卒業論文

プロトコル表現効率化に基づく
プロトコル変換器合成手法に関する研究

平成 18 年 2 月 13 日 提出

指導教員 藤田昌宏 教授

電子情報工学科

40368　石川　悠司

目次

1 **研究の背景** 2

1.1 設計生産性の危機と IP ベース設計手法 2

1.2 IP ベース設計の問題点 3

1.3 本研究の目的 4

2 **既存の研究** 5

3 **提案手法** 6

3.1 ラッパ生成手法の概要 6

3.2 対象とする IP インターフェースプロトコル 7

3.3 プロトコルの表現方法 8

3.4 バッファ 12

3.5 ラッパコントローラの生成手法 15

4 **実験** 23

4.1 ラッパコントローラ生成ツールの実装 23

4.2 例題 24

4.3 実験の条件 24

4.4 実験結果 24

5 **本研究のまとめ** 25

6 **今後の課題** 27

6.1 ラッパ生成手法の実装 27

6.2 異なるクロック間の変換を行うラッパの生成 27

6.3 プロトコル表現法の拡張 27

1 研究の背景

1.1 設計生産性の危機と IP ベース設計手法

現在、半導体の集積・製造技術の進歩は著しく、半導体に集積できるトランジスタ数は年率約 58% の割合で増加を続けている。これに対し、設計者が設計できる回路の規模の伸びは年率約 21% 程度にとどまっており、製造可能なトランジスタ数に設計可能なトランジスタ数が追いつけなくなる事態が生じている [1]。そのため、半導体生産性の伸びと設計生産性の伸びのギャップを埋めるためにコンピューター設計支援 (CAD) 技術の開発と導入が積極的に進められている。

大規模 LSI を設計する際には設計生産性を上げる方法として、トップダウン的手法とボトムアップ的手法の 2 通りの手法が用いられている。トップダウン的手法とは、RTL(Register Transfer Level) より抽象度の高いレベルからシステム設計をはじめる方法である。この手法では、SpecC[2] や SystemC[3] などのシステム設計言語で設計対象全体の仕様を記述し、ハードウェア処理とソフトウェア処理の切り分けと設計の詳細化、設計の評価を繰り返して RTL 記述に至る。これに対し、ボトムアップ的手法とは、複数の機能ブロックを組み合わせることで設計対象を設計する手法である。この 2 つの設計手法は排他的なものではなく、トップダウン的手法によって分割した仕様に対して既存の機能ブロックを割り当てたり、ボトムアップ的手法で利用する機能ブロックを仕様からトップダウン的手法で生成したりというように、実際の設計では両手法を組み合わせて使用している。

ボトムアップ的手法の設計生産性を上げる方法として、既存の回路設計の中から汎用性のある部分を切り出して、別の設計で再利用できるようにするという方法がある。このような、再利用可能な設計資産は、IP(Intellectual Property) と呼ばれている。特に、CPU やメモリコントローラなど、特定の機能を提供するモジュールは IP コアあるいはコアなどと呼ばれる場合もあるが、本論文では IP、IP コア、コアは全て、再利用可能な設計資産という意味で扱う。

IP を再利用することで、LSI の全機能・全回路のうち、新規に設計しなければならないものを大幅に削減することができる。また、動作の検証を行った上で IP をパッケージ化して提供することによって、設計中に行わなければならない検証作業の量が削減できる。検証期間が大規模 LSI の設計期間に占める割合は極めて大きく、検証作業の量が削減されることは、LSI 設計全体の期間を短縮することにつながる。

ボトムアップ的手法の一例として、IP ベース設計手法が挙げられる。IP ベース設計手法では、システムの要求仕様を IP の組み合わせで実現する。IP ベース設計には、以下のような工程が含まれる [4]。

- システムレベル設計
  要求仕様をどのようにハードウェアとソフトウェアで分担するか、どのような IP を用いて実現するかを決定する。
- IP 間の通信路の設計
  システムは複数の IP から構成されるので、IP の間で通信を行う必要がある。この作業には、通信路のバスプロトコルの決定、バストポロジーの決定などが含まれる。IP 間のデータ転送帯域や、データ転送頻度などの情報をもとに、オンチップバスの構成を決める。
- 利用する IP の決定
  設計者が保有する IP ライブラリの中に候補がある場合はそれを利用する。要求を満たす IP が手元にない場合は、別の設計者が製作した IP を入手するか、要求を満足する回路を新規に設計する。

- LSI 全体の設計
  通信路の設計に従い、オンチップバスを生成する。さらに、用意した IP を設計に組み込む。
- 評価と検証
  工程の各段階で、設計が要求仕様を満たしているか、あるいは、設計に誤りが含まれていないかを検証する。

これらの作業をフィードバックをかけながら進めていき、設計対象全体の RTL 記述を生成する。

1.2 IP ベース設計の問題点

IP を再利用して回路を設計することで設計生産性が向上することは第 1.1 節で述べたが、IP を設計に再利用するためには、解決するべき問題点がいくつか存在する。

まず、他のベンダから IP を購入する際に、IP の提供方法に明確な方針が定められていないという問題がある。例えば、IP の提供形態は RTL 記述などのソフト IP の場合や、シリコン上のマスクパターンなどのハード IP の場合があり、どのような形態をとるかについては IP ベンダによって異なっている。さらに、IP の入出力仕様や性能指標を表現する方法もベンダによって異なっており、IP 再利用を妨げる原因となっている。

さらに重要な問題点として、IP のインターフェースプロトコルの互換性が挙げられる。ここで、インターフェースプロトコルとは、通信に用いる信号線及び、通信の手続きの規格である。第 1.1 節でも述べたとおり、IP ベース設計では、オンチップバスに IP を接続することで、LSI 全体の設計を行う。この際に問題になることは、組み込みたい IP のインターフェースプロトコルがオンチップバス、あるいは他の IP のインターフェースプロトコルと適合しない場合があるということである。

IP のインターフェースプロトコルは、IP の設計時に様々な要因で決定される。データ幅、アドレス空間、転送帯域、通信頻度、1 回の転送あたりのデータ量などといった IP のデータ入出力の特性は、プロトコル選択の重要な要因となる。また、積極的な再利用や、商品として提供することを目指すならば、オンチップバスのプロトコルと親和性の高いインタフェースを選択しなければならない。これらの要因をもとにして、IP インターフェースプロトコルの選択が行われる。

オンチップバスプロトコルの代表的なものとして、ARM 社が主導している AMBA[5]、IBM 社が主導している CoreConnect[6] などが挙げられる。現在、いくつかの CAD ツールベンダは、特定のオンチップバスプロトコルとツールに登録された IP ライブラリを用いて、対話的に IP ベース設計を行う支援ツールを提供している [7, 8, 9]。これらのツールでは、設計者が簡単な操作を行うだけで、ツールが設定済みのオンチップバスを生成し、自動的に IP の組み込み、論理合成を行う。ただし、論理合成を行う際に外部の論理合成ツールを必要とするものもある。

しかしながら、他のベンダが設計した IP を設計に組み込みたい場合、IP のインターフェースプロトコルが適合しない事があるため、これらのツールを用いて設計を行うことができない。インターフェースプロトコルの互換性がない IP とオンチップバスを接続するためには、プロトコルを変換するための論理回路を間に挟む必要がある。このような論理回路は、「ラッパ (Wrapper)」あるいは「トランスデューサ (Transducer)」などと呼ばれている。本論文では、以降、「あるプロトコルの信号の論理値や信号伝達のタイミングを変換して別のプロトコルにする論理回路」を指して「ラッパ」と呼ぶことにする。

ラッパの設計作業は、IP ベース設計において本質的ではないにも関わらず、労力を多く消費し、バグが入り込む余地も大きい部分である。すなわち、IP インターフェースプロトコルの多様さとそれに起因する非互換性が IP ベース設計の妨げとなっていると言える。

ここで、プロトコルの非互換性という問題を解決する方法として、

- 標準プロトコルを定義して、全ての IP が同一のインターフェースプロトコルを使用するという方法
- 異なるプロトコル間を翻訳するラッパを自動的に生成し、プロトコルの互換性のない IP を接続できるようにするという方法

の 2 つの方法が挙げられる。

前者の方法は、プロトコルを標準化することで、IP と IP の接続、あるいは、IP とオンチップバスの接続が簡単にできるようになることを目指すというものである。現在、IP ベース設計では、オンチップバスのプロトコルとして、AMBA や CoreConnect などのバスプロトコルが多く使われている。また、各 IP ベンダからもこれらのプロトコルに準拠した IP が多数提供されている。しかし、これらのプロトコルを利用する上では、利用に対してライセンス料が必要になる場合がある。また、企業が提唱するプロトコルの場合は、1 社の都合でプロトコルの仕様が大きく変わることがありえるといった問題点もある。このような問題点をもつ既存のバスプロトコルに対し、非営利団体 OCP-IP(OCP International Partnership)[10] は、ライセンスフリーのインターフェースプロトコルである OCP(Open Core Protocol)[11] を策定している。OCP は、IP の様々な入出力形態に対応できるように、スケーラビリティの高いプロトコルとなっている。さらに、非営利団体が策定しているプロトコルであるため、OCP は、誰でも自由に利用することができ、1 社の利害で仕様が変動する可能性が低いという利点がある。

後者のラッパを自動合成するという方法は、ラッパの設計作業を人手を用いずに行うことで、設計時間を短縮することを目指すというものである。ラッパの自動合成方法を確立することで、異なるプロトコルに準拠した IP を設計に組み込むことが簡単にできるようになる。また、プロトコルを標準化した場合でも、非標準プロトコルに準拠した過去の設計を利用する場合には、非標準プロトコルを標準プロトコルに変換するラッパが必要になる。このため、ラッパを自動生成する技術の重要性は極めて高いと言える。

1.3 本研究の目的

第 1.2 節で述べたように、IP のインターフェースプロトコル非互換性という問題を解決する方法の一つとしてラッパを自動生成するという方法がある。既存の IP に対してラッパを自動生成する際に考えなければならない問題の一つに、ラッパの生成手法に対してどのようにして変換対象のプロトコルの情報を与えるかという問題がある。

既存の設計記述の入出力仕様を把握する方法として、まず一つには、設計記述の RTL 記述を解読して、入出力処理に関する情報を入手するという方法がある。この方法は、膨大な設計記述の中から入出力処理に関係する部分だけを抽出してプロトコルの仕様を得ることは簡単ではないという点と、RTL 記述が提供されない IP も存在するという点で問題がある。もう一つの方法としては、IP の仕様書かプロトコルの仕様書を入手して、インターフェースプロトコルの定義を参照するという方法がある。再利用できる設計記述には必ず仕様書が付属すると仮定すれば、この方法は現実的である。

しかし、仕様書を参照する場合でも、簡単にプロトコルの仕様を把握することはできない。まず、第 1.2 節で述べたように、IP 仕様書の明確な書式が定まっていないため、同じ情報でも IP 供給者ごとに異なる書き方で記されている可能性がある。さらに、書式が統一されたとしても、信号線同士のタイミング制約を取得する作業は簡単ではない。プロトコルの制御フローを FSM の形で表現すれば、多くのプロトコルで仕様を正確に表現できるが、実際の仕様書では、FSM の形でプロトコルが表現されることはほとんどない。仕様書で

は、リードの動作、ライトの動作など、いくつかの代表的な動作のタイミングチャートを見せる場合が多いが、タイミングチャートは、特定の条件下で動作のスナップショットを取ったものであり、信号間の遷移順序やエラー処理への移行の手順などを一度に把握することは難しい。このため、プロトコルの仕様を把握する際には、断片的に与えられた情報からプロトコルの全体像を導き出さなくてはならない。しかし、この作業を人手で行う場合、推論の入り込む余地があるため、バグが入り込む可能性がある。仕様理解にバグが含まれている場合、生成されるラッパも誤ったものになるので、バグの入り込む余地が少なくなるようにプロトコルを把握・表現する方法を定義する必要がある。

本研究では、設計記述の仕様書からインターフェースプロトコルに関する情報を抽出し、プロトコルの仕様を少ない労力で表現する方法を提案する。さらに、提案する表現形式で表したプロトコル表現を入力として、自動的にラッパを生成する手法を提案する。

2 既存の研究

ラッパを自動生成する手法については、これまでに様々な研究が行われている。これらの研究を、ラッパを生成する過程に着目して大別すると、ライブラリベースの手法と、プロトコルの仕様から直接合成を行う手法の 2 種類の方向性に分けられる。本論文ではこれ以降、IP 固有のインターフェースプロトコルを単に、固有プロトコルと記すことにする。

