまでもう一方は待たされるということを意味する。

- $*((A' \bullet B') :$
  $[f(u,v) \rightarrow C!g(u) \,[]\, h(u,v) \rightarrow C!i(v); D])$

  ポート名に $!expr$ をつけると、そのポートからのデータの送信を表す。能動通信の場合はそのデータが要求となり、受動通信の場合はそのデータが応答となる。$expr$ は適当な変数からなる任意の式である。$C!g(u)$ は、変数 u をある関数 g に適用した結果をポート C から送出する。C は能動ポートなので、データは要求となる。

$[\cdots \| \cdots]$ は条件による選択的な実行を表す。この内容は「ガード $\rightarrow$ コマンド」のリストであり、ガードが真になったコマンドのみが実行される。各ガードはどれか一つだけが真にならなければならない。この例の場合ガードは二つだけなので、$h = \neg f$ である。

受動通信の並列実行 $(A' \bullet B')$ は両方のポートに要求が揃った (同期した) 時点で双方に応答を返すことを表す。この例では：があるので両方への要求を待ち、揃ったら右辺を実行してその完了後応答を返す。

- $*(A'?a : B!inc(a)?b \bullet \langle c, d\rangle := \langle dec(a), a\rangle;$
  $D!\langle a, dec(b)\rangle \bullet C!c)$

  $B!inc(a)?b$ はポート B から $inc(a)$ を送出し、その応答として B に戻ってきたデータを変数 b に格納することを表す。これは B の相手がメモリの読みだしポートであると考えると、メモリに対するデータの読み出しを表現することができる。$D!\langle a, dec(b)\rangle$ はポート D に a と $dec(b)$ を連結したデータを送出することを表す。この記法は一般的に利用することもできるが、メモリへの書き込みに、アドレスとデータ信号を送出して応答を受けとるという意味に用いることも可能である。:= は変数への演算結果の代入を表すが、右辺と左辺に同じ変数はないものとする。これは変数のハードウェアによる実現の容易性を保つための制約である。

表 1 はこの記述言語の各記述要素とその意味を表す。演算子の優先順位は、基本的には次のようになる (左側ほど強い)。

$$!, ?, := \quad * \quad \bullet \quad ; \quad \rightarrow, : \quad |, \|$$

ただし、正確な文法は付録 A に示す。

表 1: プロセス記述言語の概要

表記	意味
A, A'	能動制御通信、受動制御通信
$A!expr$, $A'!expr$	(能動、受動) データ送信
$A?v$, $A'?v$	(能動、受動) データ受信
$A!expr?v$, $A'?v!expr$	(能動、受動) データ送受信
:=	代入
$\langle v_1, v_2\rangle$	データの連結
;	直列実行
$\bullet$	並列実行
$[expr \rightarrow \cdots \| \cdots \| expr \rightarrow \cdots]$	条件選択
$\{expr \rightarrow \cdots\}$	条件反復
:	要求受付
|	調停
*	反復

3 二相式非同期回路

3.1 休止相オーバーヘッド

要求と応答による通信で二つの動作の直列な実行を制御する場合、従来は、図 3 のような回路を用いて実現した [5, 7, 9]。

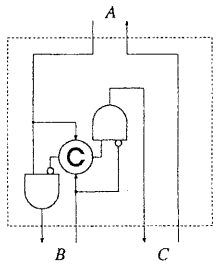


図 3: 従来の直列動作制御回路

これは、本手法の記法による $*(A' : B; C)$ を実現するものである。回路中央の素子は Muller の C 素子であり、全入力が 1 になると出力に 1 を、全入力が 0 になると 0 を出す。それ以外の場合は前の値を保持し、また初期状態は 0 と仮定する。この回路の動作は、A に要求が来ると B から要求を出し、その応答によって要求を下げ、応答が下がると C から要求を出す。C の応答によってそのまま A から応答を返し、A に対する要求が下がったら C の要求を下げて、応答が下がるのを待つ。この様子を図示すると図 4 のようになる。図中、**稼働相**とは要求を出してから応答が返るまでの期間を意味し、**休止相**は要求を下げてから応答が下がるまでの期間を意味する。矢印は図 3 の制御回路によって実現される因果関係を示す。回路が実際に何らかの意味のある動作をするのは稼働相のみであり、休止相は無駄な時間である。しかしこの回路は、どの休止相もそれが終わるまで次の動作に移れない。すなわち、このような休止相のオーバーヘッドによって効率的な動作が阻まれている。

