特別研究報告書

高速通信ボード CNIC における プロトコル・プロセッサ・インタフェースの 設計

指導教官　　富田 眞治 教授

京都大学工学部情報学科

石川 智祥

平成 13 年 2 月 13 日

高速通信ボード CNIC における プロトコル・プロセッサ・インタフェースの設計

石川 智祥

内容梗概

最近のパーソナルコンピュータやワークステーションの低価格化に伴い、並列処理を行う環境としてコストパフォーマンスの高い分散システムが注目を集めている。しかし現在の分散システムでは通信機能が付加的であるため、通信コストが高く、処理の並列化を目指したスケジューリング機能も十分とは言えない。

そこで我々は現在の分散システムが持つ長所を生かしつつこれらの問題点を解決し、システム全体のコンピュータ資源を最大限に活用するシステム「コンピュータ・コロニー」を提案する。

コンピュータ・コロニーでは通信コストを削減し高速通信を図るハードウェアと、適切なスケジューリングによってシステム内のコンピュータ資源の有効利用を図るオペレーティングシステム Colonia からなる。このようにハードウェアとソフトウェアの両面から最適なシステムを目指して研究を進めている。

現在我々は研究室においてコンピュータ・コロニー実現に向け、ワークステーションのシステムバスにネットワーク・インタフェースとなる専用高速通信ボード CNIC を装備したプロトタイプ・ハードウェアを開発中である。プロトタイプ・ハードウェアでは、システムコールを介さない通信と、プロトコルオーバヘッドの削減、そしてリモートメモリのキャッシングによって高速通信を目指している。

CNIC がシステムバスに接続されていることで、オペレーティングシステムによるシステムコールを介さずに直接リモートメモリアクセスを行うことができる。CNIC は常にシステムバスを流れる物理アドレスを監視し、リモートメモリへのアクセスの必要性があると CNIC 自身がリモートノードとの通信を行う。

さらに CNIC 上にプロトコル・プロセッサと FPGA (Field Programmable Gate Array) を搭載した。そのためプロトコル・プロセッサに対して様々なプログラムを与え、FPGA 内に様々な機能を実装することで、プロトタイプとしていろいろな通信方式を評価することができる。

本研究ではプロトコル・プロセッサ・インタフェースの設計および実装を行っ

た。CNICを介した高速通信を実現させるために、プロトコル・プロセッサは主に物理アドレスとネットワークアドレスとの変換、およびその際必要とされるページテーブルの管理を行う。プロトコル・プロセッサ・インタフェースはプロトコル・プロセッサを適切にコントロールし、プロトコル・プロセッサにこのような処理ができるような環境を提供する。

プロトコル・プロセッサ・インタフェースの基本機能は、プロトコル・プロセッサのメモリアクセスの要求に対してROMあるいはRAMへのREAD, WRITEを制御し、適切にデータの受け渡しが行われるようにすることである。プロトコル・プロセッサ・インタフェースはROM, RAM, 各種制御部およびそれらとプロトコル・プロセッサとをつなぐインタフェース部からなる。

プロトコル・プロセッサからのメモリアクセスが発生すると、どのROM, RAMが選択されているかを判断し、対応するROM, RAMに対してREADであればデータバスへデータを出力、WRITEであればデータバスからデータを入力するよう指示する。その後適切なタイミングでレディーをアサートし、出力あるいは入力の完了をプロトコル・プロセッサに知らせる。

まずROM, RAM, 各制御部などをFPGA内に設計し、プロトコル・プロセッサに簡単なプログラムを与えて実行させた。ROMにはプロトコル・プロセッサのブート時に必要なアドレス空間などの設定をするための命令を格納しておく。CNIC上にはFPGA内の信号をスヌープできる端子が装備されており、そこを観察することでプロトコル・プロセッサが期待通り動作していることを確認した。

次にFPGA内に設計したRAMの替わりに、CNIC上の大容量かつ高速なSynchronous SRAMをプロトコル・プロセッサが使うことができるように設計した。先ほどと同様テストプログラムを与えてCNIC上の端子を観察し、プロトコル・プロセッサが期待通り動いていることを確認した。このようにプロトコル・プロセッサがSynchronous SRAMにアクセスすることができるようになると、プロトコル・プロセッサに大規模なプログラムを与えて実行させることができるようになりCNICによる通信をはじめ、様々な評価を行うことができるようになる。

Design of the Protocol Processor Interface on the High-Speed Communication Board CNIC

Tomoyoshi ISHIKAWA

Abstract

As prices for PC and Workstations lower, distributed system that enables us to conduct parallel processing has gained more attention than ever before. Yet, under existing systems, its supplementary communication function results in a high communication cost and an imperfection of the scheduling function for parallel processing.

Therefore, we propose "Computer Colony," the system that makes the most use of computer resources in the whole system by adopting the advantages and, at the same time, by solving the drawbacks in existing distributed systems.

Computer Colony consists of the two parts: a hardware designed for a high-speed communication resulting from a reduction of communication cost, and an operating system "Colonia" designed for a better utilization of computer resources in a system by an adequate scheduling. Thus, our research aims to develop the best system on both hardware and software side.

We are now developing a prototype hardware of Computer Colony with a high-speed communication board CNIC as a network interface connected to the system bus of a workstation. This prototype hardware is designed for a high-speed communication by blocking system calls, reducing protocol overheads, and caching pages of remote memories.

The connection of CNIC to the system bus makes it possible to directly access to remote memories without any system calls by the operating system. CNIC always observes physical addresses in the system bus. In need of an access to remote memories, CNIC itself communicates with a remote node.

Furthermore, we installed a protocol processor and a FPGA (Field Programmable Gate Array) in CNIC. Therefore, we are able to evaluate many communication methods as a prototype by running various programs in the protocol processor and by implementing various functions in the FPGA.

I have designed the protocol processor interface. In order to realize a high-speed communication through CNIC, what the protocol processor mainly is to

convert physical addresses to network addresses and to control page tables used then. the protocol processor interface need to controls the protocol processor properly.

The basic function of the protocol processor interface is to input or output data properly for the request of memory access by the protocol processor by controlling READ or WRITE to ROM or RAM. The protocol processor interface consists of ROM, RAM, some controllers and the interface connecting them to the protocol processor.

When the protocol processor accesses memory, the interface decides which ROM or RAM is selected, and it instructs the ROM or RAM to output data to data bus when READ, and to input data from data bus when WRITE. And then by asserting READY at a proper timing, it informs the protocol processor of completion of the output or input.

At first, I designed ROM, RAM, and some controllers in the FPGA. I ran a simple program on it. I put into the ROM the instructions to set address spaces of the protocol processor at boot. CNIC has pins to snoop signals in the FPGA. By observing them I verified that it run as I have expected.

Then I designed the interface of a Synchronous SRAM on CNIC for the protocol processor to use it instead of RAM. The Synchronous SRAM offers broad workspace for the protocol processor. As above, I ran a test program on it and I verified that it run as I have expected, by observing the pins on CNIC. Since the protocol processor is able to access the Synchronous SRAM, we can run some large program on the processor and it enables us to evaluate various methods.

