修士論文

ハード/ソフト・コラーニングシステム上での FPGA ボードコンピュータの設計と実装

氏　　名	**:**	**池田 修久**
学 籍 番 号	**:**	6124020008-1
指 導 教 員	**:**	**山崎 勝弘 教授**
		小柳 滋 教授
提 出 日	**:**	2004 年 2 月 16 日

立命館大学大学院　理工学研究科　情報システム学専攻

目次(INDEX)

図目次

表目次

内容梗概

本研究ではプロセッサアーキテクチャを体系的に学習する教育システムであるハード/ソフト・コラーニングシステムの設計と提案を行った。ハード/ソフト・コラーニングシステムは、アーキテクチャ理解を目標としたソフトウェア学習と、実際に設計を行うことでアーキテクチャ理解を深めるハードウェア学習によって構成する。本システムは、GUI ベースの命令セットシミュレータ、命令レベル並列処理において異なるアーキテクチャをもつ複数のプロセッサ、及びプロセッサを FPGA 上で動作させる FPGA ボードコンピュータから構成される。本研究ではプロセッサの設計の一部と FPGA ボードコンピュータの実装を行い、Celoxica(株)の RC100FPGA ボード上での正常動作を確認した。設計に関しては、メモリ以外は IP を使用せず全て VerilogHDL による RTL 設計を行った。

FPGA ボードコンピュータは、MPU、DMA Controller、BUS Controller、Board Sequencer、命令メモリ、データメモリから構成される。MPU コアを単一サイクルアーキテクチャとした時の実装規模は 130,000 システムゲートであり、Spartan2FPGA の 62%のリソースを使用する結果となった。また最高動作周波数は 21,03MHz であり、システム要求仕様である 10MHz を満たしている。

本論文では、まずハード/ソフト・コラーニングシステムにおけるソフトウェア学習とハードウェア学習の詳細について述べ、FPGA ボードコンピュータの設計、実装、評価を行う。

1. はじめに

1.1 研究背景と目的

近年の急速な半導体集積技術の進歩によって、1 チップ上に実装可能なゲート数が著しく上昇している。その結果、LSI の性能向上、小型化、省電力化などが可能となった。しかし、集積技術の進歩が非常に速いにもかかわらず、LSI の設計生産性の向上が追いついていないため、設計生産性危機として問題化してきている[1]〜[3]。設計生産性危機に対処するため、ハード／ソフト・コデザイン[4]やコンフィギャラブル・プロセッサ[5]などの技術が注目されている。

ハード／ソフト・コデザイン（Hardware-Software Co-design）とは、対象となる組み込みシステムに対して、システム設計の段階からハードウェア設計者とソフトウェア設計者が協調してシステム全体が最適(性能・コスト・消費電力など)になるように、設計・評価(トレードオフ)しながらハードウェアとソフトウェアの分割とインタフェースを決定する設計手法である。ハード／ソフト・コデザインの一手法として、システムレベル記述言語(SpecC、SystemC など)を用いてシステム全体を記述し、制約条件を与えながら多数の実装パターンを比較・評価する方法が用いられる。しかし、計算機がハードウェアとソフトウェア分割の最適解を導き出す訳ではなく、最終的な分割の決定はあくまで人間に委ねられる。そのため、システム設計者にはハードウェア、ソフトウェア、さらにはシステム設計に関する体系的な知識が必須である[6]。

システムレベルでの最適化が求められる現在の組み込みシステムにおいて、システムに占めるソフトウェア量は機能の向上に応じて急速に増加しており、最近の携帯電話などのソフトウェア開発量はコンピュータの OS に匹敵する[7]。そして限られたメモリ容量の中では、アセンブリレベルでの最適化も必要となる。ここで求められる能力とは、いかにハードウェアを熟知しているか、つまりプロセッサの潜在能力(アーキテクチャ)を十分に活用するプログラミング能力である。

組み込みシステムに用いられるプロセッサは、PC や WS などで多数の応用プログラムの実行を想定した汎用プロセッサである必要はなく、そのシステムに特化した専用プロセッサである場合が多い。システムにおけるソフトウェア量が増加している現在、システムに柔軟性を持たせつつ、厳しい要求仕様を満たしたいという背景から、専用プロセッサを設計するコンフィギャラブル・プロセッサが注目されている。しかし、コンフィギャラブル・プロセッサの設計には、プロセッサアーキテクチャを正しく理解していることが大前提であり、さらに対象となるアプリケーションを効率良く動作させる命令やプロセッサ構成を導き出す設計空間探索能力が求められる[5]。コンフィギャラブル・プロセッサの生成技術に関しては次節にて述べる。

このように、組み込みシステムの設計にはシステムレベルでの最適化からソフトウェア、ハードウェアでの最適化が求められる。そして産業界では、システムを体系的に理解し、具体的な実現技術(ハードウェア、ソフトウェア、実装等)を幅広く習得した上で、要求仕様

と制約条件のもとに組み込みシステムを実現して行く能力を持ったシステムアーキテクトの育成が望まれており[8]、立命館大学でも 2001 年度から株式会社半導体理工学研究センターSTARC 指導の下、MELPEC(Micro-Electronics Professional Engineer Course)にて寄附講座がスタートした[14]。

ここで、システムを体系的に理解するとは一体どういうことであろうか。ハード/ソフト・コデザインとコンフィギャラブル・プロセッサに共通する要素を考えてみる。ハード/ソフト・コデザインでは、プロセッサで処理するソフトウェアと専用ハードウェアによって最適解を探索し、コンフィギャラブル・プロセッサではプログラムを効率よく実行できるようプロセッサを再構成する。つまり、プロセッサの性能を基準にして最適化が行われるのであり、プロセッサのアーキテクチャを理解することがシステムを体系的に理解することに繋がると考えられる。

大学におけるコンピュータアーキテクチャ学習は多くの報告事例がある[15]〜[25]。そこで我々は、プロセッサアーキテクチャを体系的に学習し、その上で最適なソフトウェアを設計することを目標とする教育システムを構築する。すなわち、観測性を重視したシミュレータによる演習と、オリジナルプロセッサを用いたハードウェア設計演習を融合させ、プロセッサアーキテクチャをテーマにハードウェアとソフトウェアをバランスよく学習するハード/ソフト・コラーニングシステムを構築することが本研究の目的である。

1.2 関連研究

（1）コンフィギャラブル・プロセッサ生成技術

専用の命令の付加や、アーキテクチャを自由に設計するコンフィギャラブル・プロセッサ生成技術は開発方法の違いによって、①テンプレート方式と、②プロセッサ記述方式に分類される。表 1 に両者の特徴をまとめる。

表 1：コンフィギャラブル・プロセッサの分類

	テンプレート方式	プロセッサ記述方式
システム例	Xtensa[10], PEAS-Ⅰ,-Ⅱ[5], MeP[11]	LISA[12], MIMOLA, GO, AIDL,
特徴	・基本命令セットが準備 ・RISC アーキテクチャとの相性	・専用の言語を用いてプロセッサの構造や各命令のマイクロ動作を記述
利点	・SW 開発環境の生成が比較的容易 ・ニーズが合えば高い性能を発揮	・設計の自由度大 ・スーパースカラ、VLIW も記述可能
問題点	・アーキテクチャ選択・制御方式の自由度が低い ・CISC 系命令の実現が難しい	・専用言語の学習が必要 ・応用システム設計者向きではない ・設計空間の探索には適さない

テンプレート方式はアーキテクチャ選択などの自由度は低いが、ニーズが合えば性能を発揮することができ、Xtensa(Tensilica 社)のベンチマークでは、MP3 プレーヤーなどの音

声処理において、ARM10 や PowerPC に比べて最大 2 倍の高速化を実現したとの報告がある[13]。一方、プロセッサ記述方式は、コンパイラ生成からのアプローチで発展した技術であり、専用の記述言語を用いて自由にアーキテクチャを記述できる反面、論理合成が不可能で、コンピュータアーキテクチャに精通していないと使えないなどの問題もある。

（2）大学における計算機工学教育

（A）教育用プロセッサを用いたシステム

- PICO²[15]

  慶応義塾大学で開発された教育用パイプラインプロセッサ PICO² を用いたシステム。4 ビットから 16 ビット命令セットのフレームワークが用意されており、学習者は簡単な命令を持つパイプラインプロセッサを改良し、命令やフォワーディングやストールの機能を追加しながら、パイプラインプロセッサの完成を目指す。

- City-1[16]

  広島市立大学が開発。命令セットアーキテクチャや、アドレス空間を自由に設計させるシステム。合成可能だが不完全な記述を学生に提供し、それを元にプロセッサを設計させる。

- KITE[17]

  九州工業大学が開発した教育用プロセッサ KITE を用いた学習システム。16 ビット長で教育用として最低限の命令を持ち、アーキュムレータ方式で動作する KITE1 と、割り込み処理などを追加した KITE2 の 2 種類がある。KITE システムの特徴は、観測性のあるボードコンピュータとホームページを利用した e-Learning の充実である。

（B）プロセッサシミュレータ

- Mikage[18]

  広島市立大学が開発した教育用スーパースカラプロセッサシミュレータ。最適なスーパースカラプロセッサを設計するためのハード/ソフト評価ツール。パイプラインの使用状況を可視化し、スーパースカラの並列度が可変。

- DLX-View[19]

  観測性と対話性を兼ね備えたパイプラインプロセッサシミュレータ。プロセッサの動作を正しく理解するのが目的で、命令セットの理解、デバッグ、プロセッサの性能評価にも使用できる。

- VisuSim[20]

  香川大学が開発した計算機の内部構造や動作原理を視覚的に理解させることが目的のシミュレータ。Web 上からダウンロードして使用することを考慮に入れた設計となっている。

（C）ハード/ソフト協調学習システム

- 港[24]

拓殖大学が開発したシステムソフトウェア教育支援環境。プロセッサ、OS、コンパイラの相互関係を意識しながら改良が行えるという実装的観点からのアプローチを取ったシステムプログラミング学習環境である。つまり、プロセッサ、OS、コンパイラの実装知識をバランスよく身に付ける事ができるシステムである。

- SEP4[25]