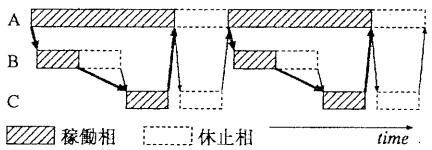


図 4: 従来の直列動作

3.2 休止相と他の稼働相の並列動作

一方、プロセス記述 $*(A' : B; C)$ の意味は、稼働相のみについて、ポート B と C での通信がオーバーラップなしに順次実行され、ポート A での通信がそれを包含することを表す。すなわち図 4 の太い矢印の因果関係だけが実現される回路でも、記述された仕様を満たすことができる。すなわち図 5 のような因果関係を実現する回路を用いることによって、回路の動作の効率化が期待できる。実際 B の休止相が完了する前に C の稼働相が開始でき、また C の休止相が完了する前

に B の稼働相が開始できるため、繰り返しの周期は図 4 の約半分である。

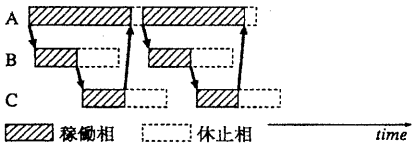


図 5: 改良した直列動作

これを実現するには、図 6 の回路を用いる。回路中の ASM は自掃モジュール (Auto-Sweeping Module) と呼ぶもので、下位の稼働相の後上位に応答を返すと上位からの要求が下がるのを待つことなく下位の休止相を開始させる。この名称は、休止相を回路の清掃とみなすと、上のような動作が使用の終わった回路を自動的にきれいにするように見えることから与えられた。その回路と STG は図 7 で表される。

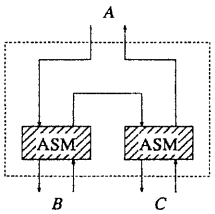


図 6: 改良した直列動作制御回路

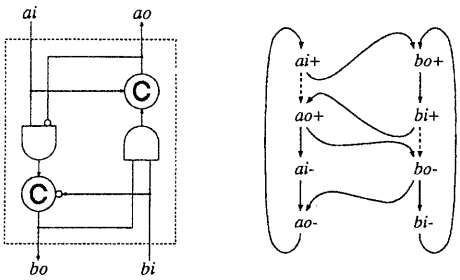


図 7: 自掃モジュール (ASM) とその STG

一般に自掃モジュールは要求と応答によって通信が行われる任意の部分に挿入することができる。回路の論理的な意味は稼働相のみにより決まり、休止相の実行されるタイミングによらないため、挿入する／しないによってその意味が変わらないからである。しかし、データ通信が行われる部分に挿入するための自掃モジュールは回路が複雑になり、そのオーバーヘッドの方が効果を上回ることになる。

また自掃モジュールを直列に (上下に) 接続しても回路全体の意味は変わらないが、一般には動作が遅くなる。下位の自掃モジュールによって、休止相は十分に短くなっているので上位の自掃モジュールは不要である。図 8 のように不要なモジュールを削除する必要がある。

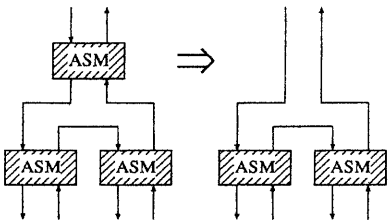


図 8: 自掃モジュールの削除

4 合成

4.1 プロセス記述の分解

4.1.1 基本プロセス

プロセスの記述全体を一度に回路に変換することは難しいので、プロセスの分解について考える。簡単な例を挙げると、$*(A' : B; C; D)$ を $*(A' : E; D)$ と $*(E' : B; C)$ の二つに分けることにより、前節の直列動作をするプロセスと同形のプロセスが二つ生成されるが、このような小さなプロセスには対応する回路を容易に作ることができる。このプロセスに関しては図 6 をそのまま用いればよい。

これを拡張すると、もし、どのようなプロセス記述も限られた種類の簡単なプロセスに分解できるならば、あらかじめその種類だけ対応した回路を用意しておくことで、回路そのものを合成するという作業が全く不要になる。このような単純なプロセスは、原則的に記述言語における記述要素に対応し、その記述要素の提供する機能だけを持つと考えることができる。例えば典型的には先の直列実行のみを行うプロセスや条件選択のみを行うプロセスとなる。こうしたプロセスを数え上げ冗長なものを取り除くと、どんな記述を分解するのにも必要十分なプロセスの集合が得られる。