高速通信ボード CNIC における
プロトコル・プロセッサ・インタフェースの設計

目次

第1章　はじめに

最近では研究室やオフィスなどにおいて多数のコンピュータをネットワークで結合し、コンピュータ資源の有効利用を目指した分散システムが一般的になってきた。しかし各ノードとなるパーソナルコンピュータやワークステーションはもともと単体での利用を想定して設計されたものが多いため、通信の機能がハードウェア、ソフトウェアの両面において付加的、汎用的であり、通信にかかるコストは高くなってしまう。

実際こういった環境において通信を行う場合には、資源保護のためにシステムコールによるオペレーティングシステムの介在が発生する。また、汎用の通信プロトコルを使用するために煩雑な処理が必要になりオーバヘッドが生じるのである。分散システムにおいて細粒度の並列処理を行おうとする場合、このような通信コストは致命的な問題である。

そこで我々は分散システムにおけるこのような通信コストの問題を、ハードウェア、ソフトウェアの両面から解決し、コンピュータ資源を最大限に利用してシステムの最適化を図る「コンピュータ・コロニー」を開発している。

本稿では、「コンピュータ・コロニー」における高速通信の実現のためのハードウェア構成について示し、「コンピュータ・コロニー」実現を目指して実際に研究室で実験、開発しているプロトタイプ・ハードウェアについて述べる。そしてプロトタイプ・ハードウェアにおいて高速通信を実現する専用高速通信ボード CNIC の開発と、CNIC において高速通信に必要な様々な処理を担当するプロトコル・プロセッサのインタフェース設計および実装について述べる。

以下2章では、我々が開発を行っている「コンピュータ・コロニー」の概要を示し、その実験環境であるプロトタイプ・ハードウェアとその専用通信ボード CNIC について述べる。3章では実際に CNIC に実装したプロトコル・プロセッサ・インタフェースの設計について述べ、4章ではまとめを行う。

第2章　コンピュータ・コロニー

「コロニー」とは生物学用語でいう「群体」のことであり、群体とは同種の生物個体が多数集合して共通の体を形成し、相互に協調し合いまるで1つの生物かのように生活しているものを言う。すなわち「コンピュータ・コロニー」は、複数のコンピュータがネットワークを通じて相互に連絡し合い、利用者にとっ

ては全体でまるで1つのコンピュータであるかのように振舞うシステムを目指している [1]。

2.1 コンピュータ・コロニーの背景

科学技術の進歩に伴って、あらゆる分野において扱われるデータ量が増加し、またそれを処理するアルゴリズムも複雑化の一途をたどっている。その結果、もちろんコンピュータ単体の性能は向上しているが、それ以上にコンピュータに与える負荷は増える一方である。企業や大学の研究室など特に高度な計算が必要とされる環境では、単一のプロセッサからなるコンピュータでの処理は限界に近づいてきた。これを受けて複数のプロセッサを搭載した並列計算機を用いて全体の処理を並列化させ高速化を図るという方法が取られるようになってきた。

2.1.1 分散システム

従来並列計算機というとマルチプロセッサを搭載したいわゆる集中システムが主流であった。ところが最近のパーソナルコンピュータやワークステーションの高性能化や低価格化、さらにはネットワークの高速化ともあいまって、複数のコンピュータをネットワークで結合させた分散システムが注目されている。分散システムを形成する各ノードは大量生産によってコストが下げられており、同数のプロセッサを持つ集中システムに比べて、システムのコストもはるかに抑えられる [2]。また分散システムはその形態の特徴から、集中システムに比べて新しいノードの追加による性能向上が容易に行えるというメリットもある。

2.1.2 分散システムの問題点

しかし現在の分散システムは通信機能が付加的であるため通信コストが高く、処理の負荷分散が現実的でない。プリンタやディスクを共有する程度であればそれほど通信の高速性は要求されないが、並列処理を行うに当たっては通信のレイテンシは処理のボトルネックとなる。また、マルチプログラミング、マルチユーザ環境においては、ユーザごとに、あるいはユーザが実行するアプリケーションごとに要求する QoS (Quality of Service) は異なる。このような環境ではユーザの要求に対する適切なスケジューリングが行われる必要があるが、現在の分散システムではこういった機能は十分に実現されていない。

2.2　コンピュータ・コロニーの概要

以上のことを受けて、コンピュータ・コロニーでは現在の分散システムの長所である低価格性や拡張容易性を維持しながら、問題点であったコンピュータ資源の有効活用を目指している。

そこで我々は次に挙げる項目をコンピュータ・コロニーの設計目標として掲げている。

低価格性　従来の分散システムと同じく低価格で導入、維持することができ、また容易に拡張することができるシステムを目指す。

コンピュータ資源の有効利用　現在の分散システムにおける問題点である通信コストを削減し、プロセッサやメモリといったコンピュータ資源を最大限に有効利用できるシステムを目指す。

マルチプログラミング、マルチユーザ環境　重いジョブを投入した特定のユーザによってシステムが独占されないよう適切なジョブスケジューリングを行い、各ユーザにとって快適なシステムを目指す。

コンピュータ・コロニーは通信コストを削減し高速通信を実現するハードウェアと、動的な負荷分散によりシステムのコンピュータ資源の有効利用を目指すオペレーティングシステム Colonia からなる。このようにコンピュータ・コロニーではハードウェアとソフトウェアの両面からシステムの最適化を目指して研究を進めている。

2.3　コンピュータ・コロニーのハードウェア構成

コンピュータ・コロニーでは、プロセッサ、メモリ、NI（Network Interface）を基本単位とするモジュールカードによるハードウェア構成を採用する。コンピュータ・コロニーの構成ノードは、このようなカードを 1 つあるいは複数装着した本体と、その他モニタ、キーボードなど周辺機器からなる。このノードをネットワークで結合し、全体で分散システムを形成する。構成のイメージを図 1 に示す。

将来このようなカードが低価格で大量生産されるようになると、各ノードを容易に導入することができる。また拡張する際には本体にカードを追加するだけでよく、システムを低価格で導入、拡張することができる。このようなハー

ドウェア構成をとることによって、コンピュータ・コロニーは先に述べた要件を実現することができる。

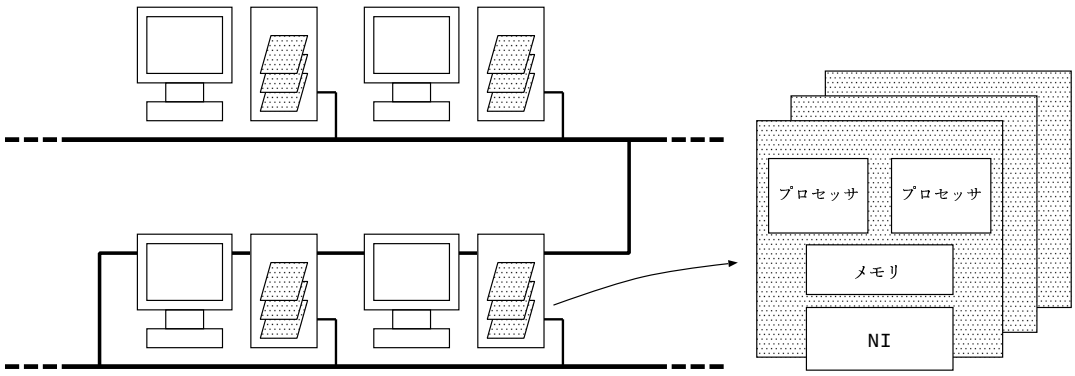

図1: コンピュータ・コロニーの構成

2.4 コンピュータ・コロニーにおける通信機構

2.4.1 高速通信の要件

コンピュータ・コロニーにおいて高速通信を実現するためには満たすべき要件を以下に示す。

1. **システムコールを介さない通信**
   通信資源を保護するためのシステムコールによるオペレーティングシステムの介在を削減する。
2. **プロトコル処理によるオーバヘッドの削減**
   汎用のプロトコルを使用すると、膨大なプロトコル処理によるオーバヘッドが生じる。これを削減する。
3. **リモート メモリのキャッシング**
   リモートノードにあるメモリの内容を自ノードのメモリにキャッシングし、2回目以降の参照をリモートメモリに対してではなく自ノードのメモリへのアクセスにすることで高速化を図る。

2.4.2 コンピュータ・コロニーにおける共有分散メモリ

以上のような高速通信の要件を満たすため、コンピュータ・コロニーでは先に提案したモジュールにおいてNIをシステムバスに接続し、共有メモリベース

の通信機構を採用する [3]。ノード内においてはコンピュータ・コロニーでは仮想メモリを採用しており、ユーザはシステムの提供する仮想空間にアクセスし、MMU（Memory Management Unit: メモリ管理ユニット）がそれを物理アドレスへ変換しシステムバスへ流す。このときシステムバスに接続されたNIが直接リモートメモリアクセスの必要性の有無を感知し、必要であればシステムコールを介さずにリモートアクセスを行うことができる。またNIがシステムバスに接続されていることで、キャッシュコヒーレンス制御をNIによってハードウェア的に高速に行うことができる。

この共有メモリベースの通信をコンピュータ・コロニーでは、メインメモリ上の物理ページを以下のように区別し実装する。(図2)

