設計自動化 68－22
(1993. 10. 29)

2相式非同期回路高速化のための基本制御モジュールとその応用

籠谷裕人　　南谷 崇

東京工業大学 工学部
〒152 東京都目黒区大岡山 2–12–1

あらまし 非同期式回路のデータパスを稼働相と休止相の2相で動作させる場合に、次の動作に進む前に本来の機能ではない休止相の完了を待つことは無駄である。本稿では休止相を待たずに次に進むことを許す基本制御モジュールを提案し、従来の制御モジュールをこれに置換することによる高速化を議論する。単純な置換ができない箇所は、依存性グラフの解析により明らかになる。このような箇所においても、付加的な回路を使用することで、ゲートの遅延に依存せずに休止相のほとんどを隠蔽することが可能になる。論理シミュレーションにより、本手法でゲート数の増大なしに大幅な速度向上が可能なことが実証された。

和文キーワード 非同期式回路, 依存性グラフ, 2相制御, 休止相, 高速化

On Handshake Control for 2-phase Delay-insensitive Processors

Hiroto Kagotani　　Takashi Nanya

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

Abstract In 2-phase delay-insensitive circuits, approximately a half of the processing time is wasted by the second phase called idle phase that does not perform any actual operation. We propose a handshake controller that enables next operations to start without waiting for the completion of the idle phase so that we can reduce the processing time. Replacing a conventional control module simply with the new module is not always allowed because of the dependencies. We solved this problem by using additional AND gates. Logic simulation shows that this method can improve the throughput of circuits without increasing the number of the gates.

英文 key words delay-insensitive circuit, dependency graph, 2-phase control, idle phase concealment

1 まえがき

近年の素子技術はスイッチング遅延が1ピコ秒にせまる高速なデバイスを実現しつつある。しかし従来の同期式プロセッサ回路はチップ全体へのクロック分配が必要であり、配線遅延の相対的な増大によるクロックスキューのため、こうした素子を活用できるような高速のクロックを用いることができない[4]。

素子の高速性を有効に活用する一つの方法は、プロセッサを非同期式回路で構成することである。非同期式回路は、同期式回路の設計にあるような論理設計とチップ設計の相互依存性を排除でき、また、回路を拡張する場合のタイミング設計のやり直しも不要となり拡張性に富むといった利点を持つ。

我々は文献[6]で、2相式非同期回路の合成手法を提案した。次節でその概要について触れ、3節では2相のひとつである休止相を隠蔽するためのモジュールとその使用可能な箇所を述べる。4節ではすべてのハンドシェークをこのモジュールで制御するための手法を提案し、最後にこれによる速度向上の評価を行う。

2 依存性グラフを仕様とする回路合成

2.1 回路モデル

非同期式回路は、使用する回路要素の遅延がある範囲で変動しても正しく動作しなければならない。本稿ではその遅延として、回路のすべての論理ゲートや配線に、有限ではあるが上界が未知の遅延の存在を仮定する(Delay Insensitivity)。しかしこれだけでは実用上回路が構成できないので[2]、配線の分岐がある場合には各分岐先への遅延が等しいものとする(Isochronic Fork)。

システムの構成は、図1のように制御部とデータパスから成る。制御部はハンドシェークの要求信号によって処理の開始を指示し、データパスは応答信号によって処理の完了を通知する。一般に、この一つのハンドシェークで制御される処理はレジスタ間のデータ転送であり、その転送途中にデータの加工も行われる。このような処理の単位を本稿では基本操作と呼ぶ。

2.2 基本操作の2相動作

一つの基本操作に対応するハンドシェークは図2によって実現される。ハンドシェークの制御回路側にある箱は、ハンドシェークを制御するためのモジュール(2相制御モジュール)と呼ぶ)の一種で、Qモジュール(Q-element[3])と呼ばれるものである。このモジュールの回路に使用されている©はMullerのC素子と呼ばれ、全入力値が一致した時にのみ、出力がその入力値に等しくなるという記憶素子である。