ライブラリベースの手法は、変換元プロトコルを内部標準プロトコルに変換してから、変換先プロトコルに変換する手法である。固有プロトコルと内部標準プロトコルの間を変換するラッパの設計記述をライブラリ化しておき、ライブラリの要素を組み合わせることで様々な固有プロトコル間の変換を行うことができるようになる。この手法の利点は、新しい IP インターフェースプロトコルを導入する際に内部標準プロトコルへのラッパを製作すれば、以後、既知のプロトコル同士の変換を設計時間の面で低コストに行うことができるということである。反面、内部標準プロトコルを経由して変換するため、固有プロトコル同士で直接変換する場合に比べて余分な論理回路が生成されるといった短所がある。この手法を運用する際には、内部標準プロトコルへ変換するラッパの生成に対するコストにラッパライブラリの利用頻度を加味したものが、固有プロトコル同士で直接変換する場合のコストと比べて優位であるか考える必要がある。現在、ライブラリベース手法に関する多くの研究では、内部標準プロトコルやラッパの構造を工夫してスループットの向上や、アクセス遅延の短縮などの付加価値を与えることに重点が置かれている。Roman L.Lysecky らによる研究 [12] ではラッパにデータプリフェッチの機構を搭載してアクセス遅延を低減する手法が提案されている。また、Ferid Gharsalli らによる研究 [13] ではメモリと通信するモジュールと、プロセッサと通信するモジュールを組み合わせてラッパを構成することにより、抽象度の高い設計の中でメモリを抽象化して扱えるようにする手法が提案されている。多くの論文では、ライブラリが十分に豊かであることを仮定しており、固有プロトコルの情報をもとにライブラリの要素を作成する方法については詳しく述べられていない。

直接合成を行う手法とは、2 つのプロトコルの仕様記述を与えることによって、それらのプロトコル間を翻訳する論理回路を生成するというものである。Roberto Passerone らによる研究 [14] では、変換したいプロトコルの仕様を有限オートマトンで表現し、ラッパの FSM(Finite State Machine, 有限状態機械) を生成する手法が提案されている。ここで、オートマトンとは、プロトコルの制御フローを状態遷移図で表現したものである。オートマトンの状態遷移は、表現対象のプロトコルの入出力に対応している。Passerone らの提示した手法では、2 つのオートマトンから状態を一つずつ選択して組をつくり、プロトコル間を翻訳する FSM の状態とする。生成される FSM の状態遷移に伴う入出力は、対応する 2 つのオートマトンの状態遷移に伴う入出

力を合わせたものになる。状態の組を選択する際には、オートマトンの初期状態同士の組からスタートして、そこから遷移できる全ての組を深さ優先で探索していく。探索時には、データ依存に違反していないかチェックが行われ、違反状態にしかたどり着けない状態は、探索木のバックトラックによって削除される。ここで、データ依存違反とは、ラッパの送信側インターフェースで出力するデータが、ラッパの受信側インターフェースで受け取られていないことを指す。探索の結果、オートマトンの終状態の組にたどり着くことができれば、初期状態から終状態に至るパスが、この手法の解であり、ラッパの FSM ということになる。さらに、探索中に複数の解が見つかった場合は、終状態に至るまでに経由する状態数が最も少ないものを解として採用する。こうすることで、全ての解の中から最もラッパによる遅延が小さい解を得ることができる。論文中で示されている成果では、簡単なプロトコルを扱う方法しか示されていないが、この方式をベースとして、実用的なプロトコル間でプロトコル変換を行う研究が進められている [15]。また、プロトコルの状態遷移図に探索アルゴリズムを適用してラッパの FSM を生成する手法としては、Passerone により提案された手法の他に、D'Silva らにより提案された手法 [16] などがある。

既存の研究の多くは、ラッパ生成手法に対して変換したいプロトコルの仕様を伝える方法として、FSM またはオートマトンを用いている。プロトコルを正規表現の形で表現する場合もあるが、正規表現とオートマトンは抽象度が等しいので、この 2 者は手法の入力形式として見た場合は同じものと見なすことができる。これに対し、FSM・オートマトン以外の入力形式をもとにラッパを合成する手法として、James Smith らによる研究 [17] がある。Smith らは、提案する手法を POLARIS というツールに実装している。POLARIS は、インターフェース部分の記述を含む Verilog-HDL の設計記述を入力として受け取り、設計記述のインターフェースプロトコルを共通プロトコルに変換するラッパを生成する。しかし、Smith らの手法には、ラッパ生成手法を適用する端子を人手で指定しなければならないことや、データやアドレスなど信号線の意味を反映した処理を行うことができないといった欠点がある。また、生成されたラッパは、入力した設計記述に対してしか適用できないので、ラッパをライブラリに保存して再利用することは難しい。

3 提案手法

3.1 ラッパ生成手法の概要

本研究では、ライブラリベースのラッパ生成手法に基づいて、少ない作業量でラッパを生成する手法を提案する。図 1 に示したのは、提案手法で生成されるラッパのブロック図である。このラッパは、マスタインタフェースが出力した固有プロトコル A の信号を受け取るマスタコントローラ、固有プロトコル B の信号をスレーブインターフェースに出力するスレーブコントローラ、両者を接続するバッファから構成される。以降、マスタコントローラとスレーブコントローラの 2 つを総称してラッパコントローラと呼ぶ。ライブラリベース手法に基づいて考えると、ラッパコントローラは固有プロトコルをバッファの読み書きという内部標準プロトコルに変換するラッパである。

提案するラッパ生成手法では、図 2 の手順に従ってラッパの HDL 記述を作成する。変換したいプロトコルのラッパコントローラが既にライブラリに登録されている場合は、必要なマスタコントローラとスレーブコントローラをライブラリから取得する。変換したいプロトコルがライブラリに登録されていない場合は、3.3 節で提案する表現方法を用いてプロトコルの仕様書からプロトコルの仕様記述を作成し、プロトコルの仕様記述とラッパコントローラのテンプレートからラッパコントローラの HDL 記述を生成する。プロトコルの仕様記述を作成する際には、仕様書に記載されたタイミングチャートを主な情報源としてプロトコル仕様記述を作

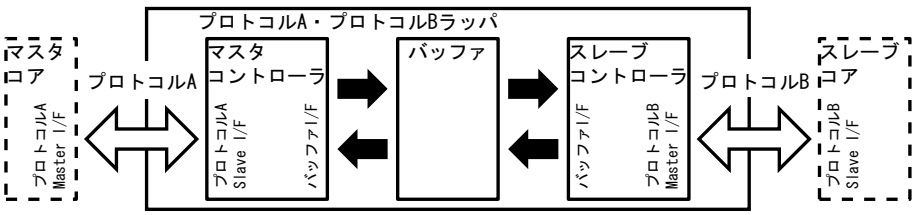

図 1　提案するラッパのブロック図

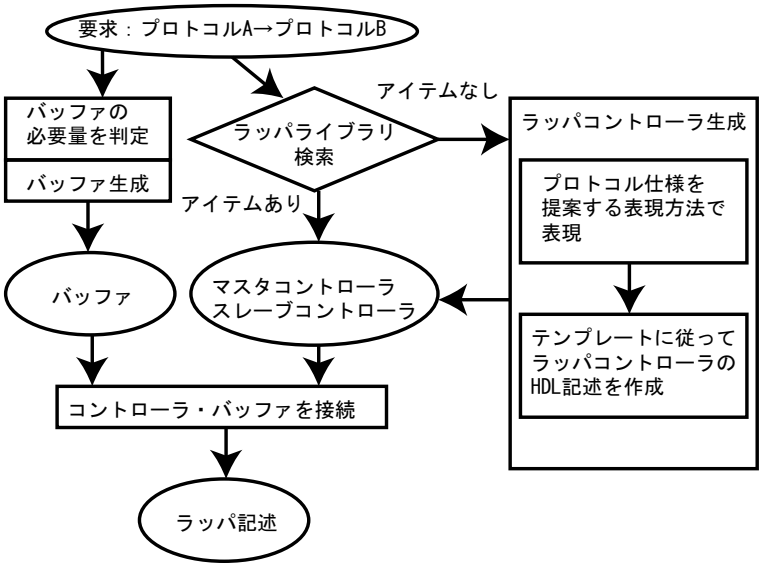

図 2　ラッパの生成手順

成する。最後にマスタコントローラとスレーブコントローラをバッファに接続することでラッパは完成する。バッファの容量の選択については、変換するプロトコルの性質と、ラッパを装着する IP の性質を考えて、適切な容量のものを選択する。

提案するラッパ生成手法では、ラッパライブラリを利用することでラッパコントローラの新規設計の必要回数を減らすことができる。さらに、ラッパコントローラの新規設計を行う際には、提案するプロトコル表現方法を用いてプロトコル仕様記述をタイミングチャートから簡単に作成できる。このことにより、入力に FSM やそれと等価な表現を用いる手法と比べて設計者の作業量が少なくなり、ラッパコントローラ生成手法の自動化とあわせることでラッパコントローラの新規設計をより簡単に行うことができる。すなわち、ラッパコントローラの再利用と新規設計の簡単化により、従来手法よりもラッパ生成作業における設計者の作業量を削減することができる。

3.2　対象とする IP インターフェースプロトコル

本研究で提案するプロトコル表現方法およびラッパ生成手法の対象とする IP インターフェースプロトコルは、以下のような条件を満たすものと仮定する。

- **クロック、リセットに関する仮定**

  対象のプロトコルは、クロックに同期してデータをやりとりし、クロックに同期したリセット信号を

持つものとする。さらに実装を簡単にするため、ラッパの両端のインターフェースは同じクロックに同期して動いているものとする。

- **転送形態に関する仮定**

  提案する手法では、ラッパ生成の対象とするプロトコルを、メモリマップタイプのパラレル通信プロトコルと仮定する。シリアル通信のようにクロックごとにデータの断片が渡されるプロトコルは対象としない。また、送受信するデータそのものを伝える信号線と、データの送受信の手順を制御する信号線が明確に分かれていると仮定する。さらに、データリード・データライトのリクエストを発行するマスタインターフェースと、リクエストにレスポンスを返すスレーブインターフェースが明確に区別できるとする。

  また、実装を簡単にするために、取り扱うプロトコルの信号線は全て単方向通信であるとして、データバスは全て同じバス幅をもつことにする。双方向通信線は、スリーステートバッファを通して入力と出力の単方向通信線に分けて扱うことによって対応可能である。また、バス幅に関する仮定は、ビット幅の異なるデータを取り扱うルール（複数回に分けて転送、複数回分のデータを一括して転送など）を定めることによって取り払うことが可能である。

- **待機機構に関する仮定**

  対象とするプロトコルは、ビジー状態が解除されるまで、あるいはリクエストに対してアクノリッジが返るまでマスタが待機するしくみを持っていると仮定する。リクエストを発行してからレスポンスが返るまでのサイクル数が固定のプロトコルでは、ラッパの挿入によるサイクル遅延の増加によって、レスポンスが間に合わなくなる可能性や、ビジー状態を無視して発行されたリクエストが消失する可能性があるためである。

- **その他の仮定**

  リクエスト、レスポンスの動作と異なるタイミングで伝達される信号、例えば割り込み要求のような信号は存在しないものとする。

- **ブロッキングプロトコル・パイプラインプロトコル**

  本研究では、ブロッキングプロトコル・パイプラインプロトコルの両方を取り扱う。ここで、ブロッキングプロトコルとは、リクエストに対するレスポンスが返るまで次のリクエストを発行できないプロトコルのことである。これに対し、パイプラインプロトコルとは、リクエストの処理とレスポンスの処理が独立しており、レスポンスが返る前に次のリクエストを先行して発行することができるプロトコルのことである。ただし、パイプラインプロトコルの場合は、レスポンスはリクエストが発行された順番に返ってくるものと仮定する。

3.3 プロトコルの表現方法

この節では、固有プロトコルの表現方法の提案をおこなう。我々は、今回プロトコルの表現方法を提案するにあたって、オンチップ通信プロトコルの仕様を集めて分析を行った。また、提案するプロトコル表現方法で対象の固有プロトコルが表現できない場合でも、提案するラッパコントローラ生成手法を適用せずにラッパコントローラを人手で記述してライブラリに取り込むことで、他のプロトコルとの変換を行うことができる。