- ホームページ
  物理メモリが実装されている物理アドレスに位置し、自ノードのメモリ上にデータの実体が存在する。ホームページが存在するノードをホームノードという。
- ダミーページ
  物理メモリが実装されていない物理アドレスに位置し、リモートメモリ上にデータの実体が存在する。直接アクセスすることはできない。ダミーページへのアクセスがあるとNIがそれを感知し、ホームページへリモートアクセスを行う。
- コピーページ
  物理メモリが実装されている物理アドレスに位置し、ホームページの内容をキャッシングしている。このページが無効化されているときにアクセスが行われると、ダミーページへのアクセスと同様にホームページへリモートアクセスを行う。

まずリモートノードに存在する共有するページはダミーページにマッピングされる。そして頻繁にアクセスされるような場合、コピーページにマッピングし、ノード内のメモリにキャッシングする。

また、リモートメモリアクセス時にシステムバスに流れる物理アドレスをそのまま利用するのは、スワップ等が発生した場合に対する柔軟性に欠ける。よってシステムで唯一である仮想的なネットワークアドレスを用いる。リモートメモリへのアクセス時にはNIが自身で管理している変換ページテーブルを参照

し、物理アドレスをネットワークアドレスに変換し通信を行う。(図3)

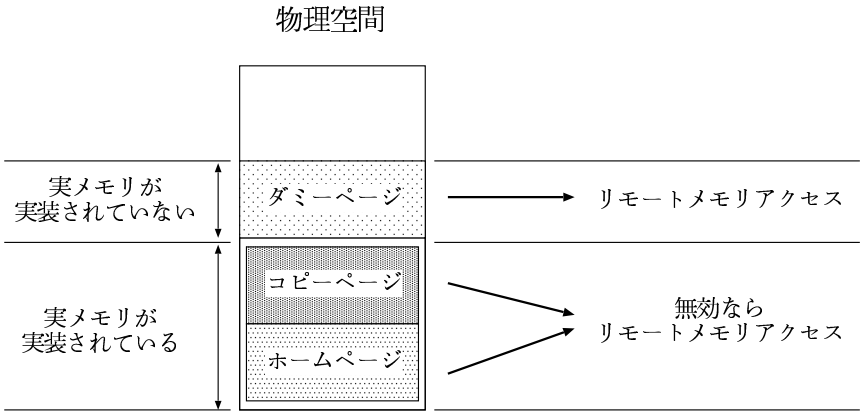

図2: 物理ページの分割

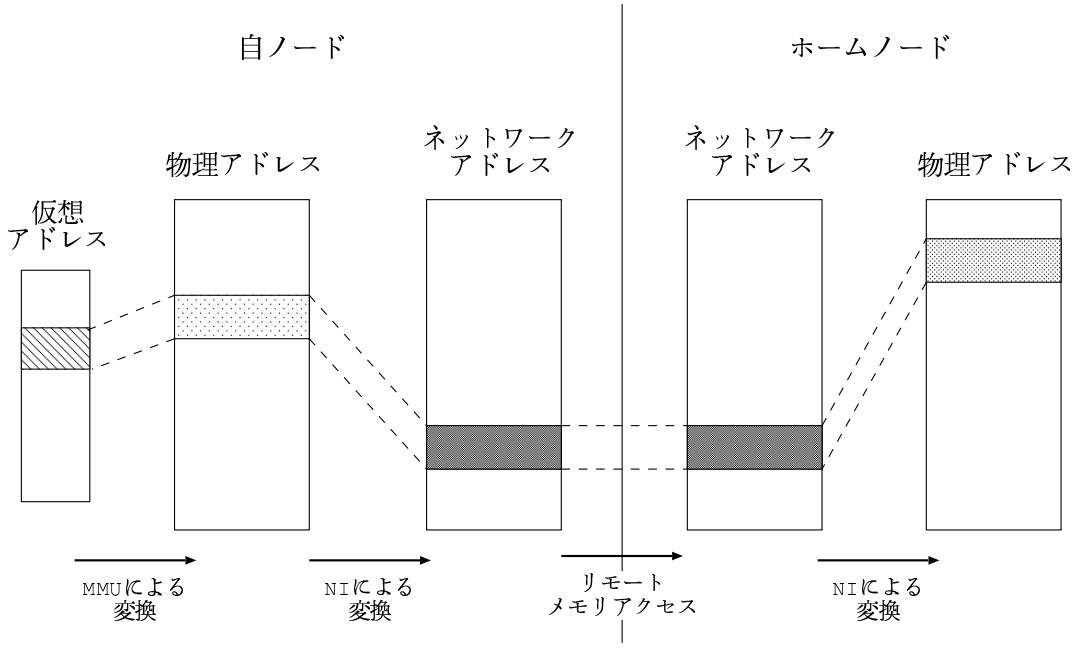

図3: リモートメモリアクセス

2.5 プロトタイプ・ハードウェア

以下では、コンピュータ・コロニーを実現するために、研究室で開発しているプロトタイプ・ハードウェアを示す。

2.5.1 プロトタイプ・ハードウェアの概要

プロトタイプ・ハードウェアは、研究室で先に述べた高速通信機構の実験、評価を行うために次の機能を実装する。

- システムコールによるオペレーティングシステムの介在を低減することができる
- プロトコルオーバヘッドを削減するため、汎用で煩雑なプロトコルではなく専用のプロトコルを用いることができる
- 様々な通信方式を実験、評価を行うことができる

2.5.2 プロトタイプ・ハードウェアの構成

現在開発しているプロトタイプ・ハードウェアはノードとしてSPARCstation20を採用している。SPARCstation20はシステムバスとしてMBusスロットを持っており、そのスロットに専用の高速通信ボード CNICを装備することによって、システムバスを利用しコストを削減した共有メモリベースの通信を行うことができる。また、CNICはプロトコル・プロセッサを搭載しており、様々な通信方式を容易に試すことができる。

プロトタイプ・ハードウェアは、SPARCstation20のプロセッサであるSuperSPARCとメインメモリ、そしてそのシステムバスにNIとなるCNICを接続することで、コンピュータ・コロニーの採用する、プロセッサ、メモリ、NIからなるモジュール構成をとる。このモジュールを1つのノードと見なし、Fibre Channelを介してノードを結合し互いに通信を行い実験、評価を行う。

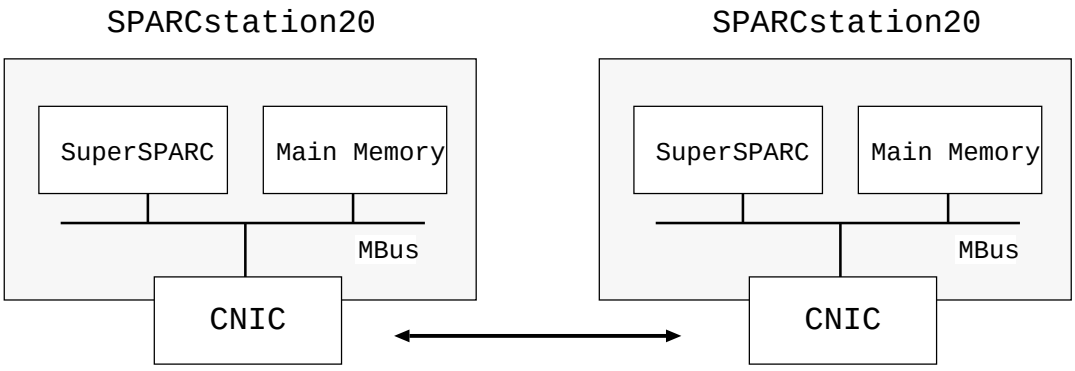

図4: プロトタイプ・ハードウェアの構成

2.6 CNIC

次にコンピュータ・コロニーのプロトタイプ・ハードウェアにおけるNIとして我々が開発している高速通信ボード CNIC (Computer-Colony Network Interface Card) について述べる。

2.6.1 CNIC の概要

CNIC の基本機能はローカルノードの物理アドレスとネットワークアドレスを対応付けることである [4]。システムバスを常に監視しリモートメモリへのアクセスが必要な場合、物理アドレスをネットワークアドレスに変換し当該ノードへの通信を行う。逆にリモートノードからのアクセスに対してネットワークアドレスを物理アドレスに変換する。

2.6.2 CNIC の構成

CNIC は、プロトコル・プロセッサ、メモリ、Fibre Channel用Chipset、FPGA (Field Programmable Gate Array) からなる。CNIC はプロトコル・プロセッサを搭載しており、また各種コントローラやバッファを FPGA 内に設計を行うことで、様々な通信方式の実験、評価を容易に行うことができる。以下に主な部分の説明を行う。

プロトコル・プロセッサ 組み込み用プロセッサ、SPARClite を採用する。主にリモートメモリアクセス時のアドレス変換、またアドレス変換に必要なページテーブルの管理を行う。

メモリ メモリには以下のような種類がある。

- **B_Map**
  自ノードのメインメモリの各キャッシュブロックについて、共有／無効等の情報を保持し、リモートメモリのキャッシュコヒーレンスに使用する。
- **E_Cache**
  プロトコル・プロセッサ用の2次キャッシュ。自ノードのメインメモリの内容をキャッシュしておく。またプロトコル・プロセッサのメインメモリとして使用する.