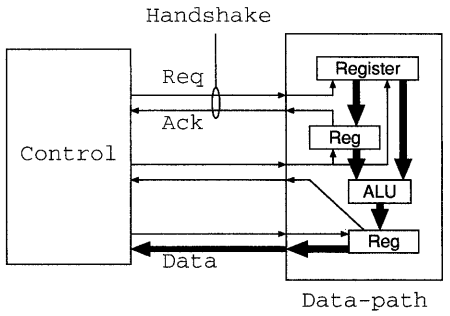


図1: 制御回路とデータパス

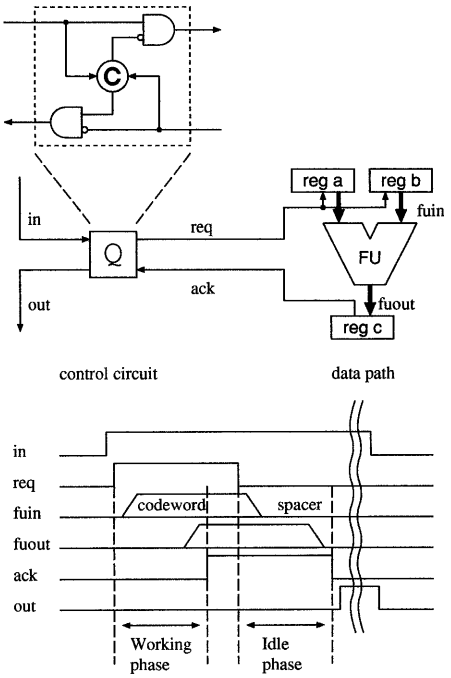


図2: 基本操作の実現

基本操作の開始条件が整うと、2相制御モジュールを通して要求信号reqが出される。この信号はそのままレジスタa,bに対するデータ出力要求となり、出力されたデータは演算回路で加工されるなどして、転送先レジスタcに到着する。到着したデータが正しく格納されるとレジスタcは応答を返すが、この信号がそのままハンドシェークの応答信号ackとして使われる。ここまでの相を稼働相(Working

phase) と呼ぶ。

さらにこのハンドシェークやデータパスを次に使う時のためにクリアする必要がある。要求信号を 0 にすると、レジスタ a,b からのデータ出力が止まり、転送先レジスタ c が応答信号を 0 にする。この期間を休止相 (Idle phase) と呼ぶ。

以上のように、転送先レジスタではデータが到着しているかどうかを判定して応答信号を生成するので、データ信号を 2 相符号化 [5] しなければならない。2 相符号とは、信号線がすべて 0 の状態 (スペーサ) となんらかの符号語である状態とを交互に繰り返す方式である。構成の単純な 2 線 2 相符号では、符号語として 2 線式符号を用い、1 ビットのデータごとに 2 本の信号線を使用する。

この符号を用いる 2 ビットレジスタは図 3のように実現できる。このような方式のレジスタは、構造が単純ではあるが、読み出し中に書き込みが起きないことを制御回路によって保証することが必要である。

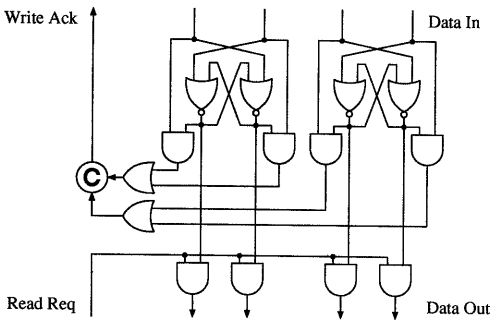


図 3: 2 線 2 相方式の 2 ビットレジスタ構成例

2.3 依存性グラフ

本稿で用いる回路合成の仕様記述方式である依存性グラフとは、システムで実行される基本操作間の依存性を有向グラフで表現したものである。依存性とは二つの基本操作をある特定の順序で実行しなければならないという制約である。一方の基本操作が他方の結果を利用する場合や、基本操作間で演算回路を共有する場合などに依存性が発生する。

例えば、図 4は、あるプロセッサでの命令フェッチの動作を表す依存性グラフである。丸く囲まれた代入文は基本操作を表す。基本操作 A と C は並列に実行することができるが、D を実行するには A と C の完了を待たなければならないことなどが記述されている。小さな○で表された並列な動作への分岐のノードをフォーク、待合せのノードをジョインと呼ぶ。