静岡大学が開発した 16 ビット 3 段パイプラインプロセッサ SEP4 を用いたハード・ソフト協調学習システムである。ハードウェア設計では、ASIP Meister を用いてパイプラインプロセッサ設計を行う。ソフトウェア演習ではハザードを回避するための命令スケジューリングなどを行う。

1.3 本研究の概要と論文の構成

本研究では他大学における計算機工学教育を参考に、現在のシステム LSI 設計には何が必要かを考察した上で、プロセッサアーキテクチャを体系的に学習できる教育システムであるハード/ソフト・コラーニングシステムの設計を行った。

ハード/ソフト・コラーニングシステムでは、アーキテクチャ理解を目標としたソフトウェア学習と、実際にプロセッサ設計を行うことでアーキテクチャ理解を深めるハードウェア学習によって構成する。ソフトウェア学習では、いかにプロセッサアーキテクチャを教育するかということを考察し、以下の 2 つの基本方針を決定した。

- 複数のプロセッサアーキテクチャを用意
- 視覚的な理解を促す GUI ベースのシミュレータ

簡単なアーキテクチャから体系的に学習させることで、パイプライン技術やスーパースカラ技術の理解を目指す。そのために本システムでは、単一サイクルアーキテクチャ、マルチサイクルアーキテクチャ、パイプラインアーキテクチャ、スーパースカラアーキテクチャを用意し、順を追って体系的に理解させる。また、その理解には視覚的なアプローチが必要であると考え、本システムの特徴となるような GUI ベースのシミュレータの開発を行った[33]。そのシミュレータを用いてアセンブリプログラムを実行させることで、命令セットとプロセッサアーキテクチャの関係理解を促す。

一方ハードウェア学習では、ソフトウェア学習によるアーキテクチャ理解の確認と実践の意味を込め、以下の 3 通りの基本方針を決定した。

- アーキテクチャ理解のためのプロセッサ設計
- ハードウェア設計手順の理解
- FPGA への実装

シミュレータを用いて理解したプロセッサアーキテクチャを、実際に設計することでより深く学ぶのがハードウェア学習の目標である。基本的な要素として、ハードウェア設計手順の理解があげられる。ハードウェア設計もシミュレーションで終わるのではなく、実際に FPGA へと実装し、動作確認を行う。そのためには、FPGA 上に MPU コアとその周

辺回路を搭載するコンピュータシステムを実現する必要がある。そこで、本研究ではハードウェア学習において、柱となるコンピュータシステムの設計を行った。

FPGA ボードコンピュータとは、ハード/ソフト・コラーニングシステムにおける実機検証に用いるプロセッサ検証システムである。Xilinx 社の SpartanⅡFPGA 上に MPU とその周辺回路を実装した。本論文では主に FPGA ボードコンピュータの設計と実装に関して述べる。設計に関しては、メモリ以外は IP を使用せず全て VerilogHDL による RTL 設計を行った。ターゲットとなる FPGA ボードは Celoxica㈱の RC100 ボードである。

本論文では、まず 2 章にてハード/ソフト・コラーニングシステムの設計について述べる。3 章では FPGA ボードコンピュータの設計、4 章にて FPGA ボードコンピュータの実装、そして 5 章にて FPGA ボードコンピュータの評価と今後の課題を述べる。6 章にて現在までの成果と今後の課題について述べる。

2. ハードソフト･コラーニングシステムの設計

2.1 システムの概要

ハード/ソフト･コラーニングシステム(以下コラーニングシステム)はプロセッサアーキテクチャを意識したプログラミング学習を行うためのハードウェアとソフトウェアの協調学習システムである。ソフトウェア面ではアーキテクチャが可変な命令セットシミュレータを用いて、プロセッサアーキテクチャの理解、アセンブリ言語や C 言語で設計したプログラムや命令セットの評価を行う。また、最適化コンパイラの設計を通してアーキテクチャの更なる理解を促す。ハードウェア面ではシミュレータで理解したプロセッサアーキテクチャの知識を基に、まず HDL によるプロセッサ設計を行う。次に、設計したプロセッサを FPGA ボードコンピュータに搭載し、周辺回路、及び開発したソフトウェアと共に実機検証を行う。このように、HDL によるプロセッサ設計とソフトウェア開発を融合させることで、アーキテクチャを意識したプログラミングが行えるようになることがこのシステムの目的である。

2.2 システムの特徴

本システムの特徴は、ソフトウェアシミュレータによる可観測性とハードウェア設計を融合した、ハードウェアとソフトウェアの協調学習である。同一の命令セットで 4 種類のプロセッサアーキテクチャ(単一サイクル、マルチサイクル、パイプライン、スーパースカラ)を選択可能にすることで、体系的なプロセッサアーキテクチャの理解を促す。また、命令実行の様子やレジスタの中身を可視化した命令セットシミュレータを用いて、アーキテクチャを変更しながらソフトウェア開発を行うことで、アーキテクチャに適したプログラミングや、プログラムにあったプロセッサアーキテクチャの探索が可能になる。そしてプロセッサアーキテクチャに対する最適化コンパイラ設計を通じて、プロセッサアーキテクチャの理解を深める。さらに、HDL(Hardware Description Language)によるプロセッサ設計を行い FPGA に実装することで、シミュレータだけでは理解できないハードウェア設計についても学習することができる。

2.3 システムのフロー

コラーニングシステムは、命令セットシミュレータを用いてアセンブリプログラミングと C 言語プログラミングを行うソフトウェア学習、及び MPU コアを設計し、DMA Controller などの周辺回路と共に FPGA に実装して実機検証を行うハードウェア学習から構成される。図 1 にハード/ソフト･コラーニングシステムの全体像を示す。

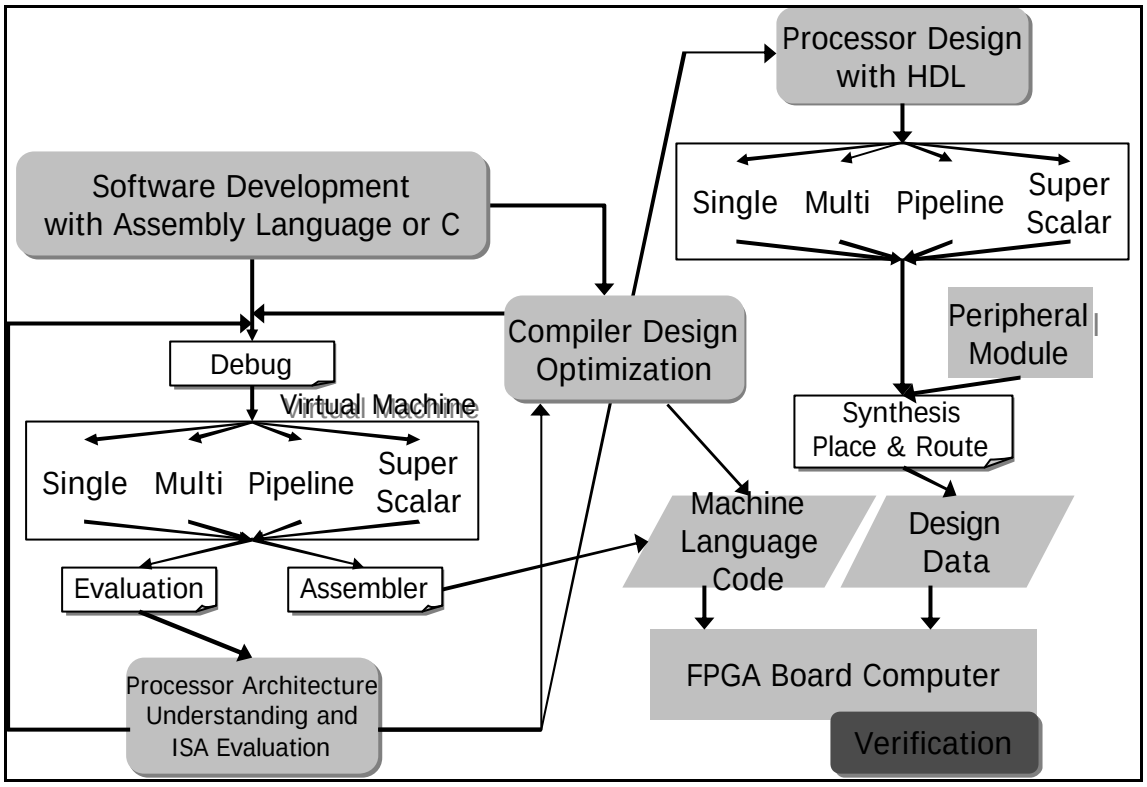

図 1：ハード/ソフト･コラーニングシステム

図 1 の左半分がソフトウェア学習、右半分がハードウェア学習である。次節においてソフトウェア学習、ハードウェア学習の概要を示す。

2.4　ソフトウェア学習の提案

ソフトウェア学習では、C 言語やアセンブリ言語によるソフトウェア開発を行った後、命令セットシミュレータを用いてデバッグを行う。また同一のソフトウェアを、アーキテクチャを変更した上で実行させることで、プロセッサアーキテクチャの違いによる評価･比較を行う。そして、プロセッサアーキテクチャの特徴を理解した上で、最適化コンパイラ設計を行い、プログラムとプロセッサの関係をより密なものとする。

このソフトウェア学習で用いる命令セットシミュレータの特徴を以下に述べる。

- **4 種類のプロセッサアーキテクチャを選択可能**
- **プロセッサで命令が実行されている様子をデータパスの強調表示で可視化**
- **レジスタやメモリの内容の表示**
- **複数の実行モード**
- **ハザード通知**
- **アーキテクチャやプログラムの評価シートの出力**