Fibre Channel 用 Chipset Fibre Channel を介した通信を行う際、プロトコルの処理の一部を担当する。

FPGA　FPGA 内には以下のようなものを設計する。

- コントローラ
  プロトコルプロセッサのコントローラ、メモリのコントローラ、及び MBus インタフェースや Fibre Channel 用 Chipset、各種バッファのコントローラ。
- 送受信バッファ
  MBus 側、Fibre Channel 側それぞれのインタフェース・バッファ。

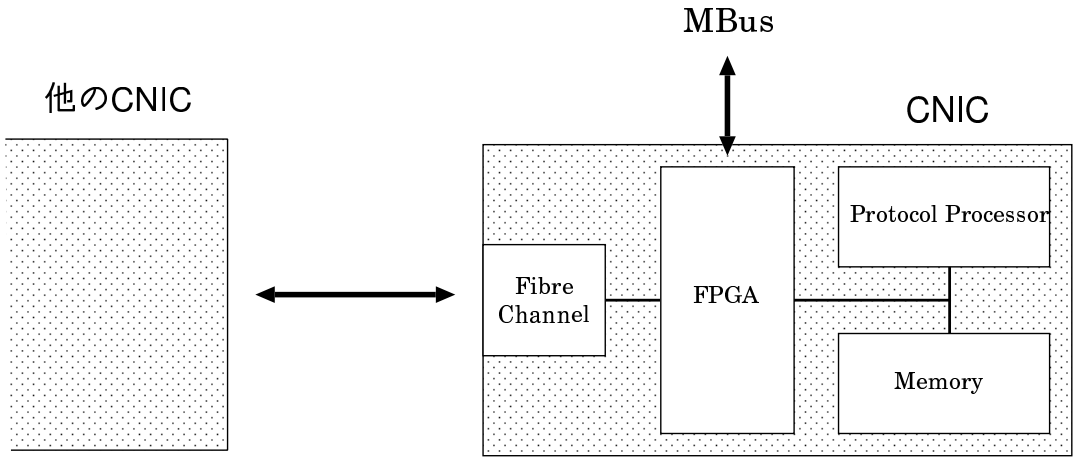

図5: CNIC の構成

2.6.3　CNIC の段階的設計

先に述べた通り、CNIC の基本機能はプロトタイプ・ハードウェアにおける SPARCstation20 のシステムバスに流れる物理アドレスを監視し、リモートアクセスの必要性があれば、物理アドレスとネットワークアドレスを変換しリモートアクセスを行うものである。CNIC の開発はオペレーティングシステム Colonia の開発の進行状況に大きく左右される。ゆえに CNIC 本来の機能を開発する前にまず CNIC 単独で開発できれば、効率的な開発が期待できる。

CNIC 上では、Fibre Channel 用 Chipset と Fibre Channel インタフェース・バッファを合わせてネットワーク・インタフェースの機能を果たしている。ここでプロトコル・プロセッサを CNIC のメインのプロセッサ、E_Cache として使用するシンクロナス SRAM をメインメモリと見ると、CNIC はプロセッサ、メモリ、ネットワーク・インタフェースから構成されるモジュールと見なすことができる。

このネットワーク・インタフェースはCNIC上のプロトコル・プロセッサとメモリをつなぐシステムバスに結合されており、このモジュールはコンピュータ・コロニーの提唱する、プロセッサ、メモリ、NIをシステムバスでつないだモジュール構成と同様の形態をしている。CNICにおいてプロトコル・プロセッサをメインプロセッサとして用いることで様々なプログラムを実行させることができ、さらにネットワーク・インタフェースを介して、他のCNICと通信を行うことも可能である。

ここでCNIC開発の第1段階としてCNICをSPARCstation20と切り離し単独のモジュールと見て、このモジュール同士だけで通信を行うことを考える。このモジュール同士で通信を行い評価することができれば、次の段階においてこのモジュールにアドレス変換などの機能を付け足すことでスムーズに目標である、プロトタイプ・ハードウェアのNIとしてのCNICの開発を進めることができる。

このようにCNICの開発の第1段階として、CNICを単独のモジュールとして見なしたものをプロトタイプCNICと呼び開発を行う。

2.6.4 プロトタイプCNICの構成

次にプロトタイプCNICの主な構成を示す。

プロセッサ CNICにおけるプロトコル・プロセッサを使用する。プロトタイプCNICのメインプロセッサである。

メモリ プロトコル・プロセッサの2次キャッシュ、E_Cacheをプロトコル・プロセッサのメインメモリとして使う。

FPGA FPGA内にプロトコル・プロセッサ、メモリの制御、ブートに必要なROM領域、通信を行う際の各種バッファを設計する。

第3章 プロトコル・プロセッサ・インタフェース

プロトコル・プロセッサ・インタフェースとはCNIC上のプロトコル・プロセッサの制御を行うものである。コンピュータ・コロニーのプロトタイプ・ハードウェアにおいてCNIC上のプロトコル・プロセッサは、通信に関してアドレス変換をはじめとする様々な処理を担当する。そのためには、プロトコル・プ

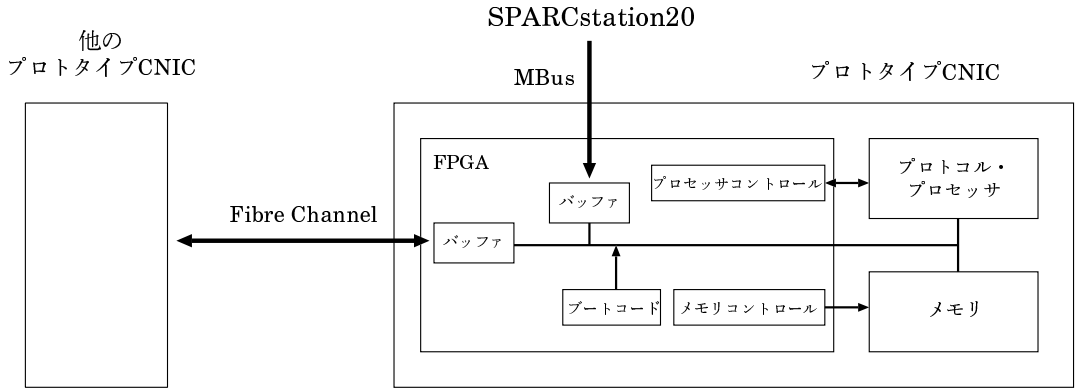

図6: プロトタイプ CNIC の構成

ロセッサを適切に制御し、様々な処理を行うことのできる環境を設計しなければならない。プロトコル・プロセッサ・インタフェースの制御によって CNIC を介した高速通信が可能になる。

3.1 プロトコル・プロセッサ・インタフェースの設計方針

まず最初にプロトコル・プロセッサが我々の期待通りに動作する環境を設計する必要がある。そこで最初の段階として、プロトコル・プロセッサを制御しながら適切な設定でブートしてやらなくてはならない。まずは ROM, RAM などを全て FPGA 内に設計し、簡単なプログラムを与えて実行させ、期待通りの動作が行われていることを確認する。

次にプロトコル・プロセッサのメインメモリとして FPGA 内に設計した RAM の替わりに CNIC で E_Cache として利用するシンクロナス SRAM が利用できるように設計する。シンクロナス SRAM が使えるようになると、プロトタイプ CNIC においてもプロトコル・プロセッサに大規模なプログラムを与えて実行させることができるようになり、様々な評価も行える。