提案するプロトコルの表現方法は、IP インターフェースの端子の構成、データ線とデータ属性の関連付け、コントロールフロー・データタイミングの表現、信号極性・エンコーディングの表現の 4 セクションから構成される。以降、第 3.3.1 節から第 3.3.4 節でそれぞれのセクションについて説明し、第 3.3.5 節でプロトコル仕

様書からプロトコル仕様記述を作成する際の手順について説明する。

3.3.1 端子の構成

ここでは、固有プロトコルで用いられる信号線の名前とビット幅、信号線のドライバ、コントロール線とデータ線の区別を表現する。まず、信号線の名前のリストを作成して、IP の端子の中から、プロトコル変換を適用する端子を特定する。その後、信号線のドライバが、マスタインターフェースか、スレーブインターフェースか、外部のユニットかを特定する。これらの情報は、仕様書の中で入出力端子の一覧という形で提供されているはずである。さらに、クロックの信号線と、同期リセットの信号線を確定し、残りの信号線がデータ線とコントロール線のどちらにあたるかを分類する。データ線は送受信されるデータそのものを伝える信号線であり、コントロール線はデータの送受信の手順を制御する信号線である。現在のところ、両者の区別は人手で判断する必要がある。

3.3.2 データ線とデータ属性の関連付け

データ線のカテゴリに分類された信号線に対しては、「信号線 ADDR はアドレスを伝達」のように、データ線とデータ属性を関連付けする必要がある。代表的なデータ属性としては、アドレス、書きこみデータ、読み出しデータなどが挙げられる。この関連付けは、変換先プロトコルのデータ線と対応をとる際に用いられる。

3.3.3 コントロールフロー・データタイミングの表現

この節では、コントロール線同士のタイミング関係とコントロール線とデータ線の間のタイミング関係を表現する方法について説明する。提案するプロトコル表現法では、コントロール信号のエンコーディングとコントロール信号のタイミングを分離して扱う。以降で説明するタイミング表現方式では、信号は有効値または無効値という抽象的な値をとると仮定しており、実際の信号値がどのような値をとるかについては、信号のエンコーディング表現で扱う。

本論文では、何の転送も行っていない待機状態のことを IDLE 状態と呼び、IDLE 状態から動作を開始し、再び IDLE 状態へ戻るまでの一連の動作のことをシーケンスと呼ぶことにする。例えば、データを 1 ワード書き込む際の一連の信号遷移などがシーケンスにあたる。多くの場合、一つのシーケンスは仕様書のタイミングチャート一枚に相当する。すなわち、プロトコルは、単一ないし複数のシーケンスから構成される。

提案するタイミング表現では、プロトコルの 1 シーケンスを、以降で説明する 4 種類のステートメントを用いて表現する。また、記述するプロトコルがパイプラインプロトコルの場合は、リクエストに関する処理とレスポンスに関する処理に分けてシーケンスを記述する。ステートメントの定義では 2 種類の引数型を使用する。まず、引数型 signal はタイミング制約の対象とする信号線を指定する型である。そして、引数型 time はクロックサイクルの単位で表した時間である。

1 Handshake(signal SignalStart, signal SignalEnd)

Handshake() 文は 2 つの信号線を引数にとって、シーケンス上で 2 時点によって区切られる期間を表現する。以後の説明では、この期間のことをハンドシェーク期間と呼ぶ。図 3 にタイミングチャートの形で、引数とした信号とハンドシェーク期間の関係を示す。なお、タイミングチャート中では、下線は信号が無効値であること、上線は信号が有効値であること、網掛けはその時点で信号の値を参照しないことを表す。Handshake() 文では、SignalStart が有効値をとることで、ハンドシェーク期間に入り、SignalEnd が有効値をとることで、ハンドシェーク期間が終了すると定義する。また、ハンドシェーク期間中では SignalStart の値は無視し、ハンドシェーク期間外では SignalEnd の値は無視する。さらに、SignalStart が読み込まれ

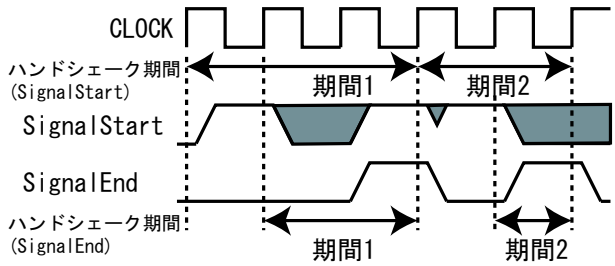

図 3　タイミング制約表現 Handshake()

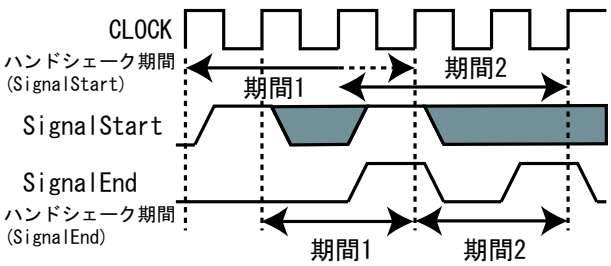

図 4　タイミング制約表現 OverlapHandshake()

る時刻を $\tau_{SignalStart}$、SignalEnd が読み込まれる時刻を $\tau_{SignalEnd}$ とすると、$\tau_{SignalStart} \leq \tau_{SignalEnd}$ が満たされるとする。SignalStart の発行者と SignalEnd の発行者が異なるならば、Handshake() 文を用いてリクエスト・アクノリッジのような待機機構を表現することができる。また、多くのプロトコルにおいて、Handshake() 文で区切られる期間は、そのまま 1 つのシーケンスの期間に相当する。

なお、一つのシーケンスに複数の Handshake() 文、次に述べる OverlapHandshake() 文が宣言された場合、全体がどういう意味を持つかは未定義である。これは、シーケンス中に複数の待機機構が存在する場合が実用性から考えると想定できないためである。

2 OverlapHandshake(signal SignalStart, signal SignalEnd)

Handshake() 文では、ハンドシェーク期間中は SignalStart の読み込みは行わないと定義したので、SignalEnd が有効値をとった次のクロックで SignalStart の読み込みを行う。これに対し、OverlapHandshake() 文では図 4 に示すように、SignalEnd が有効値をとったクロックで SignalStart の読み込みを行う。したがって、SignalStart の発行者から見たハンドシェーク期間は重複している。そのため、SignalStart が読み込まれる時刻を $\tau_{SignalStart}$、SignalEnd が読み込まれる時刻を $\tau_{SignalEnd}$ とすると、$\tau_{SignalStart} < \tau_{SignalEnd}$ である。Handshake() 文と異なり、等号は含まない。

SignalStart と SignalEnd の発行者が異なるならば、SignalStart の発行者は SignalEnd がいつ発行するか知りえないので、SignalStart のサンプリングは SignalEnd の発行後か SignalEnd と同時のどちらかでしか起こりえない。したがって、2 時点で区切られる期間は Handshake() 文か OverlapHandshake() 文のどちらかで表現できる。

3 Data_Hold(signal SignalTarget, time latency)

Data_Hold() 文は、信号 SignalTarget が、ハンドシェーク期間中有効値をとることを表現する。ただし、OverlapHandshake() 文の宣言時に SignalStart の発行者は、Data_Hold() 文を使用できない。これは、ハンドシェーク期間が重複しているためである。latency は、SignalStart が有効値をとってから SignalTarget の値が有効値をとるまでの遅れ時間を表す。例えば、latency=0 では SignalStart が出力されると同時に SignalTarget が有効値になる。図 5 に latency=0 の場合と latency=1 の場合を示す。

4 Data_Oneshot(signal SignalTarget, signal SignalTrigger, time latency)

Data_Oneshot() 文は、信号 SignalTrigger が有効値をとった時点から一定の時間差をとって、信号 SignalTarget が有効値をとることを表現する。この時間差は、引数 latency で与えられる。latency は通常非負の値であり、SignalTrigger が有効値をとってから SignalTarget が有効値をとるまでの遅れ時間を表す。latency が負の値をとることが許されるかについては、タイミング表現を入力とする手法が負の値を許すかによって決まる。提案するラッパ合成手法でどのような場合に負の値が許されるかについては、第 3.5.5 節で述

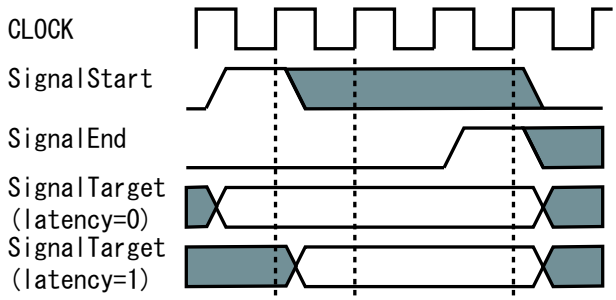

図 5　タイミング制約表現 Data_Hold()

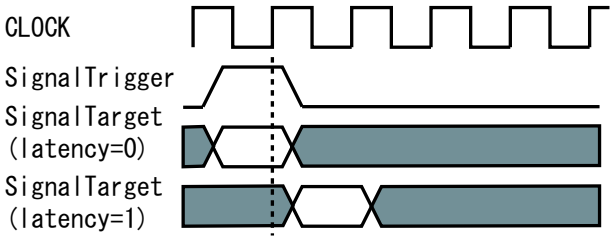

図 6　タイミング制約表現 Data_Oneshot()

べる。Data_Oneshot() 文では、Handshake() 文または OverlapHandshake() 文が宣言されている場合は、SignalStart か SignalEnd を SignalTrigger として指定する。また、Handshake() 文が宣言されていない場合は、シーケンス開始時刻を規定する信号を、Data_Oneshot(signal1, signal1, 0) のように自身を参照することで表現する。他の信号はこの基準信号を SignalTrigger として、Data_Oneshot() 文を宣言する。図 6 に latency=0 の場合と latency=1 の場合を示す。

最後に、タイミング表現を作成する手順について、例題を用いて補足説明する。図 7 に例題とするシーケンスを示す。信号 STROBE・ACK はコントロール線、信号 ADDR・DATA はデータ線であるとする。設計者が最初にするべきことは、シーケンスの開始を定義する信号とシーケンスの終了を定義する信号を見つけ出すことである。この 2 つの信号は通常、コントロール線の中から選択する。開始を定義する信号と終了を定義する信号の両方が見つかった場合は、その 2 つの信号を Handshake() 文あるいは OverlapHandshake() 文で表現する。開始を定義する信号のみ見つかった場合は、その信号を Data_Oneshot() 文で表現する。例題では、信号 STROBE がシーケンスの開始を、信号 ACK がシーケンスの終了を定義する信号である。さらに、ACK が有効値になったサイクルでは STROBE の値は参照されていない。したがって、両者の関係は HandShake(STROBE, ACK) と表現することができる。シーケンスの開始・終了を定義する信号が見つかった後は、他の信号のタイミングを Data_Hold() 文と Data_Oneshot() 文で記述する。例題では、STROBE, ACK を基準として他の信号のタイミングを記述する。信号 ADDR は、STROBE と同時に有効値になり、1 クロック期間だけ有効値が維持されるので、Data_Oneshot(ADDR, STROBE, 0) と表現される。なお、2 クロックの期間にわたり有効値が維持される信号は、同じ信号に対して Data_Oneshot() 文を複数用いることで表現する。信号 DATA は、STROBE から 1 クロック遅れて有効値となり、ハンドシェーク期間中は有効値が維持されるので、Data_Hold(DATA, 1) と表現される。

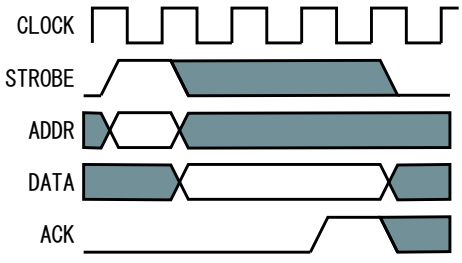

図 7　例題シーケンス

3.3.4 信号極性・エンコーディングの表現

タイミング表現では、信号は有効値・無効値の 2 値をとると仮定していたが、エンコーディングの表現では、有効値・無効値が実際のコントロール線のどのような値に対応するかを規定する。例えば、同期リセットを指示する 1 ビットの信号線に着目して考えると、リセットが有効になるのは、信号値が論理 0 の場合と論理 1 の場合が考えられる。このような対応関係に対して、「タイミング表現での Reset の有効値は、固有プロトコルでの論理 0 にあたる」のような、変換規則を定義することになる。