上記の特徴を持たせることで、HDL によるプロセッサ設計を行う前にプロセッサアーキ

テクチャを理解するために利用するだけでなく、ソフトウェア開発とデバッグにも利用できる。命令セットシミュレータの概観を図 2 に示す。

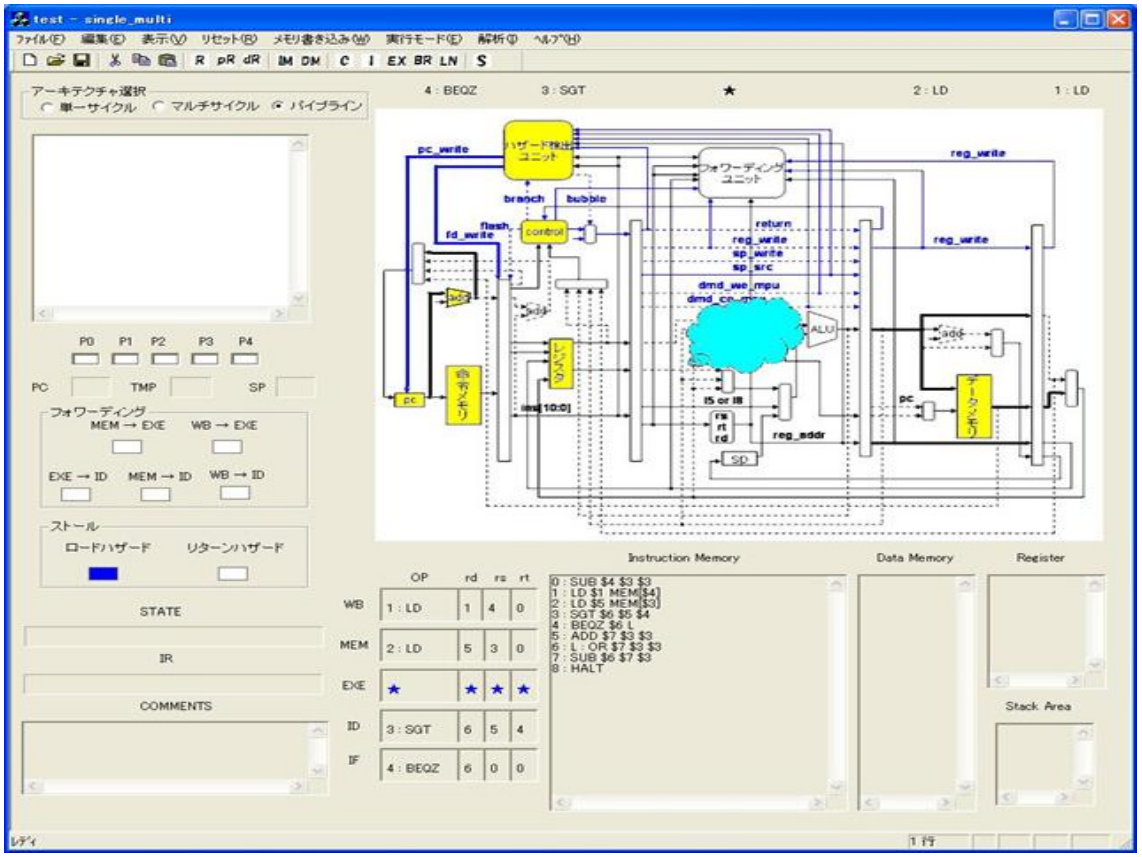

図 2：MONI 命令セットシミュレータの概観

2.5　ハードウェア学習の提案

ハードウェア学習では、2 通りのアプローチによる学習方法が考えられる。1 つはソフトウェア学習の延長に位置する学習で、プロセッサアーキテクチャを理解した後、実際に VerilogHDL を用いて MPU コアの設計を行う。ここで設計のターゲットとする MPU アーキテクチャはシミュレータにおいて学習した 4 種類のアーキテクチャであり、シミュレータにて表示されるデータパスを再現すべく RTL 設計を行う。MPU コアの設計は PC を用いてデバッグを行い、MPU 単体でのゲートレベルシミュレーションを完成させる。その後、設計済みの周辺モジュール(BUS Controller、DMA Controller、命令メモリ、データメモリ、Board Sequencer)を結合し、ボードコンピュータシステム全体を配置配線にかける。そして得られた構成データを FPGA へダウンロードし、シミュレータにてデバッグを終えたソフトウェアプログラムと変数データを命令メモリ、及びデータメモリに格納した上で、実機検証を行うというアプローチである。これはプロセッサアーキテクチャの理解を目標とした学習である。

もう一つのアプローチとして、コンフィギャラブル・プロセッサ合成技術に必要な空間探

索能力を身に付ける学習である。これは数種類の適当な C 言語プログラムを与え、それらを実行させる必要最小限の命令セットを自由に創造させる。そして MPU アーキテクチャに関しては自由に設計させる。この学習によって、C 言語プログラムを独自に定義したアセンブリプログラムに変換した際の静的な命令数や、実行した際の動的な命令数、そして CPI などのデータを比較･評価することで、与えられた仕様に対する最適解を考察する能力を身に付けることができると考える。図 3 に設計空間探索能力の育成学習のイメージを示す。

図 3：ハードウェア学習における設計空間探索能力の育成

上記に示す通り、コラーニングシステムにおいて命令セットシミュレータと MPU コアの設計を繰り返すことで、プロセッサアーキテクチャの体系的な学習が可能であり、ハードウェアとソフトウェアの協調学習を実現し、ソフトウェア開発、及びハードウェア設計における基礎技術を身に付けることができる。現在、命令セットが固定であるが、将来的に命令セットシミュレータにおいて自由な命令セットの設計を可能とすることで、現在のプロセッサ合成技術に必要な設計空間探索能力を身に付けることができるシステムになるのではないかと考える。

3. FPGA ボードコンピュータの設計

近年、FPGA の大容量化と低価格化が進んでおり、ソフト IP としての MPU コアを論理合成し、システムの一部として FPGA に組み込むことも可能となった。Xilinx 社の Vertex2 FPGA のように、PowerPC をハードマクロとして内蔵している FPGA も登場している。ASIC に比べて部品単価が高かった FPGA だが、低価格化が進むにつれ ASIC 同様の使われ方も可能となる。Xilinx 社の Spartan3 は 100 万ゲートを 10 ドルで実現しており、今後のユビキタス社会において FPGA は我々の生活により身近な部品となる。FPGA の使われ方に関して今後は、仕様の変更が考えられる部位を、FPGA を用いて専用ハードウェア化し、今後の仕様変更にも柔軟に対応する方法と、前述した通りソフト IP としての MPU コアを用いてソフトウェアで実現する方法がある。今後はシングルプロセッサによる性能向上というアプローチより、FPGA の柔軟なハードウェアの特徴を活かして、対象となるアプリケーションによってマルチプロセッサ環境を実現し、最適なシステムを構築していくことも可能である。

書き換え可能という特徴を持つ FPGA は、ハードウェア設計教育には最適な教材である。コラーニングシステムにおけるハードウェア学習でも、FPGA を対象として MPU コアを設計する。FPGA を対象とすることで、実機動作検証を容易にし、今後の仕様変更や機能拡張にも柔軟に対応できる。今回 FPGA ボードコンピュータを実装する上で、「1 チップでコンピュータシステムを実現する」という点と、「システムの汎用性を高める」という点を考慮してシステムの仕様を策定した[31]。本章では、FPGA ボードコンピュータのシステム構成と内部モジュールについて説明する。

3.1 FPGA ボード

今回のコラーニングシステムにおける FPGA ボードコンピュータの対象とする FPGA ボードは Celoxica 社の RC100 ボードとする。このボードはスタンドアロンで使用可能な FPGA ベースのプロトタイピング用ボードであり、Xilinx 社の 200,000 システムゲート規模の FPGA SpartanⅡ(部品番号:XC2S200-5-FG456)、多種の FPGA コンフィギュレーションや一般データを記憶するための 8MB FlashRAM(Intel StrataFlash)などが搭載されている。図 4 に RC100 ボードのモジュール接続関係を示す。

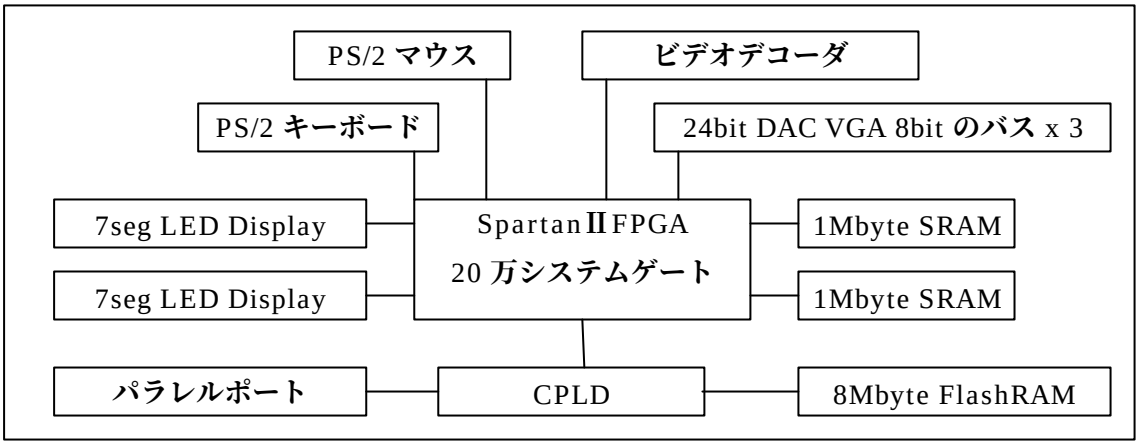

図 4：RC100 ブロックダイアグラム

本システムでは図 4 のうち、色づけされたデバイスを用いて FPGA ボードコンピュータを実現する。各モジュールの詳細に関しては本論文では割愛し、付録 1 にて詳細を述べる。

3.2　システムの構成と実行フロー

RC100 ボードを用いて実装する FPGA ボードコンピュータは、FPGA1 チップで MPU とその周辺回路(DMA Controller、BUS Controller、Board Sequencer、命令メモリ、データメモリ)を実現する。そのため、MPU が使用する RAM には後述する FPGA 内の BlockRAM を用いる。またシステムの汎用性を高めるという理由から、MPU コア以外の周辺回路のインタフェース部を共通のものとして設計し、コラーニングシステムにおける多様な MPU コアの搭載を可能とする。図 5 に FPGA ボードコンピュータのモジュール接続を示す。