以下では実際に FPGA 内にプロトコル・プロセッサ・インタフェースを設計し、与えたプログラムで期待通りの動作が実現できることを確かめる。

3.2 プロトコル・プロセッサのアドレス空間の設定

プロトコル・プロセッサに使用した SPARClite の持つアドレス空間は大きくユーザ空間とスーパバイザ空間に分かれている。主にこの中のユーザ命令領域、スーパバイザ命令領域、ユーザデータ領域、スーパバイザデータ領域をプロト

コル・プロセッサの作業領域として使用する。それぞれの領域はプロトコル・プロセッサではASI (Address Space Identifier) で識別される。

ここで、プロトコル・プロセッサに使用したSPARCliteは仮想メモリをサポートしていない。このためプロトコル・プロセッサ内部のアドレスが、MMUを通さずそのまま外部の物理アドレスとして出力される。このとき、プロトコル・プロセッサ内部でどの領域にアクセスされたかはASIとアドレスの上位をデコードして判断しなければならない。

組み込み用プロセッサであるこのプロトコル・プロセッサはCS (Chip Select) をサポートしている。CSとはプロトコル・プロセッサ内部でASIと上位アドレスがデコードされたものであり、これを用いると外部にデコード回路を設計する必要がなくなる。プロトコル・プロセッサ内のアドレス空間をCS0からCS5まで6つのCSに割り当てることができる。

今回の設計ではCSを次のように割り当てる。

CS0 プロトコル・プロセッサのデフォルトでブート領域としてスーパバイザ命令領域（0から32KBまで）に割り当てられている。プロトコル・プロセッサのブート時にはここの0番地から命令が読み始められる。

CS1 ブート領域とは別にプログラムを置く領域として使う。ブート領域とは重ならないように、スーパバイザ命令領域（1GBから2GBまで）に割り当てた。

CS2 READ, WRITEがともにでき、さらにそこにWRITEされた命令を実行できるようにするために、スーパバイザ命令領域とスーパバイザデータ領域（2GBから4GBまで）に割り当てた。

CS3 各種設定用のプロトコル・プロセッサ内部レジスタを読み書きするための特殊な領域に用いられる。スーパバイザデータ領域（32KBから33KBまで）に割り当てた。

CS4, CS5 DRAM領域として使うので、当面は使わないユーザ領域にそれぞれ割り当てた。

3.3 プロトコル・プロセッサの主な制御信号

プロトコル・プロセッサを制御するにあたって主に必要になる制御信号を下に述べる。以下の制御信号の内、頭に"X_"とあるものは負論理で扱われる。

X_READY レディー(READY)。プロトコル・プロセッサがROM, RAMに

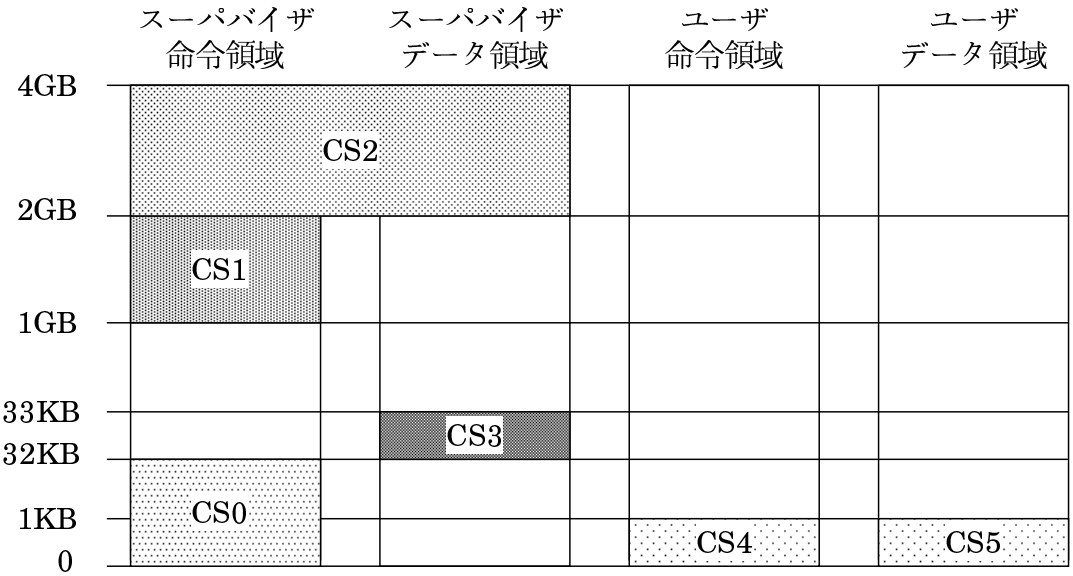

図 7: アドレス空間の割り当て

対して READ を指示してから実際にバスにデータが出力されるまでに、あるいは WRITE を指示してからバスからデータが取り込まれるまでにバスクロックで数クロックかかる。このため、ROM, RAM に対する読み書きが適切に行われたことをプロトコル・プロセッサに知らせるために、適当なタイミングで READY 信号をアサートする必要がある。

X_AS アドレスストローブ (ADDRESS STROBE)。プロトコル・プロセッサが ROM, RAM に対するアクセスの開始を指示するための信号である。

RDWR リードライト (READ WRITE)。ROM, RAM へのアクセスの際の READ, WRITE を指示する。'1' のとき READ、'0' のとき WRITE に対応する。

X_CS チップセレクト (CHIP SELECT)。X_CS0 から X_CS5 まであり、プロトコル・プロセッサのアクセスするアドレス空間に対応した X_CS が'0' になる。

X_BE バイトイネーブル (BYTE ENABLE)。プロトコル・プロセッサは 32 ビットバスモードに対応しており、通常データは 32 ビット幅で扱われる。一方、8 ビットあるいは 16 ビット単位での READ, WRITE にも対応しており、X_BE はこのとき下位ビットを指定するために使用される。今回の設計では、プロトコル・プロセッサはデータ幅を 8 ビットで扱う 8 ビット

バスモードでブートする。この8ビットバスモードにおいてアドレスの下位2ビットを指定するときに使用する。

3.4 プロトコル・プロセッサのブート回路の実装

まず最初にプロトコル・プロセッサが我々の期待通りに動作する環境を設計する必要がある。そこで最初の段階として、プロトコル・プロセッサを制御しながら適切な設定でブートしてやらなくてはならない。そしてまずは簡単なプログラムを与えて実行させ、期待通りの動作が行われていることを確認する。

3.4.1 主な構成

まず、全体の設計図の概略を図8に示す。ここで、データバスは32ビット幅、アドレスバスは30ビット幅あるので配線遅延によって全てのデータがバスに出揃う時間に差が生じる。この状況でデータやアドレスを取り込むのは誤動作の原因になるので、FPGAとのインタフェース部ではDフリップフロップによってデータ、アドレスを一律に揃える。