さらに一般的にするために、ビット幅 n のコントロール信号線を扱う方法を考える。この場合は、コントロール信号線の意味を考えて複数本の 1 ビット信号線にデコードし、デコード後の信号線にタイミング表現を適用する。逆に、複数本のタイミング表現の信号から多ビットのコントロール信号線を得るときは、プライオリティエンコーダを用いて変換する。多くの仕様書では、信号線ごとに値と意味の対応表が掲載されているので、その部分を抜粋して作業に必要な情報を得る。なお、データ線の値は、プロトコルの制御フローには影響しないので、エンコーディングを考える必要はない。

また、エンコーディング表現では、信号線間の変換規則に加えて、「固有プロトコルの Protocol_Command が 010 の時、シーケンス 1 を実行」のような、複数のシーケンスを選択する規則も取り扱う。

3.3.5 プロトコル表現作成の手順

これまでに説明したプロトコル表現方法を用いてプロトコルの仕様書からプロトコル仕様記述を作成する際の手順について説明する。プロトコル仕様記述を作成する際には、最初に端子の構成の仕様記述を作成し、その後にデータ線属性・エンコーディング表現・タイミング表現の仕様記述を作成する。この順序でプロトコル仕様記述を作成する理由は、プロトコル変換の対象とする信号線を特定しなければ他の作業ができないためである。データ線属性の仕様記述作成作業は、エンコーディング表現・タイミング表現の仕様記述作成作業と独立しており、いつ実行してもよい。

エンコーディング表現とタイミング表現の仕様記述作成作業は、多ビット幅のコントロール線の扱いを決めることから始める。全てのコントロール線が 1 ビット幅の場合は、タイミング表現で扱う信号と固有プロトコルで扱う信号は 1 対 1 で対応しているため、エンコーディング表現の仕様記述作成作業とタイミング表現の仕様記述作成作業は並行して行うことができる。多ビット幅のコントロール線がある場合は、それをどのように 1 ビット幅の信号線の集合に変換するかを決定し、エンコーディング表現とタイミング表現の仕様記述作成作業に移る。タイミング表現の仕様記述作成作業では、1 ビット幅に変換された信号に対して 4 種のステートメントを適用する。また、エンコーディング表現の仕様記述作成作業では、多ビット幅のコントロール線を変換するルールを記述する。

3.4 バッファ

この節では、図 1 におけるバッファユニットについて説明する。第 2 章で述べた通り、ライブラリベースのラッパ生成手法では、中間プロトコルの仕様がラッパの性能やラッパ生成作業の難易度に大きく関わることになる。本研究においては、簡単な手順でラッパコントローラが生成できることを目標にバッファの仕様を定めることにする。図 1 で示したように、マスタコントローラとスレーブコントローラはバッファを介して接続されている。このため、それぞれのコントローラは、バッファに蓄えられた要素に従って動作を行うことになり、相手の内部状態を気にする必要はない。これはすなわち、相手側のプロトコルの制御フローの影響を直接

表 1　本研究でバッファに実装するチャネルの一覧

チャネル名	内容	意味	書き込み端子	読み出し端子	満杯判定	ゼロ判定
Reqbuff						
Addr	データ	アドレス	Addr_wr	Reqbuff_rd	Reqbuff_full	Reqbuff_zero
Data	データ	書き込みデータ	Data_wr			
Command	列挙子	コマンド	Command_wr			
Resbuff						
Data	データ	読み出しデータ	Data_wr	Resbuff_rd	Resbuff_full	Resbuff_zero
Status	列挙子	リクエスト実行結果	Status_wr			
Resexist	列挙子	レスポンス有無	Resexist_wr	Resexist_rd	Resexist_full	Resexist_zero

的には受けないということである。

バッファユニットは、マスタコントローラからスレーブコントローラへ情報を伝達するリクエストバッファとスレーブコントローラからマスタコントローラへ情報を伝達するレスポンスバッファの 2 つから構成される。以降の説明では、それぞれを Reqbuff、Resbuff と記す。さらに、それぞれのバッファは、データ線のデータ属性に対応する FIFO レジスタと、ラッパコントローラ間で互いの状態を知らせるための FIFO レジスタから構成される。以降の説明では、この FIFO レジスタをチャネルと呼ぶ。本研究で使用する FIFO レジスタは、ごく一般的なものであり、以下の端子が存在する。

- **クロック**
  FIFO レジスタの全ての処理は、クロックに同期して行われる。本研究では、ラッパは単一のクロックで動作すると仮定している。クロックを読み出し側と書き込み側で別にすることでマスタコアとスレーブコアのクロックが異なる場合に対応することができる。
- **リセット**
  クロックに同期したリセット入力である。
- **データ入出力**
  データ線と接続し、データの入出力を行う端子である。
- **書き込み・読み出し制御**
  データ書き込み、データ読み出しの入力端子である。読み出しデータは常に出力端子から参照できるので、読み出し端子は FIFO レジスタの先頭要素を破棄するために用いる。
- **満杯判定・ゼロ判定**
  FIFO レジスタ内の要素数を教える端子である。本研究では、FIFO レジスタが空であるかの判定を行う端子と、満杯であるかの判定を行う端子が存在すると仮定している。

本研究では、メモリマップ型のプロトコルを扱う上で必要最小限の種類のチャネルを実装することにした。固有プロトコルの読み書き動作を扱うためには、最低でも、アドレスと書き込みデータを伝えるチャネルがリクエスト側に、読み出しデータを伝えるチャネルがレスポンス側に必要である。さらに、スレーブコントローラが行うべき動作を指示するチャネルと、動作の結果を返すチャネルも必要である。以上のような理由により、本研究では、表 1 に示すチャネルをバッファに実装した。さらに、バッファ全体の制御端子の一覧も表 1 にあわせて示す。提案するバッファの構成では、書き込みはチャネルごとに個別に制御して、読み出しは全

表 2　本研究でチャネルを通して授受される列挙子の一覧

チャネル名	列挙子　括弧内は意味
Reqbuff_Command	RD(シングルリード動作), WR(シングルライト動作)
Resbuff_Status	OK(シーケンス実行成功), ERR(シーケンス実行失敗)
Resexist	YES(有効レスポンス有り), NO(有効レスポンスなし)

チャネルで一括して制御する。これは、データ書き込みはデータ到着に合わせて順次行うべきであり、先頭データの破棄は、シーケンスの終了時に一括して行えばよいと考えられるためである。また、バッファの満杯判定・ゼロ判定の端子は、全てのチャネルの満杯判定・ゼロ判定の OR 値を出力する。これは、ラッパコントローラが動作の開始・待機を判断する際には、バッファ全体の満杯判定・ゼロ判定が分かれば十分だからである。しかし、バッファが読み出し制御と要素数の判定を全チャネル一括で行う構造をとるため、各チャネル内の要素数は互いに等しくなければならない。このため、ラッパコントローラは、チャネルに書き込むデータが存在しない場合は仮のデータを書き込まなくてはならない。

リクエストバッファのアドレス・書き込みデータのチャネルと、レスポンスバッファの読み出しデータのチャネルは、データ線に直接接続される。この他のチャネルは、ラッパコントローラ間で互いの状態を知らせるための信号をやり取りする。ラッパコントローラ間でやり取りされる信号では、00 なら WRITE、01 なら READ というように、数値と 1 対 1 で意味が割り当てられているとする。本論文ではこのような信号において、数値と 1 対 1 で結びついて数値の意味を表現する識別子を列挙子と呼ぶ。以降の説明では、信号線の実際の値ではなく列挙子を用いてラッパ生成手法を説明する。

表 2 に、今回行った実験で用いた列挙子の一覧を示す。Reqbuff_Command は、コマンドを伝達するチャネルである。このチャネルではスレーブコントローラがとるべき動作を指定する列挙子を伝達する。本研究では、1 ワードの読み出し（シングルリード）と 1 ワードの書き込み（シングルライト）の 2 種類の列挙子を想定している。全てのシーケンスはマスタコントローラでシングルリードとシングルライトのコマンドに分解され、バッファに格納される。スレーブコントローラは、シングルライト・シングルリードの 2 種類の動作をサポートすることが求められる。

Resbuff_Status はスレーブコントローラでシーケンスを実行した結果を伝達するチャネルである。本研究では、ResBuff_Status で成功と失敗の 2 種類の列挙子を扱うことを想定している。

Resexist は、スレーブコントローラで実行したシーケンスのレスポンス有無を伝達するチャネルである。本研究では、レスポンス利用可、レスポンス利用不可の 2 種類の列挙子を扱うことを想定している。このチャネルは、読み出しデータやステータスとは独立して読み書きされる。

バッファの容量は、自由に変更可能である。しかし、どちらかのプロトコルがブロッキングプロトコルの場合、同時に 1 つしかリクエストを発行・受理できないので、バッファには 1 つしかデータが書き込まれない。したがって、バッファの容量はパイプラインプロトコル同士を変換する時のみ意味をもつ。なお、本研究では、バッファの容量を決定する方法については扱わない。

3.5 ラッパコントローラの生成手法

3.5.1 ラッパコントローラの構成

この節では、第 3.3 節で提案したプロトコル表現からマスタコントローラ・スレーブコントローラを生成する方法について述べる。

図 8 にラッパコントローラのブロック図を示す。提案するラッパコントローラは、コンバーターと FSM から構成されている。エンコーディングは全てコンバーターの組み合わせ回路で処理され、信号のタイミングの制御やバッファの読み書き制御は FSM が扱う。これは、プロトコル表現が、コントロール線のエンコーディングと信号のタイミングを別々に扱っていることに対応している。また、ラッパコントローラの FSM は、ブロッキングプロトコルを扱う場合は 1 つのブロッキング処理 FSM から、パイプラインプロトコルを扱う場合はパイプラインリクエスト処理 FSM とパイプラインレスポンス処理 FSM の 2 つから構成される。これは、パイプラインプロトコルのタイミング表現を記す際にリクエストとレスポンスの処理を分けて記述していることに対応している。データ線は、バッファの対応するチャネルのデータ入出力端子に接続される。

図 9 にラッパコントローラを生成する作業の手順を示す。まず、プロトコル表現方法に従い、プロトコルの仕様書から端子のリスト、データ線のデータ属性関連付け、コントロール線のエンコーディング、タイミング表現を作成する。作成したプロトコル表現からラッパを生成するためには、プロトコル表現にラッパの構成に即した情報を追加する必要がある。データ属性関連付けの作業では、データ線とバッファのチャネルの対応関係を記述する。エンコーディング表現の作成作業では、第 3.3.4 節で示した規則の他に、固有プロトコルの信号と ReqBuff_Command・ResBuff_Status で用いる列挙子の間の変換規則を決定して、エンコーディングテーブルに記録する。

こうして作成されたプロトコル表現を用いて、ラッパの HDL 記述の生成を行う。端子リストからは、ラッパコントローラの入出力端子を生成し、エンコーディングテーブルからは、コンバーターの記述を生成する。そして、タイミング表現から FSM の記述を生成する。バッファ結線情報は、コンバーターと FSM を生成する際に参照される。最後に、生成した記述を結合してラッパコントローラを生成する。

以降、プロトコル表現から各部の設計記述を生成する方法を説明する。第 3.5.2 節では、コンバーターの生成手法について、第 3.5.3 節以降では、ラッパコントローラ FSM の生成手法について説明する。設計を記述する言語については特に制限はないが、具体的な説明が必要な箇所については Verilog を利用して説明する。

3.5.2 コンバーターの生成手法

コンバーターの生成作業では、エンコーディングテーブルとバッファ結線情報を入力情報とする。エンコーディングテーブルには、固有プロトコルのコントロール線と FSM の入出力信号の変換規則、シーケンス選択信号の生成規則、固有プロトコルの信号とバッファユニットの列挙子の変換規則の 3 つが記載されている。エンコーディングテーブルは、固有プロトコルの信号と FSM・バッファの信号の間の真理値表と考えられるので、エンコーディングテーブルを真理値表とする組み合わせ回路を記述すれば、コンバーターを生成することができる。また、バッファ結線情報には、固有プロトコルのデータ線とバッファの入出力端子の接続規則が記載されており、コンバータ生成作業ではこの情報をもとにデータ線とバッファの入出力端子の結線を行う。なお、本論文でおこなった実験では、FSM とコンバータを生成する際には、タイミング表現の有効値は 1、無効値は 0 とした。