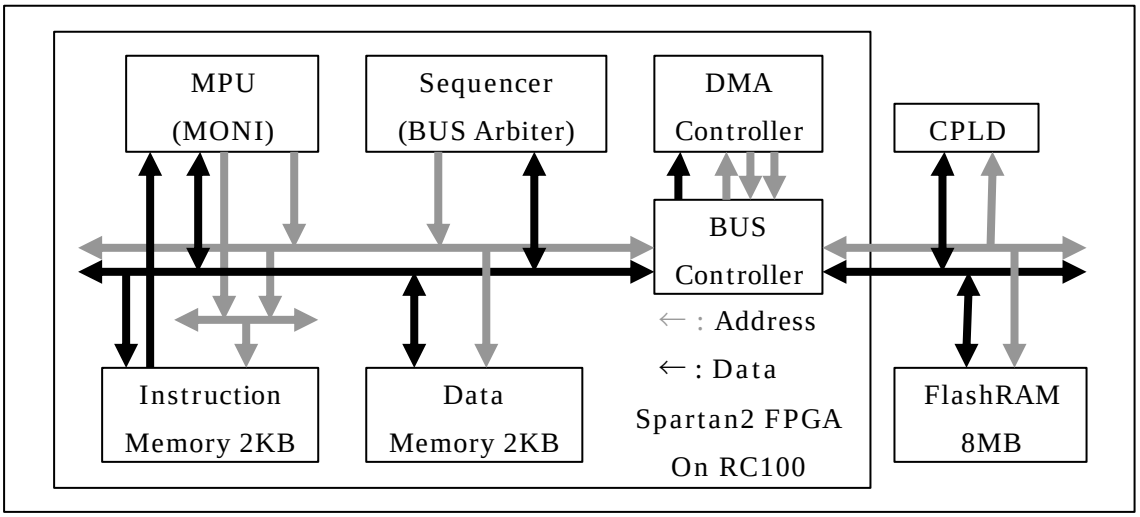

図 5：FPGA ボードコンピュータモジュール接続

図 5 に示す通り、MPU などの FPGA 内部モジュールは 2 本のバス(アドレスバスとデータバス)に接続される。また BUS Controller がブリッジとなり、FPGA 外部の FlashRAM とバスを結合する。そして本システムではそれらのモジュールを同一のアドレス空間(システムアドレス空間)で管理する。図 6 にシステムアドレス空間のマッピング例を示す。

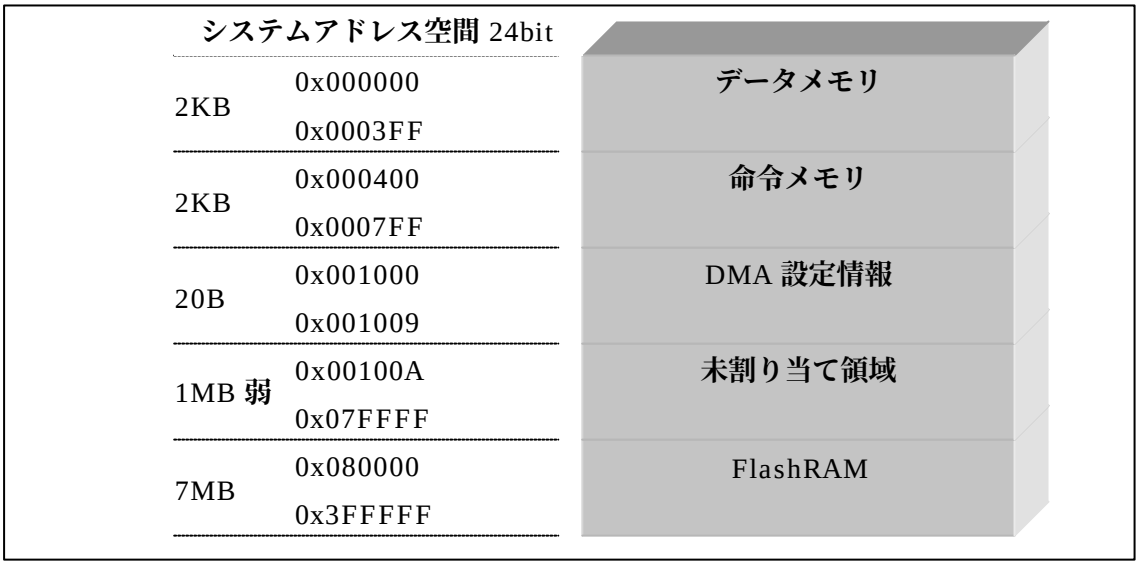

図 6：システムアドレス空間のマッピング

システムアドレス空間は、1 アドレス 1 ワードを基本とする。本システムでは図 6 に示すように、FlashRAM の 24 ビットアドレス 8MB 空間のうち、1MB を内部モジュールに割り当て、共有化を図っている。0x001000~0x001009 は DMA Controller 内のメモリマップドされたアクセスレジスタを示し、DMA 転送設定の際はこのアドレスにアクセスする。

今後、RC100 ボードに搭載されている SSRAM をシステムに組み込む際も、同一のアドレス空間に割り当てることでシステムの拡張性を確保できる。このように、システムアドレス空間を共有することで、システムの汎用性を高めることができる。

FPGA ボードコンピュータは、後述する Board Sequencer モジュールによって図 7 に示すシーケンスが制御される。

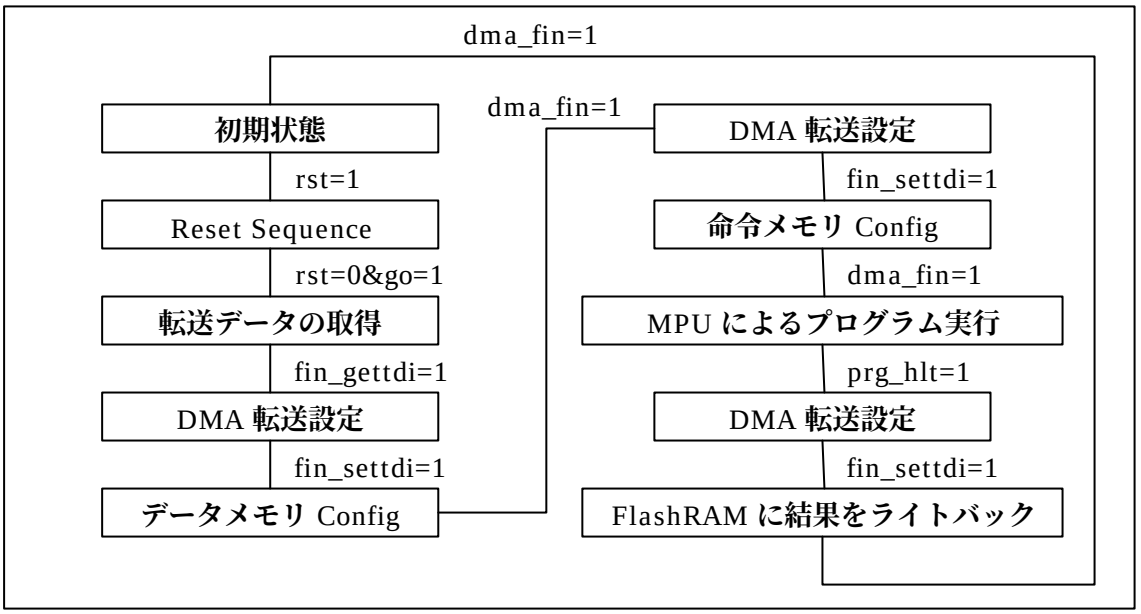

図 7：Board Sequencer のシステム制御

FPGA ボードコンピュータは、FlashRAM に格納された FPGA に構成データがダウンロードされる初期状態に始まり、MPU によって実行された演算結果を最終的に FlashRAM に書き戻すまでの 10 ステップで 1 サイクルの動作を終える。

システム起動時の BlockRAM(命令メモリとデータメモリ)は、中身が空の状態である。そこで DMA 転送によって FlashRAM から BlockRAM へ適当なデータが転送されることによって MPU が実行可能となる。この DMA 転送を設定する際、MPU による命令で設定を行うのではなく、本システムでは TDI(Transfer Data Information)と名付けた 64 ビットレジスタによって転送元、転送先、及び転送量が設定される。この設定は Board Sequencer が DMA Controller 内の 0x001009 番地にメモリマップドされたレジスタにアクセスすることで行われる。図 8 に TDI のビット割り当てを示す。

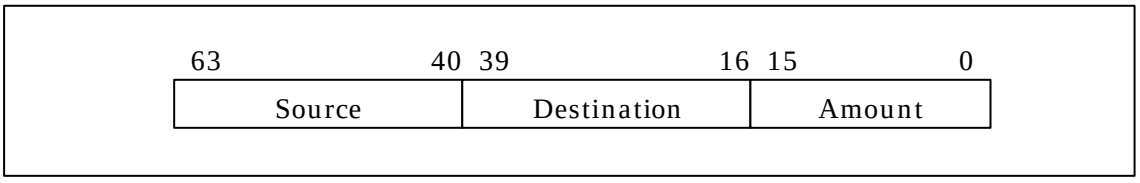

図 8：DMA 転送情報 TDI

TDI は Board Sequencer が FlashRAM の特定番地(0x700000～0x700018)にアクセスすることで Board Sequencer 内のレジスタにロードされる。つまり、システムの起動前に予め FlashRAM 内に TDI を格納しておく必要がある。同様に、DMA 転送によって転送される命令列やデータも FlashRAM 内の特定番地に格納されている必要がある。図 9 に FlashRAM に格納すべきデータとそのアドレスの対応を示す。