本方式の記述言語に関して、必要となる基本プロセスを数え上げると表 2 のようになる。この中には、例えば代入のプロセスなどが含まれないが、これはデータ送信と受信で代用できるため、冗長となるので取り除かれている。

4.1.2 分解

プロセス記述を基本プロセスに分解する方法はいくつか考えられる。単純には、記述中の特定のパターンを見つけて適当に分割していくという方法がある。例えば記述中に $X; Y; Z$ というパターンを見つけて、これを $W; Z$ に置き換え、新たに $*(W' : X; Y)$ というプロセスを生成するというような分割規則を多数作れば、記述の分解が可能である [9]。

一方、基本プロセスはもともと記述言語の基本的な要素に由来しているということに着目すると、基本プロセスの種類や配置は、記述の構文の構造に強く影響されることがわかる。実際、上の手法による基本プロセスの接続構造と、記述言語の構文解析木の構造を比較すると、これらはほぼ完全に対応する。そこで構文解析木の各ノードを順々にたどって、それに対応するプロセスを作っていくことで、基本プロセスの集

表 2: 基本プロセス

$*A$	能動プロセス
$*A'$	空プロセス
$*(A' : B; C)$	直列実行プロセス
$*(A' : B \bullet C)$	並列実行プロセス
$*((A' \bullet B') : C)$	受動同期プロセス
$*(A' :[expr_1 \rightarrow B_1 \| \cdots \| expr_n \rightarrow B_n])$	条件選択プロセス
$*(A' : \{expr \rightarrow B\})$	条件反復プロセス
$*(A'_1 : B_1 \| \cdots \| A'_n : B_n)$	調停プロセス
$*((A'_1 : B_1); C_1 \| \cdots \| (A'_n : B_n); C_n)$	ノンブロッキングプロセス
$*(A' : B!expr)$	能動送信プロセス
$*(A' : B?v)$	能動受信プロセス
$*(A' : B!expr?v)$	能動送受信プロセス
$*(A'!expr : B)$	受動送信プロセス
$*(A'?v : B)$	受動受信プロセス
$*(A'?v!expr : B)$	受動受送信プロセス

合とその接続関係が得られる。例外的に：に対応するノードより上位のノードは、一まとめにして調停プロセスとノンブロッキング制御プロセスとなる。

一例として、ALU の制御を行うプロセスを記述し、その構文解析木を生成して、基本プロセスへの分解を行う。

$$
\begin{aligned}
ALU \equiv *(&A'?op : \\
&[unary(op) \rightarrow X?x; \\
&\qquad \langle z, f1\rangle := uop(op, x, f); \\
&\qquad Z!z \bullet f := f1 \\
&\| binary(op) \rightarrow X?x \bullet Y?y; \\
&\qquad \langle z, f1\rangle := bop(op, x, y, f); \\
&\qquad Z!z \bullet f := f1] \\
&| F'!f)
\end{aligned}
$$

このプロセスは、まずポート A' に来た演算の要求 op を受けとる。その演算内容が単項演算なら X からデータ要求を出し、二項演算なら X と Y からデータ要求を出して x または x,y にデータを格納する。さらに演算 uop または bop による結果をポート Z から送信する。f は ALU 制御モジュール内部で持つフラグレジスタで、ポート F に来る要求により、外部に送信する。

このプロセスの構文を、文法にしたがって解析した結果は図 9 のようになる。この中の点線で囲まれる部分が、それぞれ一つの基本プロセスとなる。基本的には、; に対応するプロセスは直列実行プロセスであり、[ ] を含むプロセスは条件選択プロセスなどである。:= については、全体を能動データ受信、受動データ送信、空プロセス三つのプロセスとして扱う。

このように分解したプロセスを全部並べると以下のようになる。プロセスを分割する際には分割されたプロセス間に新たに通信路を導入するが、A01 などの新しいポート名は、それらの通信路が接続されるポートを示す。