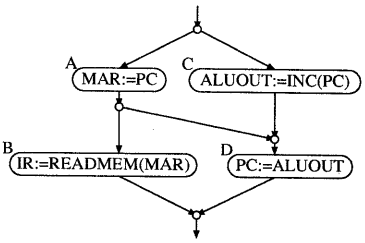


図 4: 命令フェッチ動作の依存性グラフ

また図 5は、加算とシフト演算だけを用いて乗算を行う回路を記述した依存性グラフの例である。この中で三角形で示した条件分岐を行うノードをセレクト、逆三角形の合流を行うノードをマージと呼ぶ。これらのノードを使用すると、条件判断やループを記述することができる。

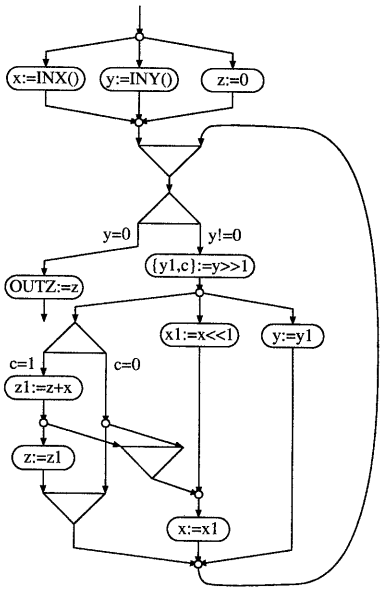


図 5: 乗算回路の依存性グラフ

以上のような記述要素で記述された依存性グラフは、アーク上のトークンの移動によって意味づけされる。すなわち各ノードの上流アークにトークンが到着して発火条件が整うと、発火して下流のアークにトークンが移動する。不正に基本操作が並列に動作してしまったり、トークンが一つのアーク上に 2 つ以上存在したりしないようなグラフを正しい依存性グラフとする。

2.4 依存性グラフからの回路の合成

依存性グラフ上では、各基本操作の開始タイミングはトークンの移動によって制御されている。そこで合成される制

御回路でも、トークンの移動を回路で実現し、これによってハンドシェークの開始タイミングを制御するという方式が使える。そのためには各アークを配線で実現し、その配線での $0 \rightarrow 1$ 変化をトークンとみなす。さらに各ノードを対応した回路ブロックで置き換える。

基本操作ノードに対応する回路は、入力信号の $0 \rightarrow 1$ 変化によって対応するハンドシェークを完了させ、出力信号を $0 \rightarrow 1$ に変化させる。これはQモジュールの動作そのものであり、基本操作ノードはQモジュールで実現できることがわかる。さらにフォークノードは単純な配線のフォークで、ジョインノードはMullerのC素子で実現できる。

セレクトノードでは変数の値を参照して演算を行い分岐する。すなわちこれに対応する回路では、必要なレジスタにデータを要求し、1-out-of-n符号を出力する回路で演算を行う。しかし演算中や分岐中にそのレジスタの内容が書き変わる可能性のある場合には、別のレジスタを用意して一旦これにデータをコピーし、それを使用して演算する必要がある。これらに対応する回路は図6のようになる。

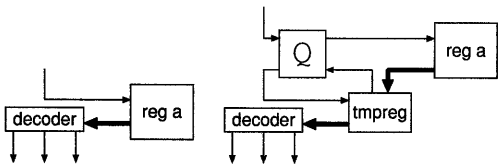


図 6: 2種類のセレクト回路

マージノードは条件分岐後の合流として使用される場合と、ループ後の合流として使用される場合がある。前者の場合は対応する回路はORゲートで実現できる。しかし後者の場合は一旦ループ全体を0でクリアしてから再度トークンにあたる $0 \rightarrow 1$ を起こす必要があるため、一方にインバート入力のあるANDゲートとなる。一般にはマージノードには二つ以上の入力があり、それぞれの入力の種類も異なりうる。そこでこれらの違いによらず一意に回路に変換するためにはEXORゲートを用いる。正しい依存性グラフから生成した回路では、それぞれの入力が同時に変化することはないのでハザードの心配はない。

以上をまとめると図7のように、各ノードと回路ブロックが対応づけられる。

データパスについては、必要なレジスタと演算回路を各基本操作に対応するように接続する。演算回路は別途用意されているものとする。使用する変数や演算の等しい基本操作が複数ある場合、それらのパスを別々に用意することはハードウェア量の増大を招くので、図8のような回路(HMM — handshake merge module と呼ぶ)で共有する。さらに同時には使用されない同一の演算を行う回路はマルチプレクサなどによって共有するが、本稿では詳しくは触れない。