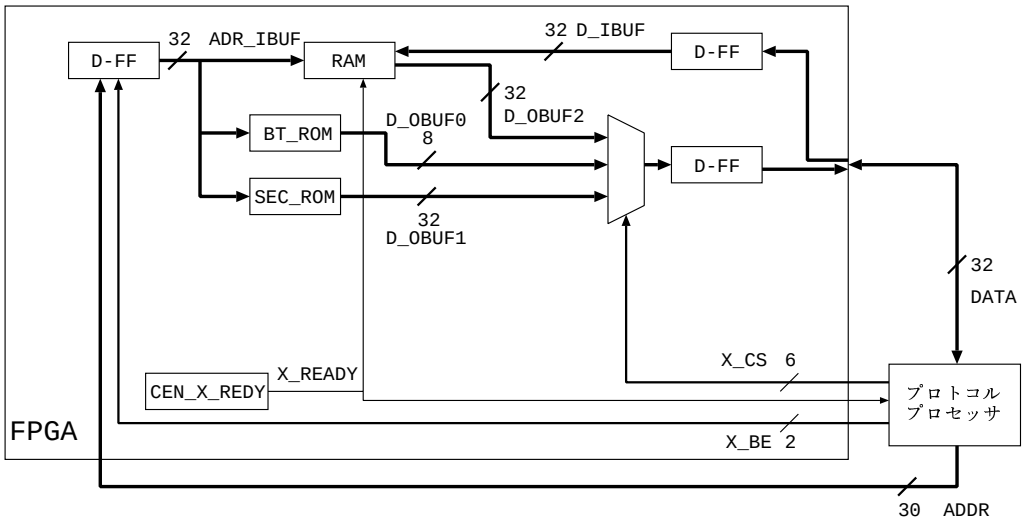

図8: ブート回路におけるプロトコル・プロセッサ・インタフェース

- プロトコル・プロセッサ

  25MHzのバスクロックで動作させる。このため以下の各部分もこれに同期させるため25MHzで動作させる。

- **ROM**

  以下に述べる BT_ROM と SEC_ROM の 2 つを FPGA 内に設計する。

  1. **BT_ROM**

     プロトコル・プロセッサのブート用の ROM 領域。CS0 に割り当てる。プロトコル・プロセッサのブート時にはアドレス空間などの初期設定のために、該当する内部レジスタの値を設定してやらなければならない。ゆえにそのための命令を保持しておく ROM 領域が必要である。ここで FPGA 内にブート用の ROM 領域を作成しブート領域に割り当て、プロトコル・プロセッサをここからブートさせる。プロトコル・プロセッサの初期設定のための命令をあらかじめ BT_ROM 内に埋め込んでおく。

     またプロトコル・プロセッサは、ブート時には 8 ビットバスモードで動作するようになっている。したがって BT_ROM のバス幅は 8 ビットで設計する。

  2. **SEC_ROM**

     32 ビットバスモード用の ROM 領域。CS1 に割り当てる。プロトコル・プロセッサは 32 ビットバスモードでの動作をサポートしており、こちらで動作させるほうが 8 ビットバスモードで動作させるより高速である。ゆえに SEC_ROM を 32 ビット幅で設計し、ブート領域を抜けたあとこちらの領域で動作させることにより高速化を図る。

- **RAM**

  FPGA 内に設計する。プロトコル・プロセッサ用の読み書き可能な作業用 RAM 領域。CS2 に割り当てる。32 ビットバスモードで動作させるため、32 ビット幅で設計を行う。

- **X_READY 生成部**

  FPGA 内に設計する。プロトコル・プロセッサの CS1, CS2 に対するアクセスを感知し、X_READY 信号を生成する。CS0 に関してはプロトコル・プロセッサの持つ内部レディーを使用する。

3.4.2 FPGA の設計

FPGA 内の設計は HDL (Hardware Discription Language:ハードウェア記述言語) の 1 つである VHDL を用いて行った。回路を VHDL で記述することで

ゲートレベルでのケアレスミスを減らし、設計を容易に行うことができる [5]。次に主な部分の設計について述べる。

1. **ブート ROM 部 (BT_ROM)**　BT_ROM の領域を 256 × 8 ビットの constant 配列として確保する。初期値としてプロトコル・プロセッサ用のコードを埋め込んでおく。
2. **32 ビット ROM 部 (SEC_ROM)**　SEC_ROM の領域を 32 × 32 ビットの constant 配列として確保する。初期値として BT_ROM 同様にプロトコル・プロセッサ用のコードを埋め込んでおく。
3. **RAM 部 (RAM)**　RAM の領域を 16 × 32 ビットの配列変数として確保する。
4. **RAM 読み書き部 (ACCESS_RAM)**

   RDWR, X_READY, X_CS2 を入力として受け取る。アドレスを入力として RAM の対応するアドレスにあるデータを出力する。

   RDWR, X_CS2 がともに'0' のときは RAM への書き込みなので、アドレスとデータを入力として保持しておき、X_READY が'0' になったときにクロックの立ち上がりに同期して、RAM の対応するアドレスへデータを書き込む。
5. **X_READY 生成部**

   GEN_X_RDY　32 ビットバスモードにおいて X_READY を生成する部分である。X_AS を入力、X_READY を出力とするステートマシンで構成する。表 1 の状態遷移と出力を行う。状態遷移は全てクロックの立ち上がりで行われる。データバスへの出力、またはデータバスからの入力にかかる時間に余裕を見て、3 サイクルで X_READY を出力するようにする。
6. **インタフェース部 (INTERFACE)**

   前述の各部分を統合制御し、プロトコル・プロセッサとのインタフェースを担当する。

   - バッファ

     ADR_IBUF　アドレスの入力バッファ。32 ビット幅。上位 30 ビットはアドレスバスを接続。下位 2 ビットは X_BE を接続。それぞれクロックの立ち上がりで入力される。

     D_IBUF　データの入力バッファ。32 ビット幅。クロックの立ち上が

現状態	次状態	出力 **X_READY**
S0	S1 (1)	1
	S0 (2)	
S1	S2	1
S2	S3	1
S3	S0	0

(1) X_AS = '0' のとき
(2) X_AS = '1' のとき

表1: GEN_X_RDY

りでデータバスから入力される。

X_RDY_BUF　それぞれX_READY生成部で作られたX_RDYと出力をつなぐためのバッファ。

D_OBUF　データの出力バッファ。BT_ROM, SEC_ROM, RAMからの出力データのバッファはそれぞれ、D_OBUF0, D_OBUF1, D_OBUF2が対応する。

- インタフェース

  アドレス　READ_BT_ROMへのアドレスはADR_IBUFの下位0ビット目から接続する。一方、READ_SEC_ROM, RAMへのアドレスは32ビット幅なので下位2ビットが必要ない。したがって下位3ビット目から接続する。

  レディー　X_RDY_BUFをクロックの立ち上がりでX_READYとして出力する。

  データ　D_OBUFからデータバスへの出力は、RDWRが'1'のとき対応するX_CSが'0'であるD_OBUFをデータに出力する。RDWRが'0'のときはデータバスへの出力はハイインピーダンスにする。

3.4.3　ROM用コードの作成

ROM用コード作成の方針としては

1. プロトコル・プロセッサと互換性のあるコンパイラでバイナリコードを生成する。

2. 生成されたバイナリコードから必要なコード部分を抜き出し、VHDLで設計されたROMに埋め込んでいく。

埋め込むコードは次のようにして作成した。

(i) **プログラムの作成**

まずプロトコル•プロセッサに使用したSPARCliteと互換性のあるSPARC用のアセンブリ言語でプログラムを作成する。ファイル名はBT_ROM.sとする。(図9)

(ii) **アセンブル**

次に研究室にあるSUNのSolarisにおいてプロトコル•プロセッサと互換性のあるコンパイラで、作成した命令をアセンブラにかけバイナリコードを生成する。

```
% as -o BT_ROM BT_ROM.s
```

(iii) **逆アセンブル**

生成されたバイナリコードはコード領域以外にもデータ領域など不要な部分を含むので、このバイナリコードを逆アセンブルしコード領域だけをファイル(BT_ROM.dis: 図10)に抽出する。

```
% dis BT_ROM > BT_ROM.dis
```

(iv) **ROM への埋め込み**

(a) **BT_ROM**

(iii)で抽出されたコードをVHDLファイル(BT_ROM.vhd: 図11)にROMとして8ビット幅ずつ埋め込んでいく。

(b) **SEC_ROM**

(iii)で抽出されたコードをVHDLファイル(SEC_ROM.vhd: 図12)にROMとして32ビット幅ずつ埋め込んでいく。

3.4.4 ブート回路の検証

以上の設計によりプロトコル•プロセッサをブートさせる。実行される命令は以下のように設計した。

```
set     INIT_PSR, %o0
mov     %o0, %psr
set     INIT_WIM, %o0
mov     %o0, %wim
set     INIT_TBR, %o0
mov     %o0, %tbr
. . .
```

図 9: BT_ROM.s

```
 0:  11 02 40 00    sethi    %hi (0x90000000), %o0
 4:  90 12 20 c7    or       %o0, 0xc7, %o0
 8:  81 8a 00 00    wr       %o0, %g0, %psr
 c:  90 10 20 01    mov      1, %o0
10:  81 92 00 00    wr       %o0, %g0, %wim
14:  11 18 00 00    sethi    %hi (0x60000000), %o0
10:  81 9a 00 00    wr       %o0, %g0, %tbr
. . .
```