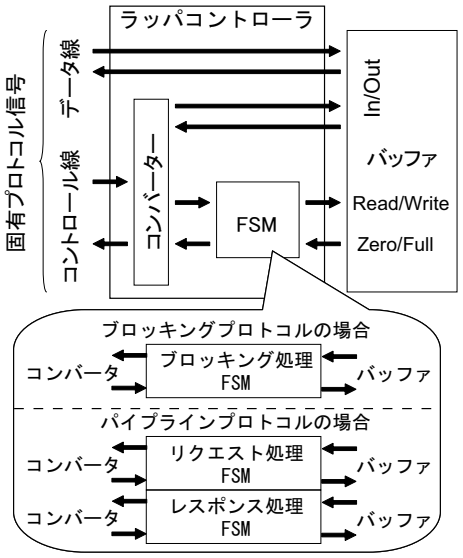

図 8　ラッパコントローラブロック図

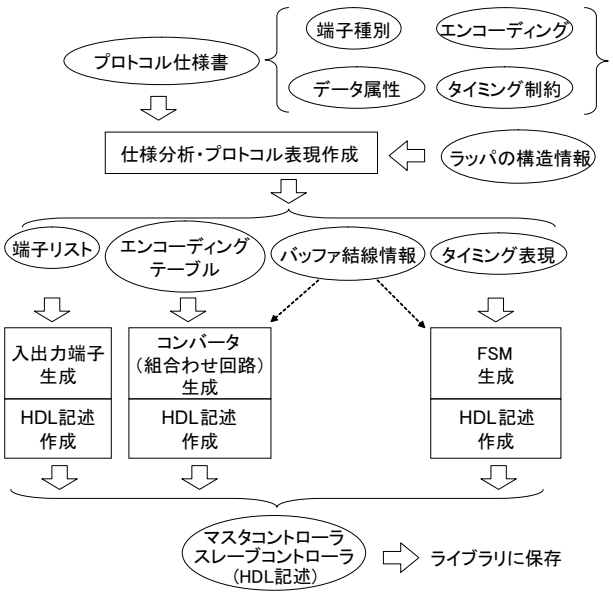

図 9　ラッパコントローラ生成手順

3.5.3　ラッパコントローラ FSM の生成手法の概要

提案する手法では、ラッパコントローラ FSM を生成する際に FSM のテンプレートを利用する。まず、ラッパコントローラ FSM の生成作業を説明する前に、各ラッパコントローラ・各プロトコルに対応した FSM を分類する。図 10 に、FSM とバッファの関係をリクエスト・レスポンスの伝達に着目して簡略化して示す。この図を参照すると、

- 固有プロトコル側からの信号を受けて動作を開始し、バッファに書き込みを行う FSM
- バッファ内に有効なデータがある時に動作を開始し、固有プロトコルに信号を送信する FSM

の 2 種類にラッパコントローラ FSM が分類できることが分かる。前者には、マスタコントローラのブロッキング処理 FSM・パイプラインリクエスト処理 FSM とスレーブコントローラのパイプラインレスポンス処理 FSM が該当する。以降、この種類の FSM を受信側 FSM と呼ぶ。また、後者にはマスタコントローラのパイプラインレスポンス処理 FSM とスレーブコントローラのブロッキング処理 FSM・パイプラインリクエスト処理 FSM が該当する。以降、この種類の FSM を送信側 FSM と呼ぶ。提案する手法ではこの分類に基づき、受信側 FSM と送信側 FSM の FSM テンプレートを用いてラッパコントローラ FSM を生成する。

図 11 にラッパコントローラ FSM の生成作業の手順を示す。まず、扱うプロトコルの種別と生成するラッパコントローラの種別から使用する FSM テンプレートを決定する。次に、各シーケンスのタイミング表現を前処理する。前処理の内容は第 3.5.4 節で説明する。タイミング表現の前処理が終わったら、処理後のタイミング表現をもとに FSM テンプレートを加工する作業を行う。第 3.5.5 節ではタイミング表現のステートメントに基づいて FSM テンプレートを加工する方法を示し、第 3.5.6 節では本研究で提案するラッパの構造に特に大きく依存している作業について説明する。タイミング表現の前処理とテンプレートの加工の 2 つの作業により、1 シーケンスに対応する FSM が生成される。以上の作業を全てのシーケンスに対して行った後にシーケンスごとの FSM を結合してラッパコントローラ FSM が完成する。FSM 結合作業の詳細は第 3.5.7 節で述

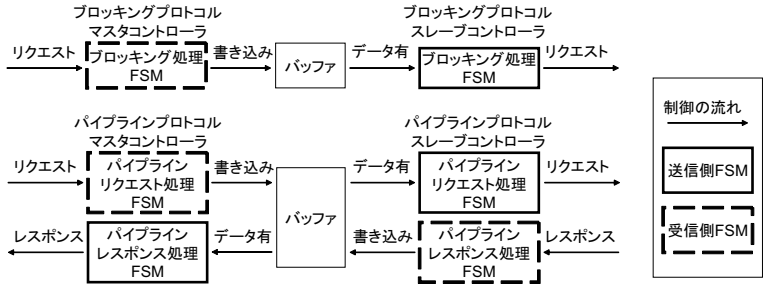

図 10　ラッパコントローラ FSM の分類

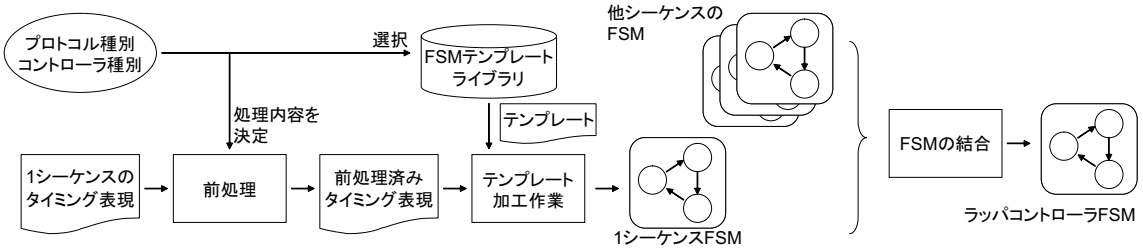

図 11　ラッパコントローラ FSM の生成手順

べる。

3.5.4　タイミング表現の前処理方法

Data_Hold() 文と Data_Oneshot() 文は、引数 SignalTarget に、コントロール線とデータ線の両方をとることができる。しかし、ラッパコントローラ FSM が出力として扱うのはコントロール線であり、データ線に対するタイミング表現があった場合はバッファの対応するチャネルの書き込み端子に出力を出す必要がある。前処理工程では、入力されたタイミング表現のうち、データ線のタイミングに関係している箇所だけを修正する。まず、HandShake() 文および OverlapHandshake() 文に対しては、引数にデータ線を含まないので、第 3.5.5 節の処理をそのまま適用する。次に、Data_Hold() 文と Data_Oneshot() 文の処理手順を図 12 に示す。SignalTarget がコントロール線の場合は、第 3.5.5 節の処理をそのまま適用する。SignalTarget がラッパへ入力するデータ線の場合は、データ線と接続されたチャネルの書き込み制御端子に出力を行う。データ線が有効値である間に 1 クロック期間だけ、書き込み制御端子に有効値を出力すればよいので、Handshake(SignalStart, SignalEnd) かつ Data_Hold (”データ線名”, t) が宣言された場合は、Data_Oneshot (”書き込み端子名”, SignalStart, t) に、Data_Oneshot (”データ線名”, SignalTrigger, t) が宣言された場合は、Data_Oneshot (”書き込み端子名”, SignalTrigger, t) に変換して、第 3.5.5 節の処理を適用する。SignalTarget がラッパから出力されるデータ線の場合は、状態数の増減には関与するがバッファの操作は必要ないので、仮の入力信号に対する Data_Oneshot() 文を作成し、第 3.5.5 節の処理を適用する。

3.5.5　FSM テンプレートの加工方法

前処理が終わったタイミング表現は、コンバータで変換された固有プロトコルのコントロール信号線とバッファの読み書き端子のタイミング関係を表現している。FSM テンプレートを前処理後のタイミング表現を用いて加工することで、1 シーケンスの処理に対応した FSM を得ることができる。

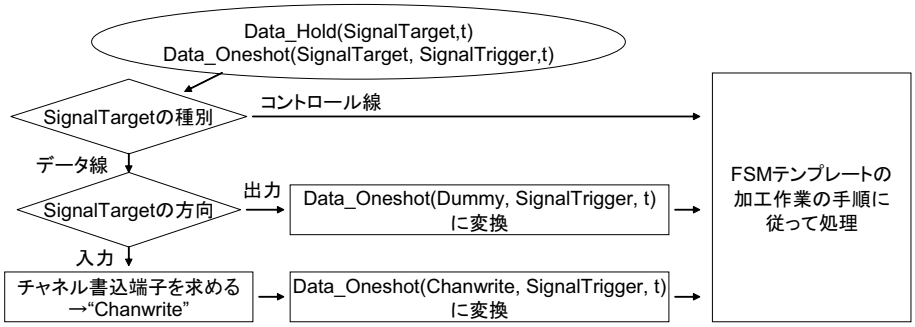

図 12　前処理の作業における Data_Oneshot() 文, Data_Hold() 文の処理

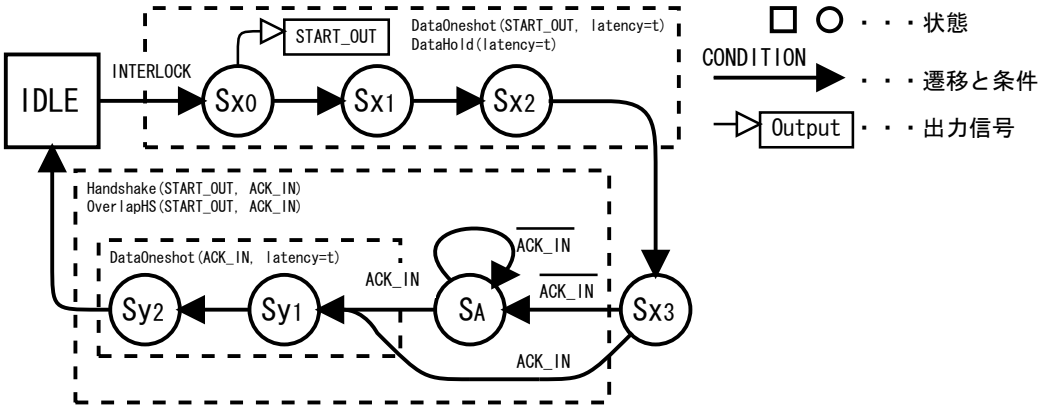

図 13　送信側 FSM テンプレート

まず、送信側 FSM テンプレートを図 13 に示す。このテンプレートは、マスタコントローラのパイプラインレスポンス処理 FSM とスレーブコントローラのブロッキング処理 FSM・パイプラインリクエスト処理 FSM に適用する。以降の説明では、START_OUT を出力信号、ACK_IN を入力信号として、Handshake(START_OUT, ACK_IN) または OverlapHandshake(START_OUT, ACK_IN) または Data_Oneshot(START_OUT, START_OUT, 0) が宣言されているとして説明を行う。送信側 FSM は、バッファ内に有効なデータがある場合に IDLE 状態から動作を開始する。テンプレートでは、この遷移条件を論理式 INTERLOCK で表現する。INTERLOCK の各 FSM での詳細は、第 3.5.6 節で説明する。また FSM テンプレートでは、FSM の出力は FSM の状態と入力で決定され、出力レジスタは生成されない。