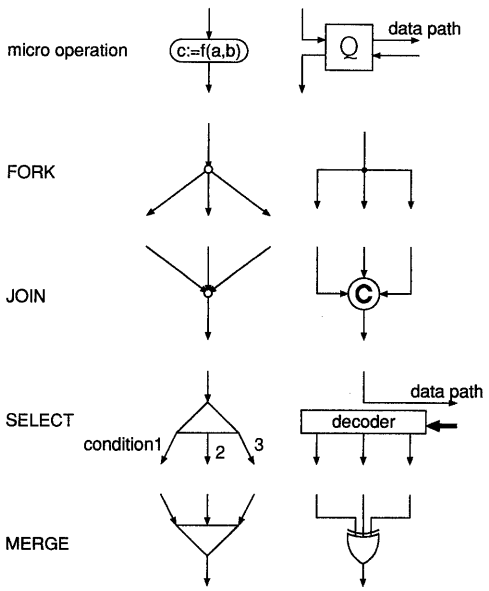


図 7: 依存性グラフのノードと回路ブロックとの対応

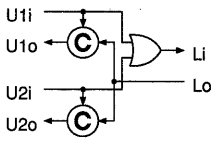


図 8: ハンドシェーク・マージ・モジュール (HMM)

3 休止相の隠蔽による高速化

3.1 休止相による速度低下

先に述べたように、休止相は使い終わった回路をクリアするための相であり、機能の遂行に必要な演算を行う期間ではない。Qモジュールを用いた2相式のデータ転送は、回路構成が単純であり、一回の動作ごとに各ハンドシェークの二つの相の終了を保証するため、期待された動作を安全に実行することができる。しかしその反面、この休止相が終了するまで次の動作に進めないという欠点がある。

稼働相にかかる時間と休止相にかかる時間は、演算回路の構成にもよるが、だいたい等しいと考えることができる。すなわちQモジュールを用いる方式では、稼働相と休止相の時間の和が全プロセスの実行時間となるため、全実行時間の約半分が休止相に費やされることになる。

3.2 休止相の隠蔽

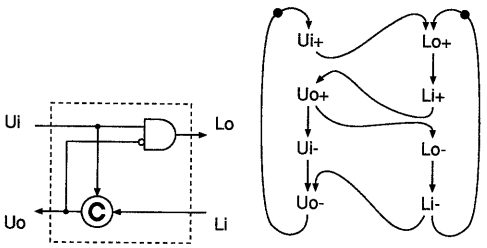
休止相による動作速度低下を抑えることは、ある基本操作の休止相を次の基本操作の稼働相と並列に実行させることで実現できる。稼働相が終わったら休止相を開始するとともに、すぐに次段をイネーブルするような2相制御モジュールを、Qモジュールの代わりに用いる。これにより、全プロセスの実行時間に休止相が影響することを避けることができる。さらに休止相を短くするための演算回路の工夫などが不要になるため、稼働相を短くしたり回路量を減らすという本来の最適化が行いやすくなると期待できる。

以上のような動作を実現するための2相制御モジュールとして、図9(a)の回路で示される自掃モジュール (auto-sweeping module: ASM) を提案する。ASMは上位の入出力対 (Ui, Uo) と下位の入出力対 (Lo, Li) を持ち、その動作は信号遷移グラフ (STG[1]) で図9(b)のように表される。このASMの動作は以下のようになる。

1. 基本操作の実行がイネーブルされると $(Ui+)$、基本操作の稼働相を開始する $(Lo+)$。

2. 稼働相が完了すると $(Li+)$、次段の基本操作をイネーブルする $(Uo+)$ とともに、休止相を開始する $(Lo-)$。

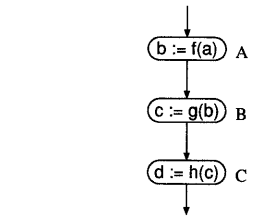
3. 休止相が完了し $(Li-)$、さらに上位の入力が元に戻ったら $(Ui-)$、上位への出力を元に戻す $(Uo-)$。