図 10: BT_ROM.dis

```
constant BT_ROM_DATA: BT_ROM := BT_ROM_RABLE'(
    BT_ROM_WORD'(x"11"), BT_ROM_WORD'(x"02")
    BT_ROM_WORD'(x"40"), BT_ROM_WORD'(x"00")
    BT_ROM_WORD'(x"81"), BT_ROM_WORD'(x"8a")
    BT_ROM_WORD'(x"00"), BT_ROM_WORD'(x"00")
    BT_ROM_WORD'(x"90"), BT_ROM_WORD'(x"10")
    BT_ROM_WORD'(x"20"), BT_ROM_WORD'(x"10")
   . . .
```

図 11: BT_ROM.vhd

```
constant SEC_ROM_DATA: SEC_ROM := SEC_ROM_RABLE'(
    SEC_ROM_WORD'(x"11024000"), SEC_ROM_WORD'(x"901220c7")
    SEC_ROM_WORD'(x"818a0000"), SEC_ROM_WORD'(x"90102001")
    SEC_ROM_WORD'(x"81920000"), SEC_ROM_WORD'(x"11180000")
   . . .
```

図 12: SEC_ROM.vhd

CS0　BT_ROM が読み込まれる。命令の主な内容は各種レジスタの設定である。CS0 に関してデフォルトでは 32 ウェイトとなっている内部レディーを 3 ウェイトに、および外部レディーは使用しないように設定する。先に述べたアドレス空間と CS との対応を設定する。そして CS イネーブルをオンにして CS を使用可能に設定する。これらの処理を終えるとブート領域から CS1 の最初へとジャンプする。

CS1　SEC_ROM が読み込まれる。32 ビットバスモードで動作する。命令の内容は、RAM に対して「CS0 へジャンプせよ」という命令を書き込む。この処理を終えると CS2 の最初へとジャンプする。

CS2　RAM が読み込まれる。32 ビットバスモードで動作する。「CS0 の最初へジャンプせよ」という命令が実行され、CS0 の最初へジャンプする。

CNIC 上には FPGA 内の信号をスヌープできる端子が装備されており、そこを観察することでプロトコル・プロセッサの動作が確認できる。具体的には X_CS0, X_CS1, X_CS2 をロジックアナライザと呼ばれる測定器で観察した。図 13 の画面において、X_CS(0), X_CS(1), X_CS(2) を見ると、CS0 → CS1 → CS2 → CS0 → ... という繰り返しが認識できるので、プロトコル・プロセッサが先ほど述べた通りに動作していることが確かめられた。

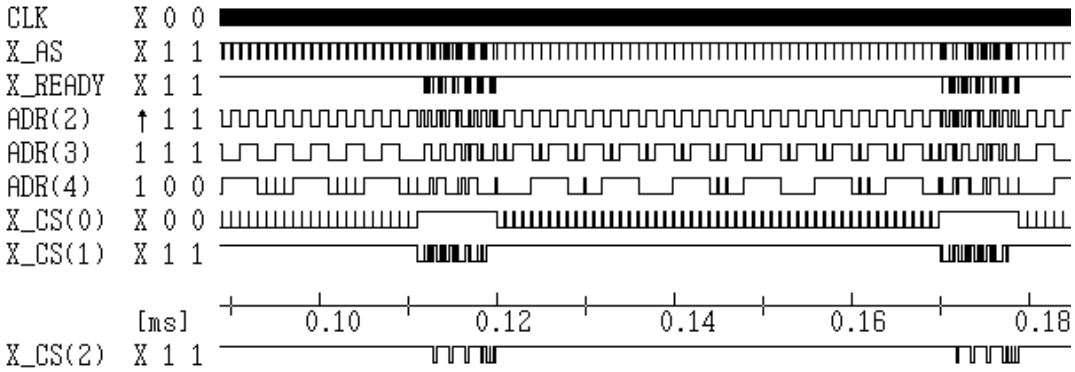

図13: ロジックアナライザ

3.5　シンクロナスSRAM・インタフェースの設計

CNICを介したリモートメモリアクセス時には、プロトコル・プロセッサがアドレス変換をはじめとする様々なプログラムを実行するために、大容量かつ高速なメモリが必要である。また自ノードのメインメモリの内容をCNIC上にキャッシュしておく領域も必要となる。

したがって、ブート回路でFPGA内に設計したRAMの代わりにSSRAMで構成されたE_Cacheをプロトコル・プロセッサからアクセスできるようにする必要がある。そこでプロトコル・プロセッサとシンクロナスSRAMとのインタフェースを設計する。

3.5.1　主な構成

まず、全体の設計図の概略を図14に示す。

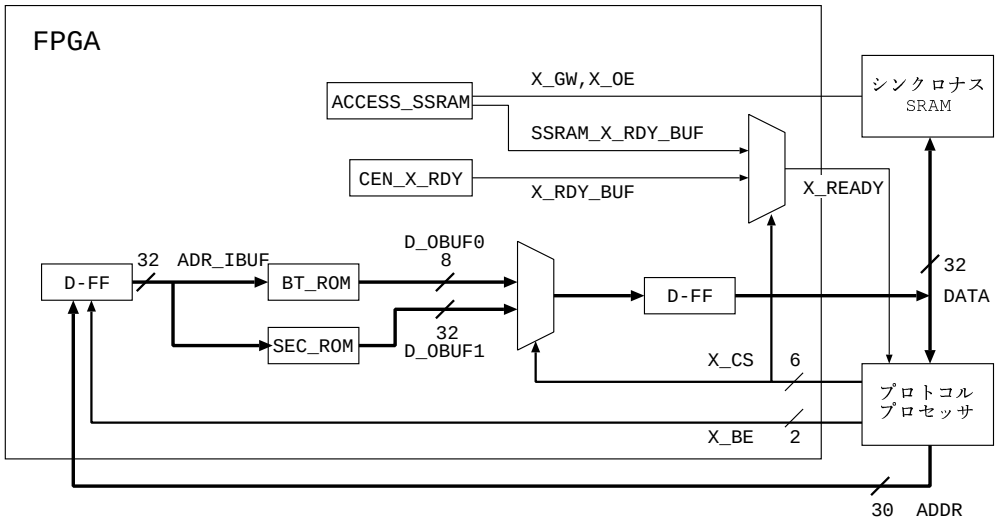

図14: プロトコル・プロセッサとのシンクロナスSRAM・インタフェース

- **プロトコル・プロセッサ**

  25MHzのバスクロックで動作させる。このためシンクロナスSRAMおよびそのアクセス部を除く各部分もこれに同期させるため25MHzで動作させる。

- **シンクロナスSRAM**

  プロトコル・プロセッサのメインメモリ。CS2に割り当てる。50MHzで

高速に動作する。

- **ROM**

  ブート回路同様、BT_ROM と SEC_ROM の 2 つを FPGA 内に設計する。

  1. **BT_ROM**

     CS0 に割り当てる。

  2. **SEC_ROM**

     CS1 に割り当てる。

- **X_READY 生成部**

  CS1 へのアクセスを感知し、X_READY 信号を生成する。

- **シンクロナス SRAM アクセス部**

  FPGA 内に設計する。CS2 へのアクセスを感知し、シンクロナス SRAM に対してリード、ライトを指示するとともに、その際にプロトコル・プロセッサに対してアサートする X_READY 信号を生成する。シンクロナス SRAM に同期するため 50MHz で動作する。

3.5.2 FPGA の設計

次に設計した各部分の内、ブート回路から変更のあった部分について説明する。

1. **READY 生成部**

   GEN_X_RDY CS1 へのアクセスに関して X_READY を生成する部分である。ブート回路と同様の遷移を行う。ただし S0 から S1 への遷移の起動条件は "X_AS = '0' and X_CS1 = '0'"である。

2. **シンクロナス SRAM アクセス部**

   ACCESS_SSRAM CS2 へのアクセスに際し、シンクロナス SRAM に対して X_GW, X_OE の制御を行う。同時にプロトコル・プロセッサに対する X_READY の生成も行う。X_GW はシンクロナス SRAM への WRITE、X_OE は READ の指示に使われる。

   RDWR が'1' のときは READ であり、表 2 の状態遷移と出力を行う。RDWR が'0' のときは WRITE であり、表 3 の状態遷移と出力を行う。正確には状態 S1 において RDWR の判定を行い、状態 S2 への遷移で分岐する。状態遷移は全てクロックの立ち上がりで行われる。