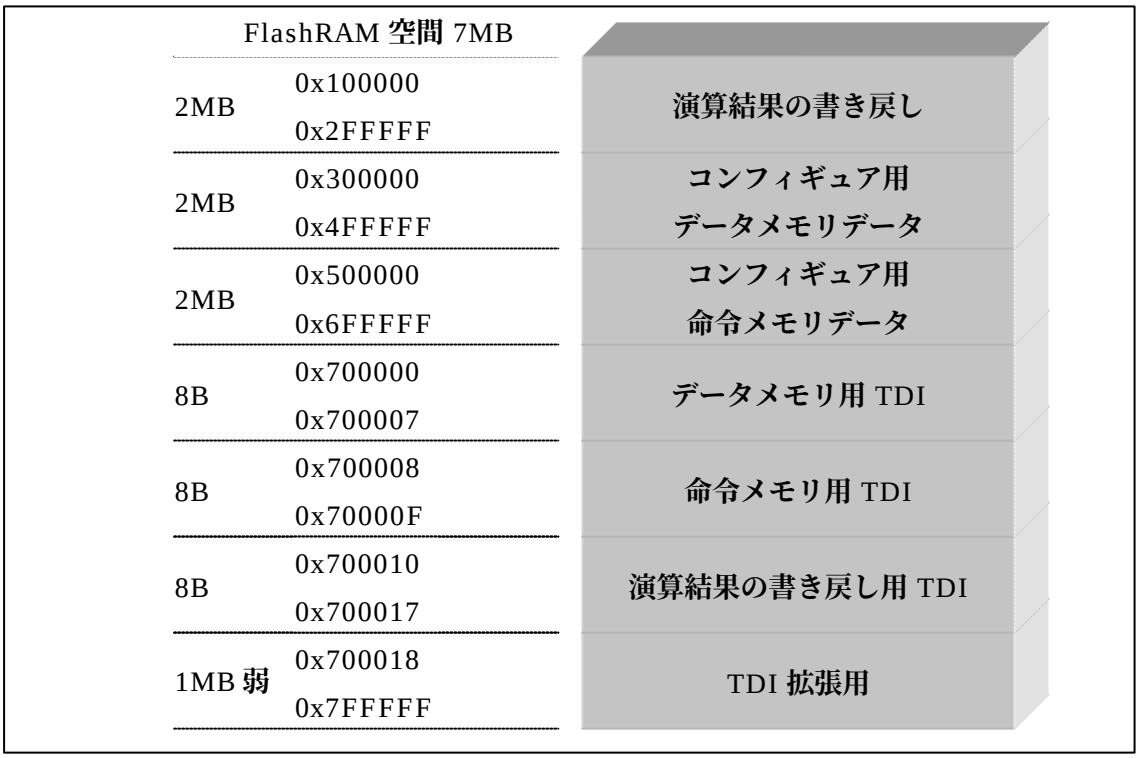

図 9：7MB FlashRAM アドレス空間の格納データ

図 7、図 9 から分かる通り、本システムでは命令メモリ用 TDI、データメモリ用 TDI、MPU による演算結果の確認用 TDI の 3 種類の TDI が必要である。FlashRAM 内には、BlockRAM の構成データ用の領域として命令メモリ、データメモリ共に 2MB を確保している。現在のシステムでは図 7 に示した 1 サイクルが終了するとシステムも停止するため、2MB の領域を確保することは無意味である。このシステムの改良手法に関しては今後の課題として第 5 章において述べる。

3.3 機能モジュールの動作

本節では FPGA ボードコンピュータの内部モジュールに関して述べる。各モジュールの入出力ピンや動作の詳細などは[31]に示すため、本論文では割愛する。この FPGA ボードコンピュータは、立命館大学情報学科 4 回生の中村浩一郎君(コンピュータシステム研究室)と共同で仕様の策定、及び設計を行った[34]。図 10、図 11、図 15 に示す各モジュールの内部モジュールのうち、色づけされた部分の設計を私が担当した。

3.3.1 Board Sequencer

Board Sequencer モジュールは、DMA 転送時と MPU コア動作時のバス･アービタ、システム動作時のエラーメッセージの受信と 7 セグメントディスプレイへのエラー表示、DMA 転送の設定など、FPGA ボードコンピュータにおけるシーケンス制御(図 7 参照)を行

うモジュールである。図 10 に Board Sequencer の内部モジュール構成を示す。

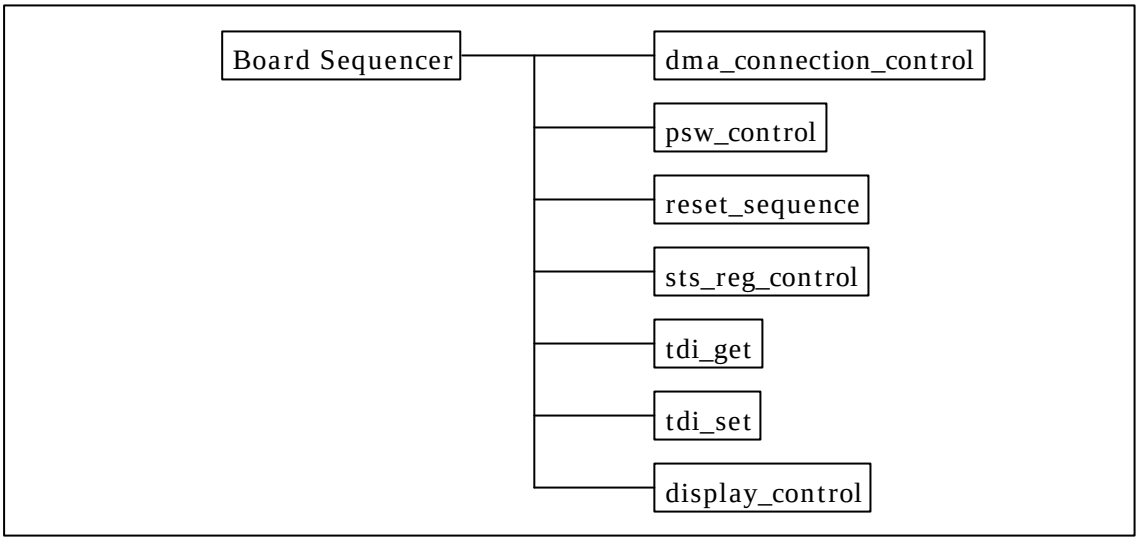

図 10：Board Sequencer モジュール構成

Board Sequencer は図 10 に示す 7 モジュールによって構成される。

- dma_connection_control

dma_connection_control モジュールは、DMA Controller との接続を制御するモジュールであり、図 7 における DMA 転送設定の際に必要となる。DMA Controller とはハンドシェークによって接続を確立し、0x001009 番地にメモリマップドされたレジスタにアクセスすることで接続要求を出す。

- psw_control

psw_control モジュールは、PSW(Program Status Word)を管理するモジュールである。PSW とは、MPU の命令実行権を管理するレジスタであり、1 の時 MPU は動作を開始する。本システムではデータメモリ、命令メモリの DMA 転送が終了後、MPU に対して実行権を与える。Board Sequencer は MPU の HALT 命令実行を確認すると PSW を 0 に変更し、MPU から命令実行権とシステムバス使用権を剥奪する。

- reset_sequence

本システムは RC100 ボードから供給される 80MHz のクロック信号を、FPGA 内部のクロック分周器に通すことで 10MHz と 5MHz に分周している。それらクロックの安定にはある程度の時間が必要となる。そこで Board Sequencer はシステム起動後、適当なクロックサイクル(現在は 127 クロックサイクル)分だけシステムを待ち状態とする。この待ち状態をカウントするのが reset_sequence モジュールであり、内部に 7 ビットのカウンタを持つ。

- sts_reg_control

sts_reg_control は図 7 に示す状態遷移を、3 ビットのレジスタを用いて管理するモジュールであり、本システムの中心制御部である。状態変化のトリガとなる信号は全てこのモ

ジュールに接続される。

- tdi_get

tdi_get は、FlashRAM の特定番地(0x700800～0x700816)番地へアクセスし、3 種類の TDI を内部レジスタにロードするモジュールである。

- tdi_set

tdi_set は、dma_connection_control モジュールによって DMA Controller との接続が確立された後、tdi_get モジュールによってロードされた TDI を DMA Controller に設定するモジュールである。

- display_control

display_control モジュールは、DMA Controller、BUS Controller、及び Board Sequencer 内部モジュールからエラー信号を受信し、エラーの内容を RC100 ボード上の 7 セグメントディスプレイに出力するモジュールである。表 2 にエラー表示と原因モジュールの対応表を示す。

表 2：エラー表示と原因モジュール

display	cause
E1	Board Sequencer
E2	BUS Controller
E3	DMA Controller
E4	DMAC busy

現在のエラー表示は E1～E4 までの 4 通りである。以下にその原因を説明する。

- E1：Board Sequencer

  Board Sequencer 内のモジュールにおいて、TDI が正常にロードできなかった場合、及び DMA Controller との接続が確立しなかった場合に表示される。

- E2：BUS Controller

  BUS Controller のアドレス変換時に、変換元アドレスに以上があった場合(規定範囲を超えているもしくは、それ以下)に表示される。

- E3：DMA Controller

  DMA Controller が DMA 転送アドレスを BUS Controller に出力した後、BUS Controller から受信応答が得られない場合に表示される。

- E4：DMA busy

  DMA Controller の初期化がうまく出来ず、DMA Controller 内のステータスレジスタに以上が発生した場合に表示される。

上記のエラーが発生した場合、システムを再起動するか、もう一度外部からリセット信号を与えてやる必要がある。

3.3.2 DMA Controller

DMA Controller モジュールは FlashRAM-BlockRAM 間の DMA 転送を制御するモジュールである。DMA 転送は、Board Sequencer によって設定された TDI を元に転送を行う。DMA 転送が終了すると、Board Sequencer に DMA 転送の終了信号を送信し、システムバスの使用権を返上する。本システムにおける DMA 転送は、図 7 に示す通り FlashRAM からデータメモリ、FlashRAM から命令メモリ、データメモリから FlashRAM への 3 回発生する。その度に TDI は Board Sequencer によって設定される。図 11 に DMA Controller の内部モジュール構成を示す。