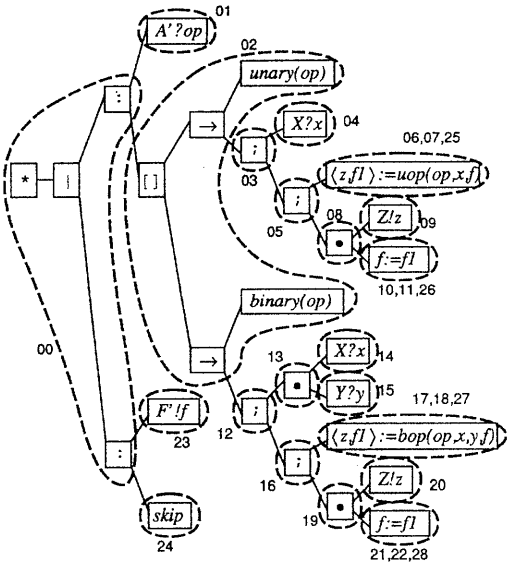


図 9: ALU 制御モジュールの構文解析木

$$
\begin{aligned}
ALU00 &\equiv *(A01' : A02 | F23' : F24) \\
ALU01 &\equiv *(A'?op : A01) \\
ALU02 &\equiv *(A02' :[unary(op) \rightarrow A03 \\
&\qquad \| binary(op) \rightarrow A12]) \\
ALU03 &\equiv *(A03' : A04; A05) \\
ALU04 &\equiv *(A04' : X?x) \\
ALU05 &\equiv *(A05' : A06; A08) \\
ALU06 &\equiv *(A06' : A07?\langle z, f1\rangle) \\
ALU07 &\equiv *(A07'!uop(op, x, f) : A25) \\
ALU08 &\equiv *(A08' : A09 \bullet A10) \\
ALU09 &\equiv *(A09' : Z!z) \\
ALU10 &\equiv *(A10' : A11?f) \\
ALU11 &\equiv *(A11'!f1 : A26) \\
ALU12 &\equiv *(A12' : A13; A16) \\
ALU13 &\equiv *(A13' : A14 \bullet A15) \\
ALU14 &\equiv *(A14' : X?x) \\
ALU15 &\equiv *(A15' : Y?y) \\
ALU16 &\equiv *(A16' : A17; A19) \\
ALU17 &\equiv *(A17' : A18?\langle z, f1\rangle) \\
ALU18 &\equiv *(A18'!bop(op, x, y, f) : A27) \\
ALU19 &\equiv *(A19' : A20 \bullet A21) \\
ALU20 &\equiv *(A20' : Z!z) \\
ALU21 &\equiv *(A21' : A22?f) \\
ALU22 &\equiv *(A22'!f1 : A28) \\
ALU23 &\equiv *(F'!f : F23) \\
ALU24 &\equiv *F24' \\
ALU25 &\equiv *A25' \\
ALU26 &\equiv *A26' \\
ALU27 &\equiv *A27' \\
ALU28 &\equiv *A28'
\end{aligned}
$$

4.2 基本プロセスの共有

図9を見ると、同じような構造が複数現れる部分があることがわかる。例えば元の ALU のプロセス記述中には $Z!z \bullet f := f1$ という部分が二箇所に現れているが、これに対応する解析木 ($ALU08$ 以下と $ALU19$ 以下) が同じ構造となって現れる。ハードウェア上これらを共有すると、ハードウェアの量を小さくすることが可能となる。プロセス記述の上で共有の操作を行うには、受動ポートの名前が異なる以外、すべての動作が同一であるような二つのプロセスの組を探す。この一方を削除し、その上位のプロセスにもう一方を共有させることで、プロセスの共有ができる。この操作によってさらに同一のプロセスが生まれることがあるので、新たな共有ができなくなるまで繰り返しこの操作を行わなければならない。

前節の例を用いて具体例を示す。まず、プロセス $ALU26$ と $ALU28$ がこの共有の条件に当てはまる。つまり受動通信 $A26'$ と $A28'$ 以外の動作が同一である。そこで、$ALU28$ を削除するが、その上位のプロセス $ALU22$ は次のように変わる。