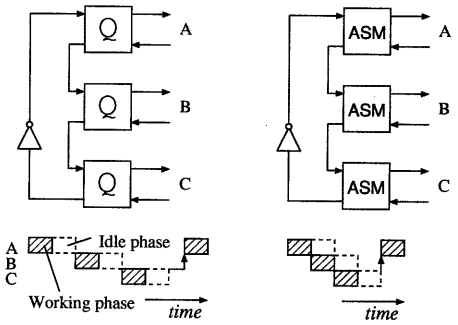
(a) 回路実現　(b) STGによる動作記述

図9: 自掃モジュール (ASM)

図10(a)は演算の連続するシステムを依存性グラフで記述したもので、これをもとに普通に合成した制御回路は図10(b)のようになるが、そこに示すタイミングチャートのように休止相によって全体の実行時間の半分が消費される。このQモジュールをASMに置換した回路が図10(c)となる。これを見ると、ASMを使用するとループ最後の基本操作を除くすべての休止相が、実行時間に影響しないように隠蔽されていることがわかる。また2相制御モジュール間を0が伝搬する時間も現われない。



(a) 依存性グラフ



(b) Qモジュールによる制御回路　(c) ASMによる回路

図10: ASMへの置換による高速化

3.3 ASM使用箇所の制限

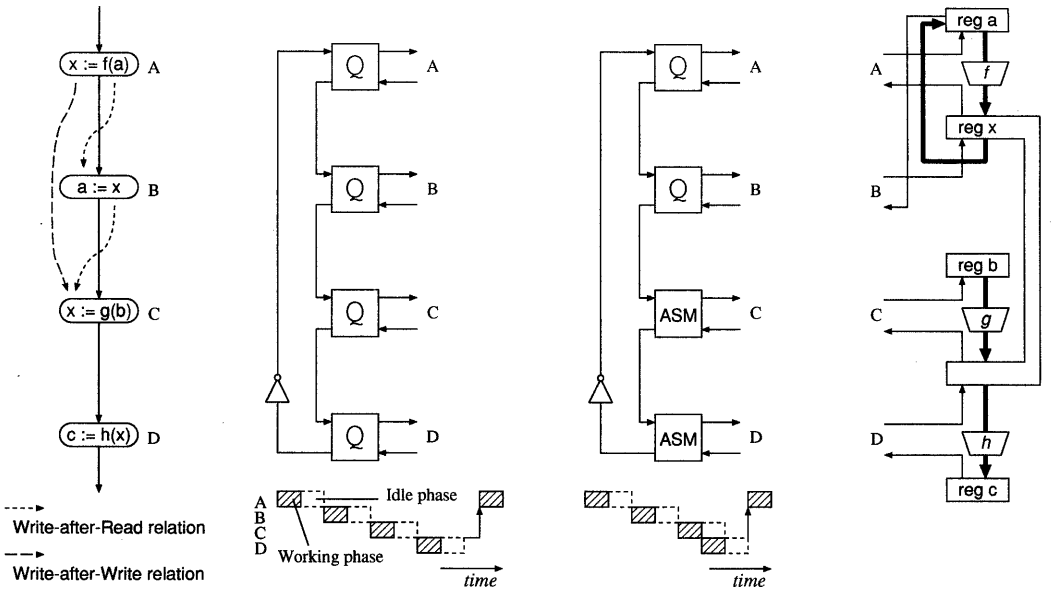
以上のように、ASMの使用により大幅な実行時間の短縮が可能となる。しかしながら、ASMはどのQモジュールとも単純に置き換えて使用できるわけではなく、そのような置き換えが可能な場所には制限がある。

図11(a)は図10よりは複雑な演算を行う依存性グラフであり、普通にQモジュールで実現した制御回路が図11(b)、データパスが図11(d)となる。

ここで仮に基本操作Bに対応するQモジュールをASMに置き換えたとする。そうするとASMのANDゲートの遅延によっては、レジスタ x に対する読み出し要求が0になる前に、次のCで x への書き込みが行われることがある。図3に示したレジスタを用いると格納されたデータは読み出し要求が1であるかぎり出力され続けるので、レジスタ a に異常なデータが到着し誤動作となる。