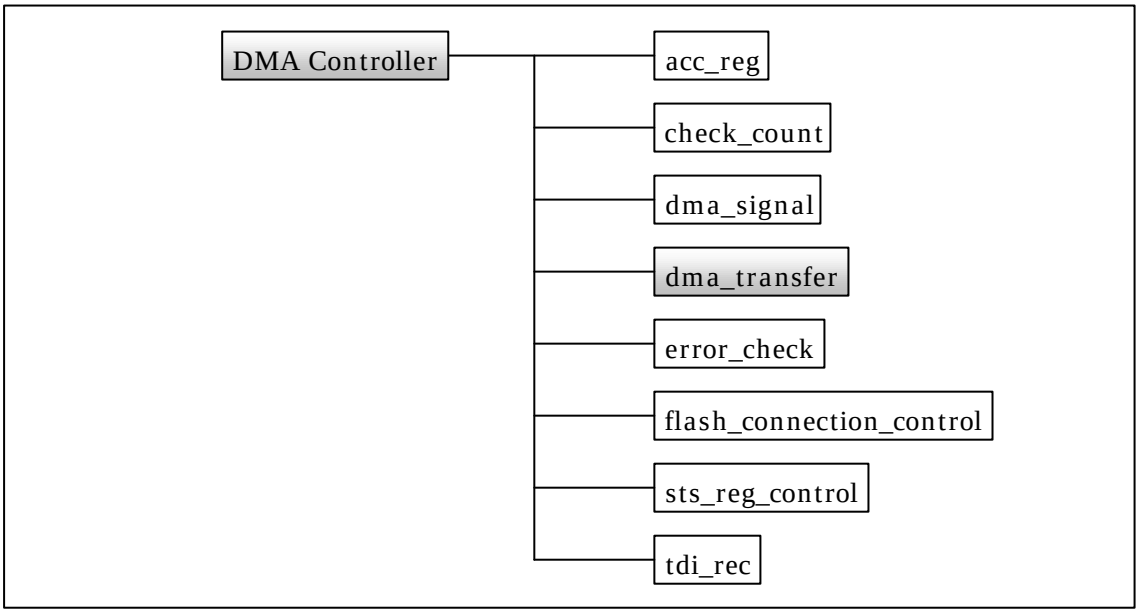

図 11：DMA Controller モジュール構成

DMA Controller は図 11 に示す 8 モジュールによって構成される。

- acc_reg

acc_reg は、Board Sequencer とのハンドシェーク通信を行うモジュールであり、モジュール内にはシステムアドレス空間 0x001009 番地にメモリマップドされたレジスタがあり、Board Sequencer からのアクセス要求があるとアクティブになる。図 12 に acc_reg モジュールの内部構造を示す。

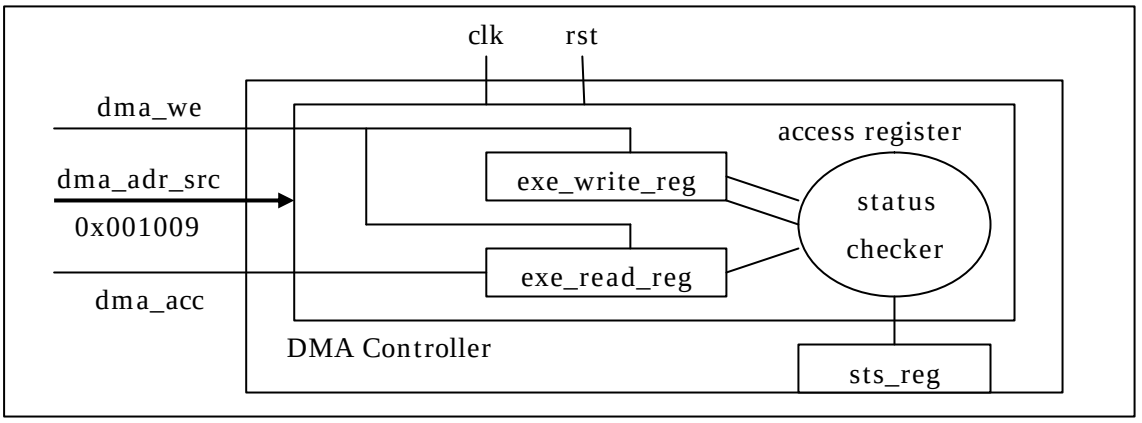

図 12：アクセスレジスタの内部構造

- check_count

check_count は、内部にカウンタを持ち、DMA 転送量をカウントするモジュールである。転送量が TDI にセットされた値と同じになると、Board Sequencer に対して DMA 転送の終了信号を送信する。

- dma_signal

dma_signal は DMA 転送時の転送先と転送元モジュールに制御信号(読み出し信号、書き込み信号)を送信するモジュールである。

- dma_transfer

dma_transfer は DMA 転送の転送先と転送元アドレスを、システムバスアドレス、及び外部アドレスバスに適切なタイミングで出力するモジュールで、転送の中心制御部である。

- error_check

error_check は DMA 転送が適切に行われているかをアドレス転送毎に確認し、エラーが発生した際はエラー信号を Board Sequencer に対して送信するモジュールである。

- flash_connection_control

flash_connection_control は FlashRAM との通信を行うモジュールである。FlashRAM への書き込みは BlockRAM への書き込みのように 1 クロックサイクルで完了するものではなく、数ステップ&数クロックサイクルを要する。その為、ステートマシンによって状態を管理し、FlashRAM との通信制御を行う役割を担う。図 13 に FlashRAM の基本的な書き込みシーケンスを示す。

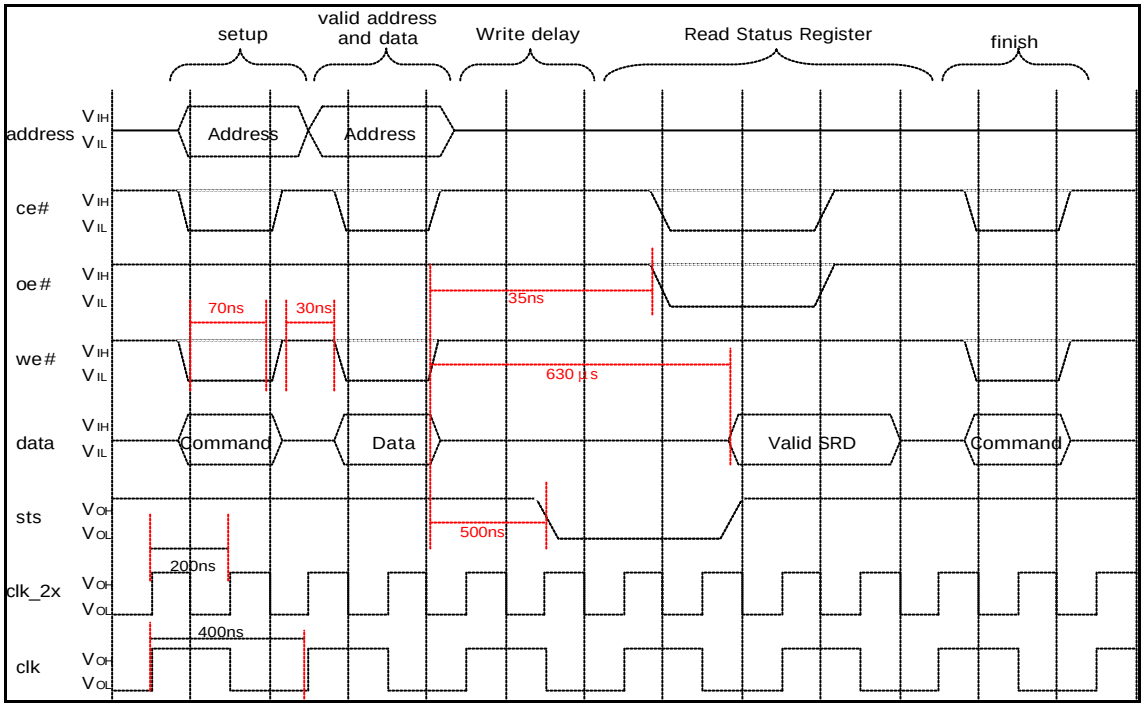

図 13：FlashRAM の基本的な書き込みシーケンス

- sts_reg_control

sts_reg_control は、3 ビットのレジスタを用いて DMA Controller のステータスを管理するモジュールである。acc_reg モジュールや check_count モジュールから状態遷移のトリガとなる信号を受信し、DMA Controller の状態を遷移する。図 14 に DMA Controller の状態遷移を示す。

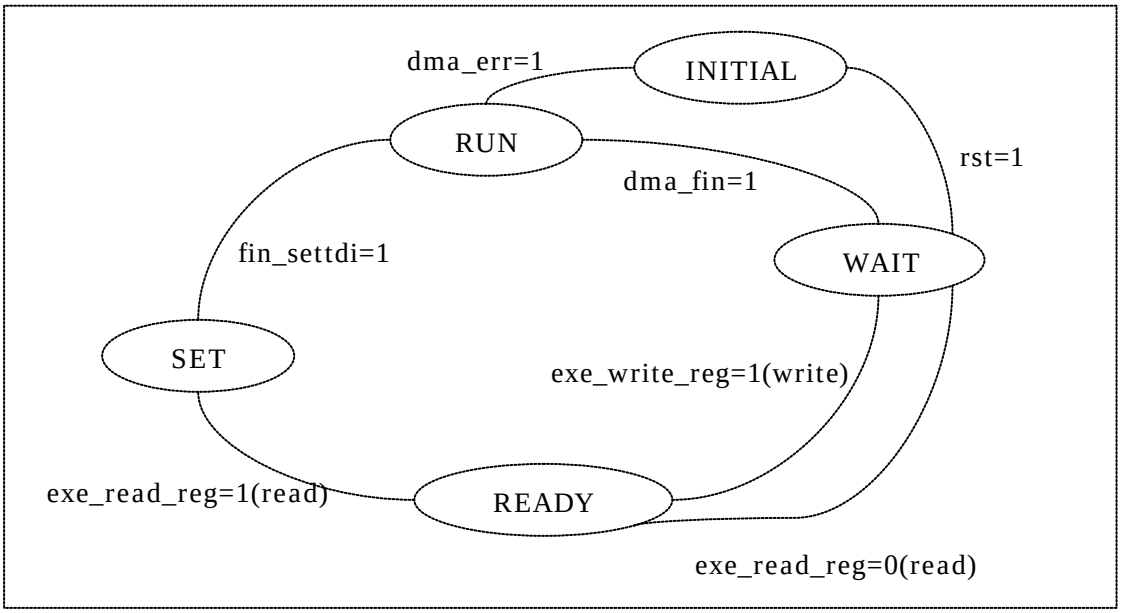

図 14：DMA Controller の状態遷移

DMA Controller は図 14 に示す 5 つの状態を持つ。まずシステムが起動されると初期状態 INITIAL に設定される。そして外部からリセット信号が入力されると WAIT 状態となる。WAIT 状態は Board Sequencer からの接続を許可する状態である。そして Board Sequencer からアクセスを受信すると READY 状態に移動し、転送設定を許可する設定可能状態 SET となる。データ転送中は RUN 状態となり、転送が正常に終了すれば WAIT 状態へ移行し、エラーが発生すれば INITIAL 状態へと以降する。