$$ALU22 \equiv *(A22'!f1 : A26)$$

ここで新たに $ALU11$ と $ALU22$ の共有ができるようになったので、$ALU22$ を削除し、$ALU21$ を次のように変える。

$$ALU21 \equiv *(A21' : A11?f)$$

このような操作を繰り返し、最終的に $ALU14, 19, 20, 21, 22, 27, 28$ が削除され、以下のプロセスが修正される。

$$ALU13 \equiv *(A13' : A04 \bullet A15)$$
$$ALU16 \equiv *(A16' : A17; A08)$$

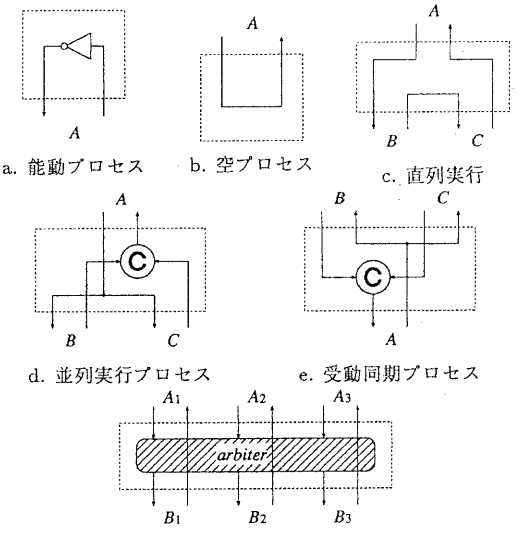
ただし、ここで注意しなければならないのは、このような操作で見かけ上プロセスの数が減っても、実際には通信路をマージするためのハードウェアが必要であり、全体としてハードウェア量が増えてしまう可能性もあることである。各プロセスを実現するためのハードウェア量を適切に評価して、増える場合はその共有をとりやめるということが必要である。

また、一つの並列実行プロセスの下位で、並列に実行されるプロセスの共有をするには、その順序関係が不定となるのでアービタを導入するなどさらにハードウェアが必要になる。一般には並列に実行される可能性のあるプロセスは共有を行うべきではない。ただし、並列に同じ動作のプロセスが実行されるのは、同じ変数からの読み出しの場合のみである。一つの変数への書き込みが並列に行われるのは、ハードウェアの動作として無意味であり、元の記述が誤っている可能性が高い。

4.3 基本プロセスの実現

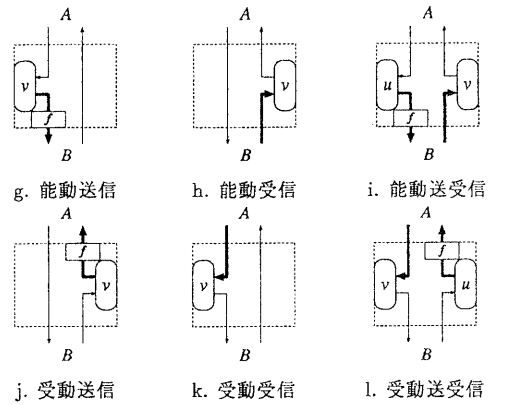
基本プロセスに対応する回路による実現を基本回路と呼ぶ。表2の15個の基本プロセスのうち、多くは非常に単純な回路または配線が基本回路となる。これらは図10となる。

さらに、変数のデータをポートによって直接やりとりするプロセスは図11の回路で実現できる。



a. 能動プロセス　b. 空プロセス　c. 直列実行
d. 並列実行プロセス　e. 受動同期プロセス
f. 調停プロセス

図 10: 単純な基本回路



g. 能動送信　h. 能動受信　i. 能動送受信
j. 受動送信　k. 受動受信　l. 受動送受信

図 11: 変数による通信を含む基本回路

これらの図中、v などは変数を実現したレジスタの一部である。変数は複数のプロセスで共有されるため、レジスタは複数の基本回路にまたがって存在することになる。レジスタ自体は図12で示される外観を持ち、

$$*(W_1'?v|W_2'?v|R_1'!v|R_2'!v)$$

といった動作をする。ただし書き込みや読みだし要求が重なった場合の動作は、アービタによる単純な調停ではない。複数の読みだし動作は同時に実行することが可能で、読み出しの実行中は書き込みを受け付けない。複数の書き込み要求が同時に来ることはないものとする。

それ以外のプロセスの内、条件選択(図13)と条件反復(図14)は自掃モジュールを内蔵している。これは、プロセスの中に変数へのアクセスがあり、プロセス間に自掃モジュールを挿入する方法では変数へのアクセスに休止相のオーバーヘッドが現れてしまうためである。ただしここで用いられている