また別の場合として、基本操作Aに対応するQモジュールをASMに置き換えたとする。その場合レジスタ a や演算回路 f の遅延によっては、f の出力が符号語のまま、Cの基本操作により符号語が別の経路から x に到着することがある。これによって x には異常なデータが書き込まれ誤動作となる。

上で考察した二つの誤動作は、ASMを使用したために起きるが、そのメカニズムは異なる。それぞれを整理すると、図11(a)のような破線や点線による有向枝で表される基本



(a) 依存性グラフ　(b) Q モジュールのみの制御回路 (c) 最大限 ASM に置換した回路　(d) 制御対象のデータパス

図 11: ASM 置換可能な場所の特定

操作間のある種の依存関係と対応していることがわかる。

- 点線: write-after-read 関係。ある基本操作と、そこで読み出したレジスタの値を最初に更新する下流基本操作との関係
- 破線: write-after-write 関係。ある基本操作と、そこで書き込んだレジスタの値を最初に更新する下流基本操作との関係

ただしこれらの関係はループの終了点や、プロセスの終点と始点をまたぐことはない。ループの終了点やプロセスの終点まで動作が進むと、次は EXOR ゲートや NOT ゲートによってループ全体やプロセス全体を 0 でクリアする。ASM の動作を図 9(b) の STG で見ると、0 でクリアされて $Uo-$ となる時には、$Li-$すなわち休止相の完了が保証されていることがわかる。例えば図 11の基本操作 C を ASM で実現しても、休止相の完了以前には A がイネーブルされないので、x に異常なデータが書き込まれることはない。

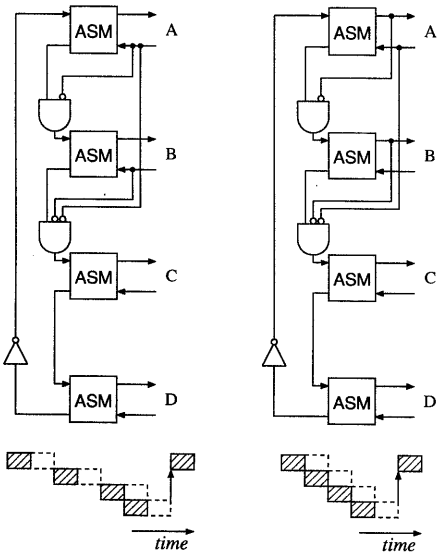
以上のように制御回路の合成の元となった依存性グラフの上で、write-after-read や write-after-write 関係をすべて探し出すと、それを元に、どの基本操作で Q モジュールの代わりに ASM が利用できるかが簡単に判別できる。すなわち、ある基本操作からいずれかの関係を表す有向枝が伸びているなら、そこでは ASM に置換できない。図 11の例では、ASM に置換できるのは図 11(c) のように C と D の基本操作だけである。

4　全基本操作での ASM の利用

前節では、Q モジュールを ASM に置換できない箇所について明らかにした。置換できない理由を繰り返すと、一度使用したレジスタは、その基本操作の休止相終了前に更新するわけにいかないということである。つまりなんらかの方法でレジスタの更新前に休止相の終了を確認できる場合には、そこに ASM を利用できる。

ある基本操作での休止相終了の確認は、それに対応する 2 相制御モジュールの下位入力 (Li) を監視して、これが 0 であることを調べれば可能である。具体的には、write-after-read や write-after-write 関係の有向枝の到着地点にあたる基本操作では、その 2 相制御モジュールの上位入力 (Ui) の前に抑制入力付き AND ゲートを挿入する。さらにその有向枝の出発地点にあたる基本操作の 2 相制御モジュールの下位入力 (Li) を、その抑制入力に接続する。この手順を全部の有向枝について行うと、全基本操作は ASM で実現することができる。図 11の例にこの手法を適用すると図 12(a) のようになる。

しかし以上の手法によって ASM を使用しても、Li の監視では高速化につながらない場合が多い。特に図 12(a) では、基本操作 A,B も ASM で実現されているが、それぞれの直後の基本操作を実行する前に $Li = 0$ を確認しているため、まったく高速化されない。ASM と直後の Li 監視 AND ゲートのペアは実は Q モジュールそのものである。