- tdi_rec

Board Sequencer との接続が確立した後、Board Sequencer から TDI が送信されると、tdi_rec モジュール内の 64 ビットレジスタへ保存される。Board Sequencer からの TDI の受信は、システムバスのデータバスが使用される。このデータバスは 16 ビットバスであるため、4 回に分けて TDI を受信する。

3.3.3 BUS Controller

BUS Controller は FPGA 内部のシステムバス(24 ビットアドレスバス、16 ビットデータバス)と、FPGA 外部の FlashRAM バス(24 ビットアドレスバス、16 ビットデータバス)を接続するブリッジの役割を担う。また、DMA 転送時に使用されるシステムアドレス空間から、物理アドレスへ変換するアドレスデコーダの機能を持つ。FPGA 内部モジュールのデータバス、及びアドレスバスは全てこの BUS Controller に接続され、Board Sequencer からの情報により適切なバスの振り分けを行う。図 15 に BUS Controller の内部モジュール構成を示す。

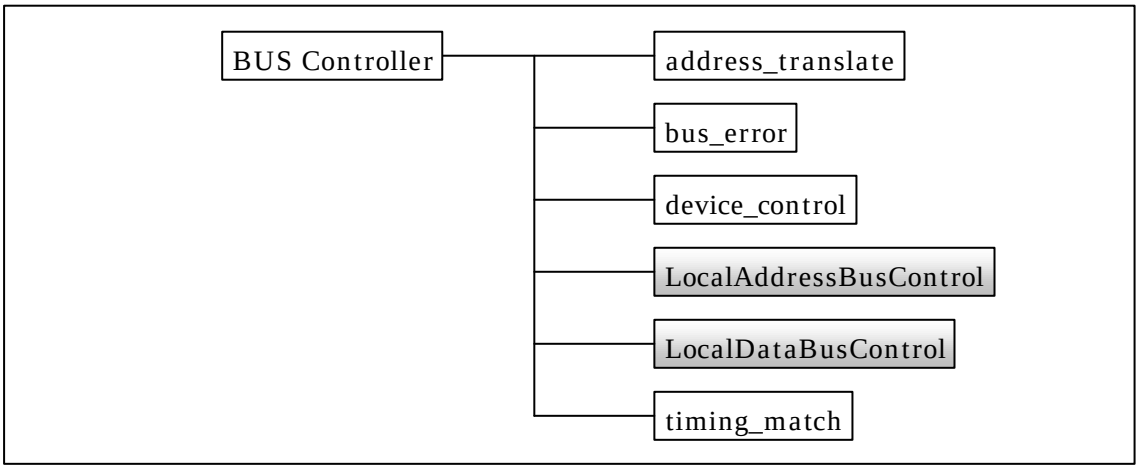

図 15：BUS Controller モジュール構成

図 15 に示す通り、BUS Controller は 6 モジュールから構成される。

- addres_translate

address_translate は、DMA 転送時に使用される図 6 に示したシステムアドレス空間から、実際の物理アドレスへと変換を行うアドレスデコーダモジュールである。実際に変換

対象となるのは、命令メモリを構成する FlashRAM-命令メモリ間の DMA 転送の場合だけである。

- bus_error

DMA Controller から受信する転送元アドレス､及び転送先アドレスに問題はないか(規定範囲に収まっているか)を判断し、エラーを検知するとエラー信号を Board Sequencer に対して送信する。

- device_control

device_control モジュールは、システムバスに接続されている命令メモリ、及びデータメモリと、外部バスに接続されている FlashRAM に対しての制御信号を管理するモジュールである。

- LocalAddressBusControl

LocalAddressBusControl モジュールは、内部アドレスバスと外部アドレスバスを接続、管理するモジュールである。図 16 にアドレスバス共有のブロック図を示す。

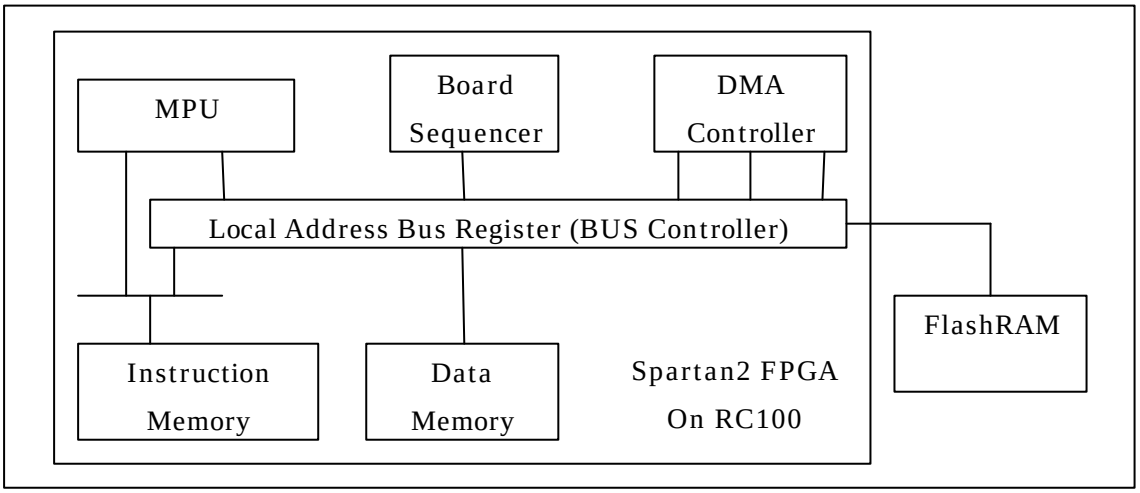

図 16：アドレスバスの共有

アドレスをラッチ回路に記憶し、Board Sequencer からのバス使用権情報に基づき各モジュールへ分配する。バスを共有というより、ラッチ回路を共有といったイメージである。

- LocalDataBusControl

LocalDataBusControl モジュールは、内部データバスと外部データバスを接続、管理するモジュールである。図 17 にデータバス共有のブロック図を示す。

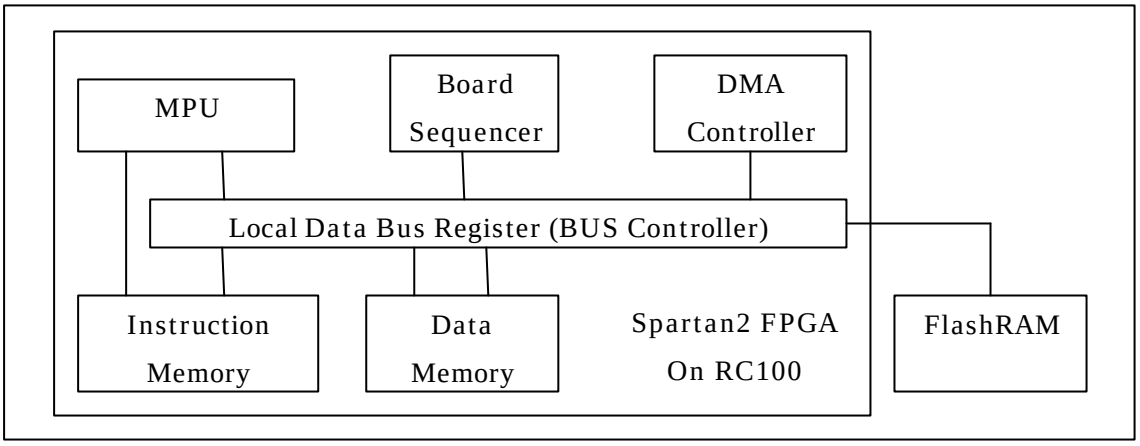

図 17：データバスの共有

アドレスバス同様、データをラッチ回路に記憶し、Board Sequencer **からのバス使用権情報に基づき各モジュールへ分配する。**

- timing_match

BUC Controller**はシステムアドレスを物理アドレスに変換したのち各モジュールに変換するため、デコードに** 1 **クロック必要となる。そこで** timing_match **モジュールによって同期を取り、適切なタイミングでアドレスとデータを出力する。**

3.3.4 命令メモリ･データメモリ

本システムでは MPU **が使用するメモリとして、**SpartanⅡFPGA **上の** BlockRAM **を使用する。この** RAM **は** Xilinx **社の** CAD **ツール** Core Generator **を用いて生成する** IP **コアであり、命令メモリ、データメモリともに** Single Port Block RAM **を生成する。**Xilinx SpartanⅡFPGA(**部品番号:**XC2S200-5-FG456)**の最大容量は** 7KB **であり、本システムではアドレスデコードの簡易化から命令メモリ、データメモリ共に** 2KB **を割り当てる。**

3.3.5 クロックの分周

RC100 **ボードには** 80MHz **のクロック発生器が搭載されている。しかし、本システムでは** FlashRAM **への読み書きがあることから、**80MHz **のままでは動作することは難しい。そこでクロック信号を** FPGA **内部で** 10MHz **と** 5MHz **に分周し、各機能モジュールへと渡している。図** 18 **にクロック** DLL **によるクロック分周を示す。**

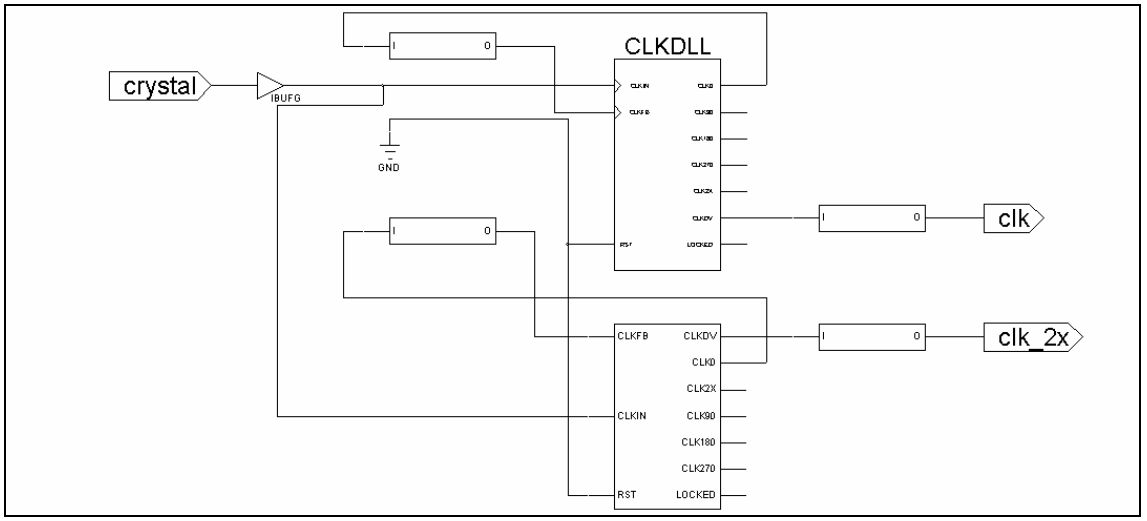

図 18：クロック DLL によるクロックの分周

図 18 **において、**crystal **信号が** 80MHz **のクロックソースである。その信号を** 2 **つの** CLKDLL **モジュールに通すことで** 5MHz **の** clk **信号と、**10MHz **の** clk_2x **信号に分周する。**