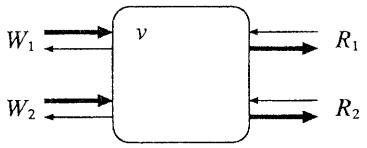


図 12: レジスタの外観

自掃モジュール (ASMn) は 1 out-of n 符号による応答信号を扱うように拡張したものである。

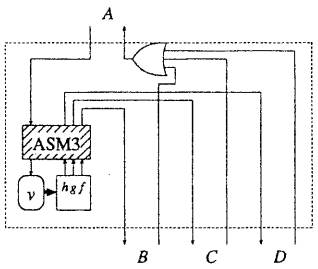


図 13: 条件選択プロセス

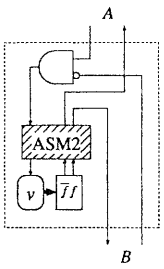


図 14: 条件反復

さらに特殊なプロセスとして、ノンブロッキングプロセス (図 15) がある。B と C の直列実行の制御の他に、C の実行中に A_1 や A_2 に来る要求を待機させておくための回路が必要になる。ただしこれは調停プロセスの直下にしか接続されず、複数の要求が同時にやってくることはないので、アービタは用いる必要がない。

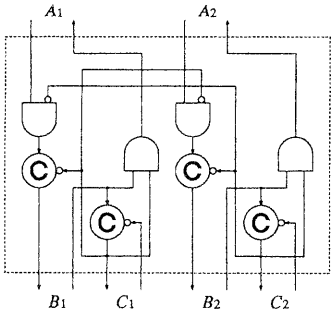


図 15: ノンブロッキングプロセス

4.4 基本回路の接続

以上のように各基本プロセスを基本回路に変換した後、これらを接続するには、通常は同名のポートの要求信号と応答信号を結合する。単一の信号の代わりにデータを用いている時も同様である。

しかし、前節の基本プロセスの共有を行った場合、または、行わなくても同名のポートに対する能動通信が複数個ある場合は、これを一つの受動ポートが受けられるようにマージする必要がある。

単純な OR では、応答を返した直後に別の要求が到着するという場合に正常に動作しない。そこで図 16 のような回路を用いる。これは A か B に来た要求を C に出しその応答はすぐ返す。その直後にもう一方に要求が到着する可能性があるが、これは待機させて C での休止相を先に完了させる。データの流れる通信路をマージする場合も、これを拡張した回路を用いなければならない。

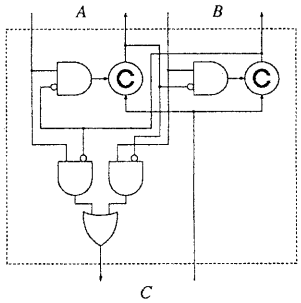


図 16: 複数の要求のマージ

最後に、休止相オーバーヘッドをなくすために、基本回路間に自掃モジュールを挿入する。その挿入すべき位置の候補は、まず能動データ通信を扱うプロセスの直前である。データ通信は一般に演算などの動作やレジスタへのアクセスを伴い、非常に時間がかかる。これは稼働相だけではなく休止相も同様で、その休止相のオーバーヘッドを自掃モジュールによって隠すことは非常に効果がある。その他の位置に自掃モジュールを挿入するのは、その通信路での休止相が時間がかかると判断される場所である。これは一般にはゲート遅延や配線遅延の総合的な評価によって、その判断を行う必要がある。4.1 節で示した例の場合は、休止相に時間がかかる場所はデータ通信の部分だけと判断でき、そこにしか自掃モジュールは挿入しない。

最終的に ALU 制御モジュールの回路は、図 17 に示すように合成される。斜線のブロックは自掃モジュールである。

5 むすび

以上、休止相のオーバーヘッドを最小にすることによる非同期式回路の高速化手法を提案し、プロセス記述から回路への変換方法、およびこの方法への高速化のためのモジュールの導入を述べた。

今後、本方式の計算機上への実装を行い、これによってある程度の規模のプロセッサを合成して、有効性を実証したい。また十分な最適化や、決まりきった回路のライブラリ化などを検討していきたい。