(a) Li のみによる監視　(b) Lo も利用した高速化

図 12: AND ゲート付加による全基本操作の ASM 実現

write-after-read 関係の場合は、参照していたレジスタが不正なデータを出力しないならば、別の基本操作でこれを更新しても問題はない。レジスタを実現する回路に依存するが、図3のような出力部を持つレジスタならば、出力要求信号が 0 であることを保証すれば更新によって不正なデータが出力されることはない。そこで完全に休止相が完了することを確認せずとも、$Lo = 0$ で休止相が開始されたことを確認すれば、ASM をそのまま使用できることがわかる。このようにして監視位置を変更した回路が図 12(b) である。これによってようやく図のタイミングチャートのように A,B,C の休止相を隠蔽することが可能となる。

ただし、HMM を使用したハンドシェークのマージがあると、OR ゲートが挿入されるため、Lo の監視ではレジスタ出力要求が 0 になったことを保証できない。このような場合には Li の監視でしか、ASM を利用する方法はない。

さて、監視先基本操作 (例えば図 12(b) の A) の休止相の確認も終えてある基本操作 (同図の B) のハンドシェークを実行している時に、突然 A が稼働相に移ってしまうことはありうるだろうか。もしこれがあると、B の ASM の上位入力 (Ui) がプロトコルを無視して 0 になってしまい、誤動作が起きる。しかしながら、この状況は A と B が並列に動作しうる関係にあることを意味する。B で更新するレジスタを A で参照または更新していたためにその監視が必要になったわけであり、もともとこの二つが並列に動作しうるような回路の依存性グラフは正しくなかったことになる。結論として、正しい依存性グラフから生成された回路では、このような誤動作の心配はない。

5 高速化の評価

図 5に示す乗算アルゴリズムに従って、$8bit \times 8bit \rightarrow 16bit$ の乗算回路の合成を行い、ASM の利用による高速化の度合をシミュレーションによって測定した。$16bit$ の加算器はリップルキャリー方式を用いた。論理ゲートは 4 入力までを 1 段として扱い、それ以上は多段に構成する。C 素子は NAND ゲート 2 段によるフィードバックで実現した。すべての論理ゲートは同一の遅延を持つと計算し、これを 1 単位時間とした。

表 1: 乗算回路の速度比較

制御回路方式	サイクルタイム	標準偏差	速度比
Q のみ	1040.0	189.2	1
Q と ASM	923.6	168.1	1.13
ASM と AND	599.5	108.9	1.73

表 1に、各制御回路の方式による測定結果を示す。入力データには 1000 組のランダムな数値を用いた。3 節で述べた一部 Q モジュールの ASM への置換だけでは、ほとんど速度は向上しない。しかし 4 節に示すように休止相を監視する AND ゲートの付加によって、全基本操作を ASM で実現すると、約 1.73 倍という高速化が達成できた。しかも元の回路は約 1220 ゲートからなるが、これと比べてゲート数はほとんど変化しておらず、むしろ減少している。ASM の導入は回路の性能向上に大きな効果があることがわかった。

6 むすび

依存性グラフを仕様として合成した非同期式回路において、2 相方式の欠点であった休止相の問題を解決する 2 相制御モジュール ASM を提案した。また基本操作間の write-after-read, write-after-write の関係を用いて、Q モジュールを単純に ASM で置換する際の制限を明らかにした。

さらに単純な置換ができない場合にも、AND ゲートによって必要な監視を行えば ASM が利用できることを示した。乗算回路を例に示したように、このようなシンプルな手法で、ゲート数を増加させることなく効果的に高速化を図ることができる。

今後、調停動作によるアービタと ASM の関連について検討していく必要がある。

本研究の一部は文部省科学研究費補助金 04452192 によるものである。

参考文献

[1] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. International Conf. Computer Design (ICCD)*, pp. 220–223. IEEE Computer Society Press, 1987.

[2] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pp. 263–278. MIT Press, 1990.

[3] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, pp. 237–283. North-Holland, 1990.

[4] Takashi Nanya. Challenges to dependable asynchronous processor design. In *Proc. Int. Symp. on Logic Synthesis and Microprocessor Architecture*, pp. 132–139, July 1992.

[5] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[6] 籠谷裕人, 南谷崇. 依存性グラフに基づく非同期式制御回路の合成. 信学技報, October 1992. FTS92–16.