図 14 に、送信側 FSM テンプレートに基づいて FSM を生成する手順を擬似コードで示す。擬似コード中では、順序つきリスト STATELIST で FSM の状態と遷移を表現する。STATELIST_PUSH_BACK は STATELIST の末尾に要素を追加する処理であり、STATELIST_POP_LAST_ELEMENT は、STATELIST の末尾の要素を削除する処理である。また、STATELIST_ELEMENT(j) は、先頭から j 番目に格納されている要素を表す。STATELIST に格納される要素の一つ一つは、FSM の状態を表し、SNAME<COND1:DEST1, COND2:DEST2, . . . > の形で表現される。以下に、各識別子の意味を説明する。まず、SNAME は、状態に対する一意な名前である。ここでは、状態名として、IDLE、S_x、S_A、S_y を使用する。次に、パラメータ CONDITION は状態遷移の条件を表す論理式である。無条件遷移は COND=1 で表される。最後に、パラ

```
[FSM 生成ルーチン]
    CALL 状態数変更ルーチン
    SETOUTPUTCONDITION(ACK_OUT, STATE=Sy0)
    FOR-EACH{ Data_Hold(), Data_Oneshot} begin
        CALL 信号出力ルーチン
    end

[状態数変更ルーチン]
    STATELIST = IDLE<INTERLOCK:next, !INTERLOCK:this>

    FOR-ALL{ Data_Hold(latency=i) or Data_Oneshot(SignalTrigger=START_OUT, latency=i) }begin
        m = get_maximum(i)
    end
    FOR-ALL{ Data_Oneshot(SignalTrigger=ACK_IN, latency=i) }begin
        p = get_maximum(i)
    end

    STATELIST_PUSH_BACK  Sx0<1:next>, Sx1<1:next> ... Sxm<1:next>

    IF-DECLARED{ Handshake(START_OUT, ACK_IN) or OverlapHandshake(START_OUT, ACK_IN) }begin
        STATELIST_POP_LAST_ELEMENT to Stemp
        STATELIST_PUSH_BACK  Stemp<ACK_IN:next2, !ACK_IN:next>
        STATELIST_PUSH_BACK  SA<ACK_IN:next, !ACK_IN:this>
    end

    STATELIST_PUSH_BACK  Sy1<1:next>, Sy2<1:next>, ... Syp<1:next>
    STATELIST_PUSH_BACK  IDLE<INTERLOCK:next, !INTERLOCK:this>

[信号出力ルーチン]
    CASE{ SignalTarget's Direction is input }begin
        IGNORE_STATEMENT
    end
    CASE{ Data_Oneshot(SignalTrigger=SignalStart, latency=i) }begin
        SETOUTPUTCONDITION(SignalTarget, STATE=Sxi)
    end
    CASE{ Data_Oneshot(SignalTrigger=SignalEnd, latency=i) }begin
        IF{ i=0 }begin
            SETOUTPUTCONDITION(SignalTarget, (STATE=SA and ACK_IN=1) or (STATE=Sxm and ACK_IN=1))
        else
            SETOUTPUTCONDITION(SignalTarget, STATE=Syi)
        end
    end
    CASE{ Data_Hold(latency=i) }begin
        SETOUTPUTCONDITION(SignalTarget, STATE=Sxi or ... or STATE=Sxm or STATE=SA)
    end
```

図 14　送信側 FSM の生成手法の擬似コード

メータ DEST は対応する COND の値が 1 となった時の遷移先を表す。DEST=this は現状態にとどまることを、DEST=next は STATELIST の格納順で次の状態へ遷移することを、DEST=next2 はさらに次の状態へ遷移することを示す。また、関数 SETOUTPUTCONDITION (SignalOut, Condition) は、出力信号線 SignalOut が 1 を出力する条件を Condition で表現する。

FSM テンプレートの加工処理は、状態数変更の処理と信号出力関係の処理の 2 つから構成される。まず、1 シーケンス中の全てのタイミング記述に対して状態数変更の処理を行い、その後に個別の Data_Hold() 文、Data_Oneshot() 文に対して信号出力の処理を行う。なお、送信側 FSM テンプレートでは、latency の値が負である Data_Oneshot() 文は無視して扱わない。

ここからは、受信側 FSM テンプレートについて説明する。まず、受信側テンプレートを図 15 に示す。このテンプレートは、マスタコントローラのブロッキング処理 FSM・パイプラインリクエスト処理 FSM とスレーブコントローラのパイプラインレスポンス処理 FSM に適用する。以降の説明では、START_IN を入力信号、ACK_IN を出力信号として、Handshake(START_IN, ACK_OUT) または OverlapHandshake(START_IN, ACK_OUT) または Data_Oneshot(START_IN, START_IN, 0) が宣言されているとして説明を行う。

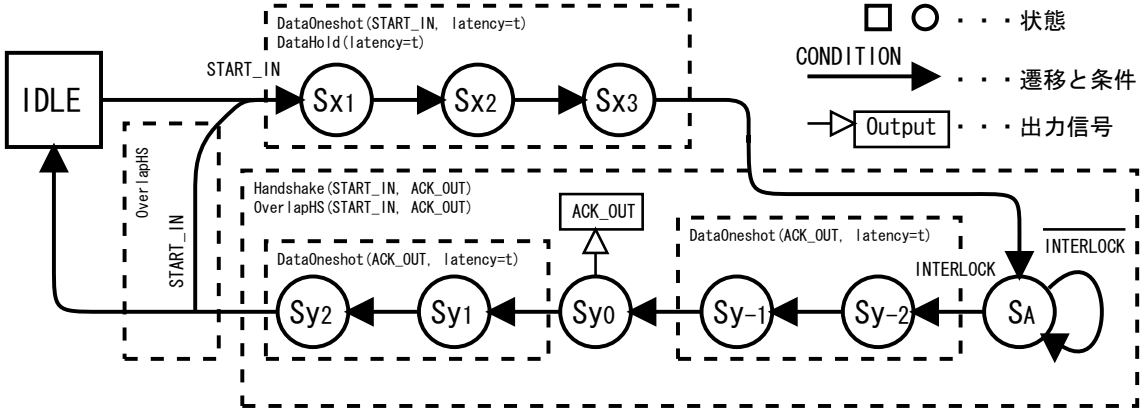

図 15　受信側 FSM テンプレート

図 16 に、受信側 FSM テンプレートに基づいて FSM を生成する手順を擬似コードで示す。擬似コード中の関数などの意味は、送信側 FSM 生成手法のものと同じである。受信側 FSM は、バッファ内のデータ数をもとに待機状態 S_A から遷移する。テンプレートでは、この遷移条件を論理式 INTERLOCK で表現する。INTERLOCK の各 FSM での詳細は、第 3.5.6 節で説明する。受信側 FSM テンプレートでは、Handshake() 文あるいは OverlapHandshake() 文が宣言されている場合に、SignalEnd を基準とする Data_Oneshot() 文の latency が負であることを許可する。

3.5.6　ラッパの構造に依存した処理

第 3.5.4 と第 3.5.5 に示した処理によって、プロトコル表現の内容を反映した FSM を機械的に生成することができる。しかし、生成した FSM がラッパコントローラ FSM として動作するようにするためには、ラッパコントローラやバッファの構造に大きく依存した入出力を取り扱う必要がある。

まず、FSM テンプレート中で遷移条件として用いられている論理式 INTERLOCK について説明する。表 3 に、本研究で行った実験で論理式 INTERLOCK に適用した式を示す。論理式 INTERLOCK は、マスタコントローラでは、レスポンスが返ってきたか、次のリクエストを受けるための空き容量がバッファにあるかを判定し、スレーブコントローラでは、発行するべきリクエストがバッファ内にあるかを判定している。現在のところ、リクエストの発行者の都合でレスポンスを遅らせるようなプロトコルは想定していないので、スレーブコントローラのパイプラインレスポンス処理 FSM は INTERLOCK を無条件遷移としている。

次に、バッファの読み書き操作に関してラッパコントローラやバッファの構造に依存した処理について説明する。表 4 に、本研究において、各ラッパコントローラ FSM が操作するバッファの端子の一覧を示す。データを伝達するチャネルでは、第 3.4 節で述べた理由により、チャネルに書き込むべきデータがない場合でも仮のデータを書きこまなければならない。この要求は、Data_Oneshot(latency=0) を初期制約として与えておき、プロトコル表現内でデータ線に Data_Hold() 文または Data_Oneshot() 文が宣言された場合に初期制約を削除することによって実現する。また、列挙子を伝達するチャネルを操作するタイミングはプロトコル表現からは与えられないため、書き込み端子に有効値を出力するタイミングを予め定義しておく必要がある。実験では、Reqbuff_Command チャネルに対しては Data_Oneshot(latency=0) で書き込みを行い、Resbuff_Status チャネルに対しては Resbuff_Data チャネルと同時に書き込みを行い、ResExist チャネルに対しては FSM における最後の状態で書き込みを行うことにした。ここで、最後の状態とは、次のサイクルで

```
[FSM 生成ルーチン]
    CALL 状態数変更ルーチン
    SETOUTPUTCONDITION(START_OUT, STATE=Sx0)
    FOR-EACH{ Data_Hold(), Data_Oneshot }begin
        CALL 信号出力ルーチン
    end
[状態数変更ルーチン]
    STATELIST = IDLE<INTERLOCK:next, !INTERLOCK:this>

    FOR-ALL{ Data_Hold(latency=i) or Data_Oneshot(SignalTrigger=START_IN, latency=i) }begin
        m = get_maximum(i)
    end
    FOR-ALL{ Data_Oneshot(SignalTrigger=SignalAck, latency=i) }begin
        p = get_maximum(i)
        n = get_minimum(i)
    end

    STATELIST_PUSH_BACK Sx1<1:next>, Sx2<1:next>, Sx3<1:next>, ..., Sxm<1:next>

    IF-DECLARED{ Handshake(START_IN, ACK_OUT) or OverlapHandshake(START_IN, ACK_OUT) }begin
        STATELIST_PUSH_BACK  SA<INTERLOCK:next, !INTERLOCK:this>
        IF{ n<0 } STATELIST_PUSH_BACK  Syn<1:next> Syn+1<1:next> ... Sy-1<1:next>;
        STATELIST_PUSH_BACK  Sy0<1:next> Sy1<1:next> ... Syp<1:next>
    end

    IF-DECLARED{ OverlapHandshake(START_IN, ACK_OUT) }begin
        STATELIST_POP_LAST_ELEMENT to Stemp
        STATELIST_PUSH_BACK  Stemp <START_IN:STATELIST_ELEMENT(2), !START_IN:next>
        STATELIST_PUSH_BACK  IDLE
    else
        STATELIST_PUSH_BACK  IDLE
    end

[信号出力ルーチン]
    CASE{ SignalTarget's Direction is input }begin
        IGNORE_STATEMENT
    end
    CASE{ Data_Oneshot(SignalTrigger=SignalStart, latency=i) }begin
        IF{ i=0 }begin
            SETOUTPUTCONDITION(SignalTarget, STATE=IDLE and START_IN=1)
        else
            SETOUTPUTCONDITION(SignalTarget, STATE=Sxi)
        end
    end
    CASE{ Data_Oneshot(SignalTrigger=SignalEnd, latency=i) }begin
        IF{ i=0 }begin
            SETOUTPUTCONDITION(SignalTarget, STATE=Syi)
        end
    end
    CASE{ Data_Hold(latency=i) }begin
        IF{ i=0 }begin
            SETOUTPUTCONDITION(SignalTarget, (STATE=IDLE and START_IN=1)
                                             or STATE=Sx1 or ... or STATE=Sxm
                                             or STATE=SA or STATE=Syn or ... or STATE=Sy0)
        else
            SETOUTPUTCONDITION(SignalTarget, STATE=Sxi or ... or STATE=Sxm
                                             or STATE=SA or STATE=Syn or ... or STATE=Sy0)
        end
    end
```

図 16　受信側 FSM の生成手法の擬似コード

IDLE 状態に遷移する状態のことである。さらに、読み出し端子を操作するタイミングについても予め定義しておく必要がある。実験では、状態遷移図における最後の状態でバッファの読み出しを行うことにした。ただし、Resbuff_Data チャネルと Resbuff_Status チャネルを読み出すのは、有効なレスポンスが利用できる場合だけであり、かつ、データの同期を保つためには Resexist のチャネルと同時に読み出しを行うことが望ましいので、Resbuff_rd 端子に有効値を出力する条件は Resexist_rd∧(Resexist=YES) とした。

表 3　各ラッパコントローラ FSM における論理式 INTERLOCK の例

FSM 種別	使用テンプレート	INTERLOCK の式 (Verilog 表記)
マスタコントローラ		
ブロッキング	受信側	`(!Resexist_zero&&(Resexist==NO))||` `(!Resexist_zero&&(Resexist==YES)&&!Resbuff_zero)`
パイプラインリクエスト	受信側	`(!Reqbuff_full)`
パイプラインレスポンス	送信側	`(!Resexist_zero&&(Resexist==NO))||` `(!Resexist_zero&&(Resexist==YES)&&!Resbuff_zero)`
スレーブコントローラ		
ブロッキング	送信側	`(!Reqbuff_zero)`
パイプラインリクエスト	送信側	`(!Reqbuff_zero)`
パイプラインレスポンス	受信側	無条件遷移 `(1'b1)`