現状態	次状態	出力		
		X_GW	X_OE	X_READY
S0	S1 (1)	1	1	1
	S0 (2)			
S1	S2	1	1	1
S2	S3	1	0	1
S3	S4	1	0	0
S4	S0	1	0	0

(1) X_AS = '0' and X_CS2 = '0' のとき
(2) (1) 以外のとき

表2: ACCESS_SSRAM (READ)

現状態	次状態	出力		
		X_GW	X_OE	X_READY
S0	S1 (1)	1	1	1
	S0 (2)			
S1	S2	1	1	1
S2	S3	0	1	0
S3	S4	1	1	0
S4	S0	1	1	1

(1) X_AS = '0' and X_CS2 = '0' のとき
(2) (1) 以外のとき

表3: ACCESS_SSRAM (WRITE)

3. インタフェース部 (**INTERFACE**)

各部分を統合制御し、プロトコル・プロセッサとのインタフェースを担当する。

- バッファ

D_OBUF　データの出力バッファ。BT_ROM, SEC_ROM からの出力データのバッファはそれぞれ、D_OBUF0, D_OBUF1 が対応する。

X_RDY_BUF, SSRAM_X_RDY_BUF　それぞれGEN_X_RDYとACCESS_SSRAMによって生成されたX_READYと出力をつなぐためのバッファ。

- インタフェース

  アドレス　READ_BT_ROMへのアドレスはADR_IBUFの下位0ビット目から接続する。一方、READ_SEC_ROMへのアドレスは32ビット幅なので下位2ビットが必要ない。したがって下位3ビット目から接続する。

  データ　D_OBUFからデータバスへの出力は、RDWRが'1'のとき対応するX_CSが'0'であるD_OBUFをデータに出力する。RDWRが'0'のときはデータバスへの出力はハイインピーダンスにする。

  レディー　X_RDY_BUF, SSRAM_X_RDY_BUFををそれぞれCS1 = '0', CS2 = '0'のとき、クロックの立ち上がりでX_READYとして出力する。それ以外の場合は、X_READYとして'1'を出力する。

3.5.3　シンクロナスSRAMインタフェースの検証

以上の設計によりプロトコル・プロセッサをブートさせる。今回はROMのプログラムを2つ設計した。まずプログラム1でブートしシンクロナスSRAMにデータを書き込む。次にプログラム2でブートしそれを読み込むことにした。各CSにおいて実行される命令は以下のように設計した。

- プログラム1

  CS0　BT_ROMが読み込まれる。命令の内容はブート回路と同じ。各種設定を終えるとブート領域からCS1の最初へとジャンプする。

  CS1　SEC_ROMが読み込まれる。命令の内容は、CS2に対して「CS1の最初へジャンプせよ」という命令を書き込む。この処理を終えるとアイドル状態（ループ）に移行する。

  CS2　シンクロナスSRAMに割り当てられている。CS1における命令によってデータが書き込まれる。

- プログラム2

  CS0　BT_ROMが読み込まれる。命令の内容はプログラム1と同じ。各種

設定を終えるとブート領域からCS1の最初へとジャンプする。

CS1　SEC_ROMが読み込まれる。CS2の最初へとジャンプする。

CS2　シンクロナスSRAMの内容が読み込まれる。プログラム1で設定された「CS1の最初へジャンプせよ」という命令が実行され、CS1の最初へジャンプする。

プロトコル・プロセッサのブート同様、プログラム2においてロジックアナライザを用いてX_CS0, X_CS1, X_CS2を観察した様子を図15に示す。CS0 → CS1 → CS2 → CS1 → CS2 → ... という繰り返しが認識されたので、プロトコル・プロセッサが先ほど述べた通りに動作していることが確かめられた。今回の設計で、あらかじめシンクロナスSRAMに書き込んでおいたプログラムを実行することができるようになったことが確認された。

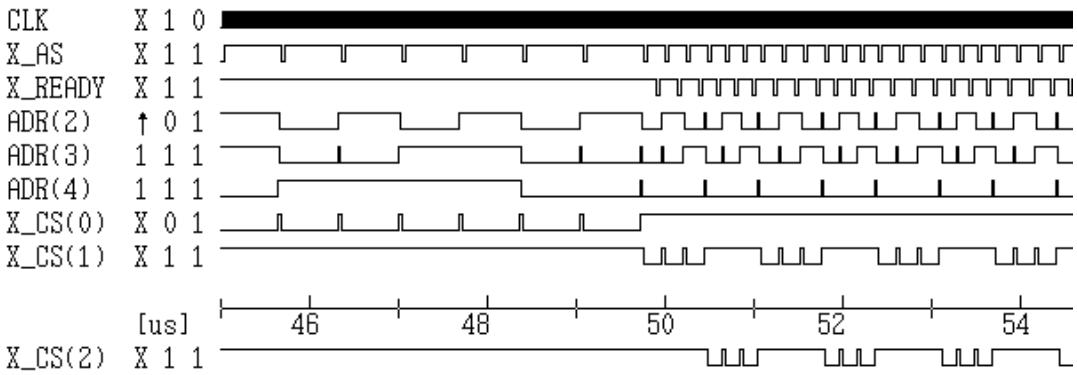

図15: ロジックアナライザ

第4章　おわりに

本稿では、ネットワークによって結合されたコンピュータ資源の有効利用を目指す分散システム「コンピュータ・コロニー」を提案し、そのプロトタイプ・ハードウェアの専用高速通信ボードCNICについて述べ、CNICにおけるプロトコル・プロセッサ・インターフェースの設計、実装について述べた。

コンピュータ・コロニーでは通信にかかるコストを削減し、従来の分散システムでは困難であった細粒度の並列処理を目指している。そのために我々が開発しているプロトタイプ・ハードウェアでは、分散共有メモリベースの通信を行

う際に必要な物理アドレスとネットワークアドレスの変換、またはその際に必要なテーブルの管理等をCNIC上のプロトコル・プロセッサに担当させる。また、CNICが直接リモートアクセスの必要性を感知しホームノードとの通信を行い、リモートアクセスにおけるキャッシングやキャッシュコヒーレンスの制御を行う。これによって、システムコールなどの通信コストを削減させることができるのである。

今回の設計でシンクロナスSRAMに書かれたプログラムをプロトコル・プロセッサが実行できるようになったので、まずプロトタイプCNICではSPARCstation20側からMBusを通してシンクロナスSRAMにプログラムを書き込むことができるようにする。こうすることでFPGA内のROMでは実現困難なある程度大規模なプログラムも実行できるようになる。また書き換えもROMに比べ簡単に行うことができるようになり様々なプログラムを評価できる。さらにFibre Channelを通して通信ができる環境を整えれば、プロトタイプCNIC同士をつなげて通信の実験、評価も行えるようになる。

今後はCNIC開発の第1段階としてプロトタイプCNICの完成を目指し、更なるプロトコル・プロセッサに対するプログラミングやプロトタイプCNICの詳細設計を行う。プロトタイプCNIC完成後、プロトタイプ・ハードウェアに専用オペレーティングシステムColoniaを実装しつつ、高速通信ボードとしてのCNICを完成させるとともに、、様々な通信方式での実験を行い考察を行っていく。

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚の謝意を表します。

また、本研究に関して適切なご指導を賜った森眞一郎助教授、五島正裕助手に深く感謝いたします。

最後に、本研究の共同研究者である増田峰義氏、鳥崎唯之氏、鳥居大祐氏をはじめとして、日頃様々な角度から助力してくださった京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝いたします。

参考文献

[1] 青木秀貴, 他:共有メモリベースのシームレスな並列計算機環境を実現するオペレーティングシステムの構想, 情報研報,97-OS-34(1997)

[2] 山添博史, 他:並列アプリケーションを指向した分散システム コンピュータコロニーの構想, 情報研報,97-OS-76(1997)

[3] 伊達新哉, 他:コンピュータ・コロニーを実現する高速通信機構, 信学技報,CPSY99-52(1999)

[4] 伊達新哉:並列応用を指向した分散システムのプロトタイプ ハードウェア, 特別研究報告書, 京都大学工学部情報学科 (1998)

[5] 深山正幸, 他:HDL による VLSI 設計 (1995)