本研究の一部は (財) 大川情報通信基金 1991 年度研究助成によるものである。

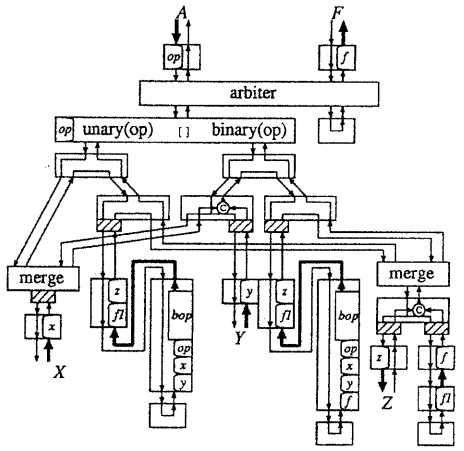


図 17: ALU 制御回路

参考文献

[1] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, Vol. 21, No. 8, pp. 666–677, August 1978.

[2] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *28th ACM/IEEE Design Automation Conference*, pp. 302–308, 1991.

[3] Kuan-Jen Lin and Chen-Shang Lin. Automatic synthesis of asynchronous circuits. In *28th ACM/IEEE Design Automation Conference*, pp. 296–301, 1991.

[4] Alain J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, Vol. 20, pp. 125–130, 1985.

[5] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *FORMAL METHODS FOR VLSI DESIGN*, chapter 6, pp. 237–283. Elsevier Science Publishers B. V., 1990.

[6] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Vol. 8, No. 11, pp. 1185–1205, November 1989.

[7] C.H. van Berkel and Ronald W.J.J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *IEEE International Conference on Computer Design*, pp. 157–162. IEEE Computer Society Press, 1988.

[8] 南谷崇. 同期式プロセッサの限界と非同期式プロセッサの課題. 信学技報, December 1990. FTS90-45.

[9] 籠谷裕人, 土居仁士, 南谷崇. 非同期式プロセッサ制御回路合成の一手法. 情報処理学会第 42 回全国大会 講演論文集 (6), pp. (6)144–(6)145. 情報処理学会, 1991.

付録

A 記述言語の拡張 BNF 定義

⟨PROC⟩ ::= [⟨NAME⟩ "="] ⟨LOOP⟩

⟨LOOP⟩ ::= "*" ⟨COMM⟩
| "*" "(" ⟨ACTIONS⟩ ")"

⟨COMM⟩ ::= ⟨PCOMM⟩
| ⟨ACOMM⟩

⟨ACTIONS⟩ ::= ⟨ACTION⟩ { "|" ⟨ACTION⟩ }

⟨ACTION⟩ ::= "(" ⟨ACTION⟩ ")"
| ⟨PACT⟩
| ⟨PACT⟩ ":" ⟨AACT⟩
| ⟨PACT⟩ ";" ⟨AACT⟩
| "(" ⟨PACT⟩ ":" ⟨AACT⟩ ")" ";" ⟨AACT⟩

⟨PACT⟩ ::= "(" ⟨PACT⟩ ")"
| ⟨PCOMM⟩ { "•" ⟨PCOMM⟩ }

⟨PCOMM⟩ ::= ⟨PORT⟩ "'" [⟨DCOMM⟩]

⟨AACT⟩ ::= ⟨AACT1⟩ { ";" ⟨AACT1⟩ }

⟨AACT1⟩ ::= ⟨AACT2⟩ { "•" ⟨AACT2⟩ }

⟨AACT2⟩ ::= ⟨ACOMM⟩
| ⟨OP⟩
| "(" ⟨AACT⟩ ")"
| "[" ⟨GCS⟩ "]"
| "{" ⟨GC⟩ "}"

⟨ACOMM⟩ ::= ⟨PORT⟩ [⟨DCOMM⟩]

⟨DCOMM⟩ ::= "!" ⟨EXPRS⟩
| "?" ⟨VARS⟩
| "!" ⟨EXPRS⟩ "?" ⟨VARS⟩

⟨OP⟩ ::= ⟨VARS⟩ ":=" ⟨EXPRS⟩

⟨VARS⟩ ::= ⟨VAR⟩
| "<" ⟨VAR⟩ { "," ⟨VAR⟩ } ">"

⟨EXPRS⟩ ::= ⟨EXPR⟩
| "<" ⟨EXPR⟩ { "," ⟨EXPR⟩ } ">"

⟨GCS⟩ ::= ⟨GC⟩ { "‖" ⟨GC⟩ }

⟨GC⟩ ::= ⟨EXPR⟩ "→" ⟨AACT⟩