表 4　本研究でラッパコントローラ FSM が操作するバッファの端子の一覧

FSM 種別	FSM が操作する端子のリスト
マスタコントローラ	
ブロッキング	Reqbuff_Command_wr, Reqbuff_Addr_wr, Reqbuff_Data_wr Resbuff_rd, Resexist_rd
パイプラインリクエスト	Reqbuff_Command_wr, Reqbuff_Addr_wr, Reqbuff_Data_wr
パイプラインレスポンス	Resbuff_rd, Resexist_rd
スレーブコントローラ	
ブロッキング	Resbuff_Status_wr, Resbuff_Data_wr, Resexist_wr, Reqbuff_rd
パイプラインリクエスト	Resexist_wr
パイプラインレスポンス	Resbuff_Status_wr, Resbuff_Data_wr, Reqbuff_rd

3.5.7　各シーケンスの FSM を結合する方法

各シーケンスに対応した FSM は IDLE 状態からスタートして、IDLE 状態に戻ることで終了するので、IDLE 状態を共有することで、各シーケンスに対応した FSM を結合することができる。この時、処理すべきシーケンスを選択する仕組みが必要になる。

ブロッキング処理 FSM と、パイプラインリクエスト処理 FSM の場合は以下に述べる手順で FSM を結合することができる。まず、マスタコントローラの場合は、固有プロトコルのシーケンスの数だけ FSM が生成される。エンコーディングテーブルに、シーケンス選択信号を生成する方法が記載されているので、コンバーターで生成されたシーケンス選択信号をもとに、処理すべきシーケンスを決定する。次に、スレーブコントローラの場合は、Command チャネルの列挙子の種類の数だけ FSM が生成される。リクエストバッファの Command チャネルの列挙子から、処理すべきシーケンスを決定する。各シーケンスへの分岐を実現するためには、IDLE 状態から遷移する条件を（開始条件）から（開始条件）∧（シーケンス分岐条件）に変更する。

図 17　パイプライン処理 FSM に対する処理シーケンス指定方法

OverlapHandshake() 文が宣言されている場合は、IDLE 状態を通らずに次のシーケンスを開始する遷移に対しても、条件の書き換えを行う。

パイプラインレスポンス処理 FSM を結合する場合は、上記の手順とは異なる方法が必要である。まず、スレーブコントローラのレスポンス FSM を結合する場合を説明する。本論文では、空ではないタイミング記述から生成された FSM が 1 種類しかない場合を想定する。これは、レスポンスの処理の手順がリクエストによって異なることに合理的な理由がないため、多くの実用的なプロトコルで、生成される FSM が 1 種類であるためである。FSM が 1 種類しか作られない時、生成された 1 つの FSM をスレーブコントローラのレスポンス処理 FSM とする。複数種類の FSM が生成される場合は、本論文では対象外とする。

次に、マスタコントローラのパイプラインレスポンス FSM を結合する方法について説明する。この場合は、各シーケンスに対応した FSM を IDLE 状態を共有する形で結合する。ここで、パイプラインレスポンス処理 FSM が処理するべきシーケンスを指定するために、図 17 に示すように FIFO レジスタを置く。この FIFO レジスタは、コンバーターで生成されたシーケンス選択信号を要素として格納する。FIFO レジスタの書き込み信号は、リクエスト処理 FSM の開始信号を使用する。すなわち、リクエスト FSM の動作開始時に FIFO レジスタに書き込みが行われる。FIFO の先頭要素は、レスポンス処理 FSM が終了する際に破棄する。

マスタコントローラのパイプラインレスポンス処理 FSM を結合する際に、このような機構を適用するのはパイプラインプロトコル同士を変換する際にレスポンスが正しく処理されることを保証するためである。一例として、読み書き両方のリクエストにレスポンスを返すことを期待するマスタコアと読み出しリクエストにしかレスポンスを返さないスレーブコアの間にラッパを挿入する場合を考える。この場合、スレーブコアからのレスポンスを単純に転送するだけでは、書き込みリクエストに対するレスポンスが返らないので、マスタコアは正しくレスポンスを受信することができない。このため、受けたリクエストを記憶しておいて正しくレスポンスを返す仕組みが必要になる。

4　実験

3 章で述べた手法に従い、プロトコル仕様記述の作成およびラッパ生成の実験を行った。この章では、実験の手順と結果を示す。なお、実験では、ハードウェア記述言語として、Verilog を使用した。

4.1　ラッパコントローラ生成ツールの実装

実験を行うにあたって、図 9 の手順のうち、端子リストからラッパコントローラの入出力端子の Verilog 記述を生成する処理を入出力宣言生成ツールとして実装し、タイミング表現から FSM の Verilog 記述を生成する処理を FSM 生成ツールとして実装した。プログラム言語は C++ を用いた。入出力宣言生成ツールが約 300 行、FSM 生成ツールが約 730 行である。実装された FSM 生成ツールでは、マスタコントローラのラッ

パコントローラ FSM を生成することができる。ただし、latency が負の値をとる Data_Oneshot() 文は扱うことができない。

4.2 例題

実験では、OCP マスタコントローラ、OCP スレーブコントローラ、AHB マスタコントローラの 3 つを生成し、生成したラッパコントローラを使用して、OCP → OCP ラッパと AHB → OCP ラッパを生成した。ここで、AHB プロトコルとは、AMBA プロトコルで定められた通信プロトコルの一つである。本実験で用いた AHB プロトコルは、1 回のデータ転送幅は 32 ビットで固定とし、マスタコントローラが返すレスポンスは、OKAY レスポンスと ERROR レスポンスの 2 種類とした。その他の条件は、AMBA プロトコル仕様書に従った。また、本章で用いる AHB の信号線名、列挙子と信号線の値の対応は、全て AMBA プロトコル仕様書に記載されている内容に従う。本実験で用いた OCP は、BasicSignals セットから、リードとライトのリクエストとレスポンスのみを扱うものとした。その他の条件は、OCP 仕様書に従った。また、本章で用いる OCP の信号線名、列挙子と信号線の値の対応は、全て OCP 仕様書に記載されている内容に従う。

4.3 実験の条件

図 9 の手順にのっとって、OCP マスタコントローラ、OCP スレーブコントローラ、AHB マスタコントローラの生成を行った。入出力端子宣言の記述を生成する作業は全ての事例で実装したツールで行い、コンバータの記述を生成する作業は全ての事例で人手で行った。タイミング表現から FSM を生成する作業では、OCP マスタコントローラの生成作業ではツールを用いて FSM を生成し、他の 2 つのラッパコントローラの生成作業では、手作業で FSM を生成した。ラッパコントローラの生成後、OCP マスタコントローラと OCP スレーブコントローラを組み合わせて、OCP → OCP ラッパを生成し、AHB マスタコントローラと OCP スレーブコントローラを組み合わせて、AHB → OCP ラッパを生成した。それぞれのラッパに対し、マスタコアとスレーブコアを接続し、正しく通信できることをシミュレーションで確認した。なお、シミュレーションで用いたマスタコアとスレーブコアの記述は、ラッパの動作確認を行うために人手で作成した Verilog 記述である。

また、生成したラッパコントローラが AHB、OCP に準拠しているか確認するために、各プロトコルのプロパティチェッカを使用して検証を行った。ここで、プロパティチェッカとは、設計記述を数学的に検証して、設計記述がプロパティを満たしているかを判定するツールである。プロパティとは、設計記述が満たすべき仕様を論理式の形で記したものである。今回の実験では、TransEDA 社 [18] の提供する imPROVE-HPK-AHB と imPROVE-HPK-OCP を使用した。imPROVE-HPK では、AHB 及び OCP に準拠した設計記述が満たすべきプロパティが予め入力されており、設計記述の動作がプロトコルに違反している場合には反例として、プロトコルに違反する動作のタイミングチャートを出力する。

4.4 実験結果

提案する表現法でプロトコル仕様を表現した結果を図 18、19 に示す。前者は OCP スレーブコントローラ生成用に作成した記述で、後者は AHB マスタコントローラ用に作成した記述である。ここには記さないが、OCP マスタコントローラ生成用のプロトコル仕様も同様にして書くことができた。OCP 用のラッパコントローラは提案表現法に基づくプロトコル仕様記述から生成することができたが、AHB マスタコントローラ生

成用のプロトコル仕様から FSM を生成する際には、手法に従って生成された FSM に対して以下のような理由で修正を行う必要があった。AHB プロトコル仕様に対応するためには、リード・ライトのシーケンスの他に、IDLE 転送・BUSY 転送のシーケンスも定める必要があるが、IDLE 転送・BUSY 転送ではデータの送受信は行われないと AMBA 仕様書に定義されている。このようなシーケンスに対して提案するラッパコントローラ FSM 生成手法を適用すると、第 3.5.6 節で述べた処理によって不正なリクエストをバッファに書き込む回路が生成され、ラッパが正常動作できなくなる。よって、今回の実験では、リクエストバッファに不正な書き込み指示を出す信号線を追加作業で削除した。この問題は、FSM テンプレートの中でラッパの構造に依存している部分を修正することによって対処することができる。

表現の簡潔さを比較する指標として、本実験では提案表現の記述行数と、生成された Verilog での設計記述の記述行数を比較した。表 5 に、各ラッパコントローラにおける、提案する表現法で記したプロトコル仕様の記述行数と提案する生成手順で生成された設計記述の記述行数を示す。本実験では、提案するプロトコル表記法の 4 ～ 5 倍の設計記述が生成された。

また、バッファの Verilog 記述が 106 行、ラッパコントローラ・バッファを接続するための配線部の記述が OCP→OCP ラッパで 112 行、AHB→OCP ラッパで 129 行であった。であった。これらを加えたラッパの総記述行数を表 6 に示す。両方のラッパコントローラを新しく作る場合は 6 ～ 7 倍、片方のラッパコントローラをライブラリから取得する場合は 10 ～ 14 倍程度の記述が生成される。提案する表現法による AHB プロトコルの記述行数が OCP に比べて多いのは、AHB で扱った入出力信号線の数が OCP に比べて多いためである。

表 5　提案する表現法でのプロトコル仕様記述と Verilog での設計記述の行数比較

ラッパコントローラ名	提案する表現法による行数	Verilog 記述行数	生成行数/仕様行数
AHB マスタコントローラ	48	195	4.1
OCP マスタコントローラ	32	189	5.9
OCP スレーブコントローラ	34	157	4.6

表 6　生成したラッパ記述の Verilog による総行数

ラッパ名	ラッパ記述の総行数	生成行数/仕様行数
AHB → OCP ラッパ	481	5.9
OCP → OCP ラッパ	458	6.9

5　本研究のまとめ

本研究では、IP インターフェースプロトコルの簡潔な表現方法と、その表現方法を入力情報として、ラッパを生成する手法を提案した。提案する表現方法では、インターフェースプロトコルの仕様書をもとにして、プロトコルの仕様を表現する。本研究では、

- データ線とコントロール線を分離して扱うこと
- 信号線間のタイミング関係と信号線のエンコーディングを分離して扱うこと
- タイミング関係を少ない種類のステートメントで表現すること

```
記述対象:OCP スレーブコントローラ

[入出力信号リスト]
信号名（ビット幅，ドライバ，種別，チャネル名）
  Clk        ( 1,      外部,      クロック, )
  Reset_n    ( 1,      外部,      リセット, )
  MCmd       ( 3,    マスタ, コントロール, )
  MAddr      (32,    マスタ,        データ, Reqbuff_addr)
  MData      (32,    マスタ,        データ, Reqbuff_data)
  SCmdAccept( 1, スレーブ, コントロール, )
  SResp      ( 2, スレーブ, コントロール, )
  SData      (32, スレーブ,        データ, Resbuff_data)

[シーケンスの表記]
シーケンス名:ステートメント
  ライトリクエスト : Handshake(ocpMCmd_A, ocpSCmdAccept)
  ライトリクエスト : Data_Hold(ocpMCmd_A, 0)
  ライトリクエスト : Data_Hold(ocpMAddr, 0)
  ライトリクエスト : Data_Hold(ocpMData, 0)
  ライトレスポンス : --no statement--
  リードリクエスト : Handshake(ocpMCmd_A, ocpSCmdAccept)
  リードリクエスト : Data_Hold(ocpMCmd_A, 0)
  リードリクエスト : Data_Hold(ocpMAddr, 0)
  リードレスポンス : Data_Oneshot(ocpSResp_A, ocpSResp_A, 0)
  リードレスポンス : Data_Oneshot(ocpSData, ocpSResp_A, 0)

[エンコーディングテーブル表記]
信号線の有効値 信号名（有効値）
  ocpResetn    (0)
  ocpSCmdAccept(1)
  ocpMCmd_A    (1)
  ocpSResp_A   (1)
多ビット線の扱い タイミング表記の信号線:固有プロトコルの信号線
  ocpMCmd_A  : ocpMCmd
  ocpSResp_A : ocpSResp
信号出力テーブル 信号名 = 値 {条件}
default は、それまでに述べたどの条件にも当てはまらないことを示す。
  ocpMCmd = WR {(ocpMCmd_A=1) and (Reqbuff_Command=WR)}
  ocpMCmd = RD {(ocpMCmd_A=1) and (Reqbuff_Command=RD)}
  ocpMCmd = IDLE {default}
  ocpSResp_A = 1 {ocpSResp!=NULL}
  ocpSResp_A = 0 {ocpSResp=NULL}
  Resbuff_Status = ERR {ocpSResp=ERR}
  Resbuff_Status = OK {default}
```

図 18　提案表現法による OCP スレーブコントローラの仕様記述

```
記述対象:AHB マスタコントローラ

[入出力信号リスト]
信号名（ビット幅，ドライバ，種別，チャネル名）
  HCLOCK    ( 1,      外部,      クロック, )
  HRESETn   ( 1,      外部,      リセット, )
  HSEL      ( 1,    マスタ, コントロール, )
  HWRITE    ( 1,    マスタ, コントロール, )
  HREADY    ( 1,    マスタ, コントロール, )
  HTRANS    ( 2,    マスタ, コントロール, )
  HSIZE     ( 3,    マスタ, コントロール, )
  HBURST    ( 3,    マスタ, コントロール, )
  HADDR     (32,    マスタ,        データ, Reqbuff_addr)
  HWDATA    (32,    マスタ,        データ, )
  HREADYOUT( 1, スレーブ, コントロール, Reqbuff_data)
  HRESP     ( 2, スレーブ,        データ, )
  HRDATA    (32, スレーブ,        データ, Resbuff_data)

[シーケンスの表記]
シーケンス名:ステートメント
  ライト    : OverwrapHandshake(HSTART, HREADYOUT1)
  ライト    : Data_Oneshot(HREADYOUT2, HSTART, 0)
  ライト    : Data_Oneshot(HADDR, HSTART, 0)
  ライト    : Data_Oneshot(RespValid, HREADYOUT1, -1)
  ライト    : Data_Oneshot(RespValid, HREADYOUT1, 0)
  リード    : OverwrapHandshake(HSTART, HREADYOUT2)
  リード    : Data_Oneshot(HREADYOUT2, HSTART, 0)
  リード    : Data_Oneshot(HADDR, HSTART, 0)
  リード    : Data_Oneshot(RespValid, HREADYOUT1, -1)
  リード    : Data_Oneshot(RespValid, HREADYOUT1, 0)
  アイドル  : Data_Oneshot(HREADYOUT2, HSTART, 0)

[エンコーディングテーブル表記]
信号線の有効値 信号名（有効値）
  HRESETn   (0)
  HSTART    (1)
  HREADYOUT1(1)
  HREADYOUT2(1)
  RespValid (1)
多ビット線の扱い 固有プロトコルの信号線:タイミング表記の信号線
  HSTART     : HREADY, HSEL
  HREADYOUT1 : HREADYOUT
  HREADYOUT2 : HREADYOUT
信号出力テーブル 信号名 = 値 {条件}
default は、それまでに述べたどの条件にも当てはまらないことを示す。
HSTART = 1 {(HREADY=1)and(HSEL=1)}
HSTART = 0 {default}
HREADYOUT = 1 {(HREADYOUT1=1)or(HREADYOUT2=1)}
HREADYOUT = 0 {default}
HRESP = ERR {(RespValid=1)and(ResExist=YES)and
             (Resbuff_Status=ERR)}
HRESP = OKAY {default}
ReqBuff_Command = WR {HWRITE=1}
ReqBuff_Command = RD {default}
シーケンス = ライト {((HTRANS=NOSEQ)or(HTRANS=SEQ))
                     and(HWRITE=1)}
シーケンス = リード {((HTRANS=NOSEQ)or(HTRANS=SEQ))
                     and(HWRITE=0)}
シーケンス = アイドル {default}
```

図 19　提案表現法による AHB マスタコントローラの仕様記述

を特徴とする、プロトコルの表現方法を定義した。また、提案するラッパ生成手法では、提案するプロトコル表現方法をもとに、機械的にラッパの HDL 記述を生成する。ラッパの合成手法はライブラリベースのラッパ構成法をもとにして、

- 各ラッパコントローラは FSM とコンバーターの 2 つのブロックで構成されること
- 信号線のエンコーディングの違いをコンバーターで取り扱うこと

- タイミング表現から、FSM テンプレートを用いてタイミング制御用 FSM を生成すること

を特徴とする手法を用いた。

本手法を、OCP と AHB プロトコルに対して適用し、生成したラッパが正常に動作することを確認した。また、本手法を適用して生成したラッパに対し、提案する手法で書いたプロトコル仕様の行数と、生成したラッパの行数を比較したところ、両方のラッパコントローラを新しく製作した場合は $\frac{1}{6}$ 程度、片方のラッパコントローラを既存の記述から用意した場合は $\frac{1}{10}$ 程度に記述量を削減できることが分かった。

6 今後の課題

6.1 ラッパ生成手法の実装

現在のところ、提案するラッパ生成手法は、FSM を生成する作業が部分的に実装されているのみで、提案するプロトコル表現法からラッパを生成する作業の多くの部分は手動で行われている。今後、提案するラッパ生成手法を実行するプログラムを完成させ、ラッパ生成手法を自動化する。

6.2 異なるクロック間の変換を行うラッパの生成

本論文で提案したラッパ生成手法は、マスタコントローラ、スレーブコントローラが同じクロックに同期していることを仮定しており、実験では、ラッパコントローラ及びマスタコア、スレーブコアが同じクロックで動く環境で動作確認を行った。しかし、実際の設計では、LSI 上の全ての領域が単一のクロックに同期しているとは限らず、バスと IP の間で動作クロックが異なる場合も多くあるため、マスタコントローラとスレーブコントローラが異なるクロックに同期していてもリクエスト・レスポンスの送受信が正しくできることが望ましい。今後、異なるクロック間でもプロトコルの変換が行えるラッパの構造を提案し、正しく動作することを確認する。

6.3 プロトコル表現法の拡張

本論文で提案したプロトコル表現方法では、シーケンス内で分岐があるようなプロトコルを表現することができない。このため、正常処理とエラーがあった場合の処理の制御フローが全く異なるプロトコルなどは提案するラッパ生成手法で扱うことができない。この問題に対処するために、プロトコル表現のステートメントに分岐を扱うステートメントを追加したいと考えている。今後、分岐を扱うステートメントの表記方法と、分岐の導入による FSM 生成作業の変更点について検討を進める。

謝辞

この場をお借りして、本研究を行うにあたってお世話になった皆様にお礼を申し上げます。

指導教員にあられます藤田昌宏先生には、研究を行うにあたって研究テーマの選定、研究の方針、論文作成などにおいて多大なご指導を頂きました。先生に心より感謝申し上げます。

小松聡助手は、研究室の運営全般を通して私たちを支えて下さいました。また、進捗発表の折には、いつも的確なアドバイスを下さいました。

瀬戸謙修氏には、研究に行き詰まり、方針を見失った際に何度も助言を頂きました。また、高位合成の研究について様々な説明を頂きました。

Subash Shankar 氏、Sakunkonchak Tyanyapat 氏、劉宇氏は、部屋が隣ということもあって、親しく話す機会は少なかったのですが、お茶の時間に御一緒した時は楽しい話を聞かせて下さいました。

高尚華氏は、深夜や休日の研究室で研究に励む姿を見せて下さいました。真後ろの席に座る私は、氏の頑張る後ろ姿を文字通りに眺めて気力を奮い立たせました。

松本剛史氏は、研究進捗の発表時に、研究の不明瞭な点について多くの的確な指摘を下さいました。その一方で、普段の研究生活では卒論生である我々に気さくに話しかけてくださり、研究室の雰囲気になじめるように配慮して下さいました。

佐々木俊介氏は、研究・プライベートの両面で様々な話題・機会を提供して下さいました。また、システム管理者として、研究室の計算機環境を整えて下さいました。研究が快適に進められたのは、小松助手・佐々木氏をはじめとするシステム管理者の皆様の御力のおかげであります。

松井健氏には、研究生活全般において様々な助言を頂きました。人に対して説明をするのが苦手な私にとって、資料の作り方やプレゼンテーションのやり方についての助言は非常に有益なものでした。

西原佑氏からは、昨年藤田研究室の卒論生だったこともあり、研究生活について様々な助言を頂きました。氏の、独特な美的センスのあふれる台詞は大変に印象深いものでありました。

秘書の大口智子氏、上出芽衣子氏は、研究室の備品の買出しやその他の様々な書類手続きでお世話になりました。卒論生中で私だけが電子情報工学科所属であるため、書類の作成時に迷惑をかけたことがあったかも知れません。また、お茶の時間には研究室の皆においしいお茶を淹れて下さいました。

１年間私を支えてくださった先生方・先輩方に感謝いたします。

安藤大介君は研究の端々で私にはないセンスを発揮しており、彼の存在は研究生活の励みでした。また、彼の語るボウリングや株の話は、門外漢の私にとっても非常に面白いものでした。

後藤真毅君の研究テーマは私の研究テーマと重なるところもあり、彼の研究は大変興味深いものでした。

高飛君は、学部実験の頃から同じ班で付き合いが長いこともあり、気兼ねなく接することができました。

この１年間を共に過ごした卒論生の皆に感謝します。

最後に、渡邊翔太氏に深くお礼申し上げます。渡邊氏は、私と異なる方法論によるプロトコル変換器生成手法を提案しており、研究を進めるにあたって渡邊氏の研究は大変参考になりました。また、研究テーマの選択から卒論の執筆に至るまで、本研究全般に関してご指導を頂きました。重ねて、お礼申し上げます。

参考文献

[1] "The International Technology Roadmap for Semiconductors" Technology Needs, 1997 ed. San Mateo, CA: Semiconductor Industry Association, 1997

[2] SpecC (http://www.cecs.uci.edu/~specc/)

[3] SystemC (http://www.systemc.org/)

[4] Reinaldo A. Bergamaschi, William R. Lee, "Desinging Systems-on-Chip Using Cores" DAC'00, Los Angeles, California

[5] AMBA Specification 2.0 (http://www.arm.com/products/solutions/AMBA_Spec.html)

[6] Core Connect white paper (http://www-03.ibm.com/chips/products/coreconnect/)

[7] Sonics Inc. SonicsStudio, SoCCreator (http://www.sonicsinc.com/sonics/products/sonicsstudio)

[8] Actel Inc. CoreConsole (http://www.actel.com/products/ARM7/)

[9] Xilinx Inc. EDK (http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm)

[10] OpenCoreProtocol International Partnership (http://www.ocpip.org/)

[11] OpenCoreProtocol Specification ver 2.11

[12] Roman L. Lysecky, Frank Vahid, Tony D. Givargis, "Techniques for Reducing Read Latency of Core Bus Wrappers" proc. Design Automation and Test in Europe, pp. 84 – 91, Mar. 2000.

[13] Ferid Gharsalli, Samy Meftali, Frederic Rousseau, Ahmed A. Jerraya, "Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC" DAC'02,New Orleans, Louisiana

[14] Roberto Passerone, James A. Rowson, Alberto Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols" DAC'98 SanFrancisco, California

[15] Shota Watanabe, Kenshu Seto, Yuji Ishikawa, Satoshi Komatsu, Masahiro Fujita, "Automatic Protocol Transducer Synthesis aiming at facilitating IP-Reuse" to be appeared

[16] Vijay D'Silva, S.Ramesh, Arcot Sowmya, "Bridge Over Troubled Wrappers : Automated Interface Synthesis" Proceedings of the 17th International Conference on VLSI Design

[17] James Smith, Giovanni De Micheli, "Automated Composition of Hardware Components" DAC'98 SanFrancisco, California

[18] TransEDA Inc. imPROVE-HPK-OCP, imPROVE-HPK-AHB (http://www.transeda.com/)