3.4　16 ビット命令セット MONI

MONI **とは本研究室で、教育をターゲットに定義した** MIPS **のサブセットに当たる命令セットであり、**16bit **固定命令語長の全** 43 **命令が用意されている**[28][29]**。それぞれの命令語のビットフィールドはそのビット配置から** 4 **種類の命令形式に分類される。**

- R **形式**

5	3	3	3	2
Opecode	Rs	Rt	Rd	Function

- I5 **形式**

5	3	3	5
Opecode	Rs	Rt	Immediate/Address

- I8 **形式**

5	3	8
Opecode	Rs	Immediate/Address

- J **形式**

5	11
Opecode	Target absolute address

命令の上位 5bit **は全ての命令において共通であり、命令の動作内容を示す命令操作コード**(OpeCode)**が配置されている。その命令操作コードによってそれ以下のビット配置が決まり、**R **形式、**I5 **形式、**I8 **形式、**J **形式の** 4 **種類に分類される。**Rs**、**Rt **はソース・レジスタを表し、演算などの入力データに使われるレジスタ番号を示す。**Rd **はディスティネーショ**

ン・レジスタを表し、演算などの結果を格納するレジスタ番号を示す。I5 形式、I8 形式における Immediate / Address は即値またはアドレスを意味し、数値を 2 進化したものが入る。J 形式の Target absolute address にはジャンプ先の絶対アドレスが入る。また R 形式における Function は OpeCode とは別に更に細かな動作内容を示すための機能を果たす。より詳しい OpeCode や Function コードの対応は参考文献[31]に示す。

3.4.1 命令セット定義における設計方針

コラーニングシステムにおけるソフトウェア学習において、アセンブリプログラムを行うことから、プログラミングのし易さを考慮し、レジスタ間演算には 3 オペランド方式を採用した。そして、ユーザが使用可能な汎用レジスタ数を出来るかぎり確保するため、データメモリ上にスタック領域を設定し、POP、PUSH 命令によりデータの退避を可能とした。また、サブルーチンからの戻り番地に関しても、レジスタを使用するのではなく、スタック領域に退避するものとした。以下にアセンブリプログラミング例を示す。

```
        POP     $4                  //$4 をスタックから POP
        CALL    WORKZ               //サブルーチン WORKZ へ分岐
        PUSH    $7                  //$7 をスタックへ PUSH
WORKZ:  ADD     $3      $2      $1  //$3＝$2＋$1
        RETURN                      //サブルーチンから復帰
```

サブルーチンからの戻り番地を、データメモリ上のスタックに割り当てることで、多くのサブルーチンコールを実現可能とした。しかし、サブルーチンからの復帰の際にデータメモリからプログラムカウンタを読みだす必要があるため、逆にこの経路が MPU のクリティカルパスとなることは確実である。この MONI 命令セットの改良に関しては、今後の課題として第 5 章にて述べる。

3.4.2 MONI 命令セットに基づくアセンブリプログラミング

MONI 命令セットを用いて数種類のアセンブリプログラムを作成した。以下に作成したアセンブリプログラムとその行数を示す。

表 3：MONI アセンブリプログラムの作成とプログラム行数

アセンブリプログラム	行数
フィボナッチ数列の第 N 項の算出	74
N 個の中の最大値の算出	35
2 数における最大公約数の算出	24
バブルソート	77
挿入ソート	78
選択ソート	117
シェルソート	114

表 3 のプログラムを大八木氏が開発した MONI 命令セットシミュレータを用いて、プログラム解析を行った。図 19 にシェルソートを要素数 100 として実行した際のプログラム解析結果を示す。

図 19：シェルソートを要素数 100 として実行した際のプログラム解析

命令セットシミュレータのプログラム解析機能を用いると、図 19 に示すようなダイアロ

グにより、命令形式ごとの実行回数や分岐回数などを得ることができる。図 19 はプロセッサアーキテクチャをパイプラインとして実行した際の結果であり、解析機能には、パイプラインハザードの回数(ストール、フォワーディング)や CPI を知ることができ、パフォーマンスデバッガとして、プログラムの修正も可能である。このシミュレータを用いることで MONI 命令セットの特徴とプロセッサアーキテクチャを理解することができる。

3.5 MONI 命令セットにおけるプロセッサの設計

コラーニングシステムは、プロセッサアーキテクチャの理解が最大の目標である。命令レベル並列処理技法におけるスーパースカラアーキテクチャを最終目標とし、その過程における単一サイクルアーキテクチャ、マルチサイクルアーキテクチャ、パイプラインアーキテクチャを用意することで、体系的な理解を目指す。そこで本研究では MONI 命令セットによる単一サイクルプロセッサの設計を行う。マルチサイクルプロセッサ、パイプラインプロセッサの設計は同研究室の大八木氏が設計を担当した[33]。パイプラインアーキテクチャは基本的な 5 段パイプラインである。スーパースカラプロセッサの設計は今後の課題とするが、実装アーキテクチャは In-Order 完了の 3 命令同時発行を予定している。

3.5.1 単一サイクルアーキテクチャ

単一サイクルアーキテクチャとは、1 命令を 1 クロックで実行するプロセッサアーキテクチャである。コラーニングシステムのプロセッサアーキテクチャ理解における入門編に位置する、最も簡単なアーキテクチャである。図 20 に単一サイクル MPU のメモリアクセスを示す。

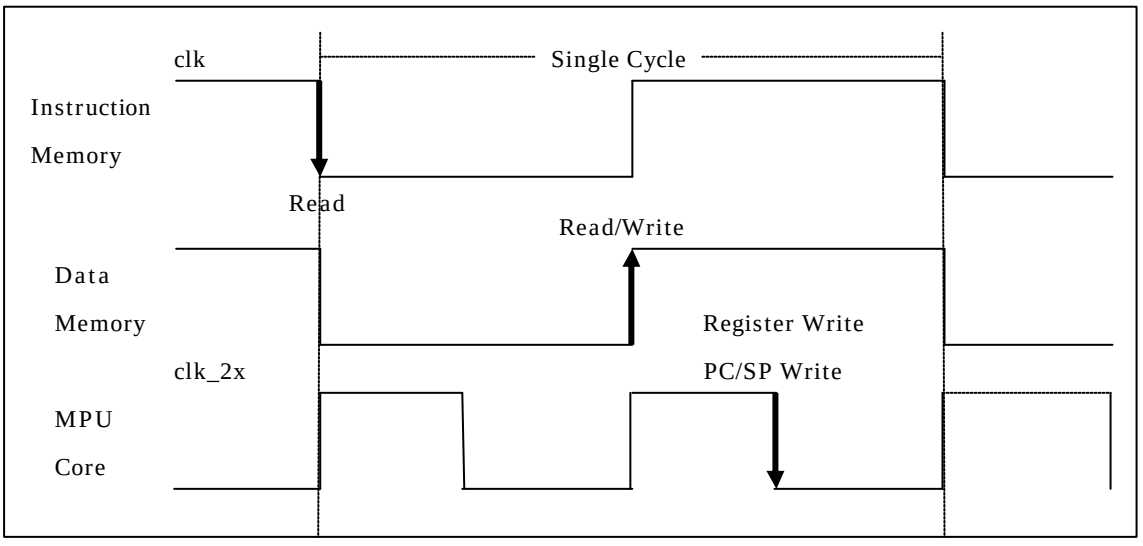

図 20：単一サイクルアーキテクチャのメモリアクセス

- 単一サイクル MPU の動作

単一サイクル MPU は 2 相クロックによって制御される。メモリアクセスは 5MHz の clk 信号によって行われ、立ち下りエッジで命令メモリから命令を読み出し、ロード命令、及びストア命令の際は立ち上がりエッジでデータメモリの読み書きを行う。全ての命令が clk 信号 1 クロックサイクルで完了する。MPU 内部には 16 ビット×8 個の汎用レジスタファイル、命令メモリのアドレスを保持するプログラムカウンタ、スタック番地を保持するスタックポインタの 3 種類のレジスタが存在する。それらは共に 10MHz の clk_2x 信号の立下りで、且つ clk 信号が 1 の時に書き込みを行う。

- スタックポインタの初期化

単一サイクルプロセッサは、データメモリ上にスタック領域を確保する。2KB の容量を持つデータメモリの 0x0003FF(=1023)番地をスタックポインタの初期値とする。

4. FPGA ボードコンピュータの実装

第 3 章に示した FPGA ボードコンピュータを、VerilogHDL を用いて設計し、FPGA へ実装した。本章では、本研究室における開発環境、開発期間、デバッグ手法を述べた後、機能モジュールとシステムの実装規模を示す。

4.1 立命館大学高性能計算研究室における開発環境

本研究では Xilinx の EDA ツール FoundationISE5.2isp3 を用いて設計を行った。FoundationISE は Xilinx 社製 FPGA、及び CPLD に特化した設計統合開発環境ツールであり、機能設計、シミュレーション、論理合成、配置配線、そしてビットストリームの生成までを 1 つのウィンドウ上で管理できる。

4.2 実装フロー

Board Sequencer 内の dma_connection_control モジュールを例にとり、システム全体の実装フローを示す。

4.2.1 VerilogHDL による設計

VerilogHDL による一般的なモジュール設計の例を図 21 に示す。

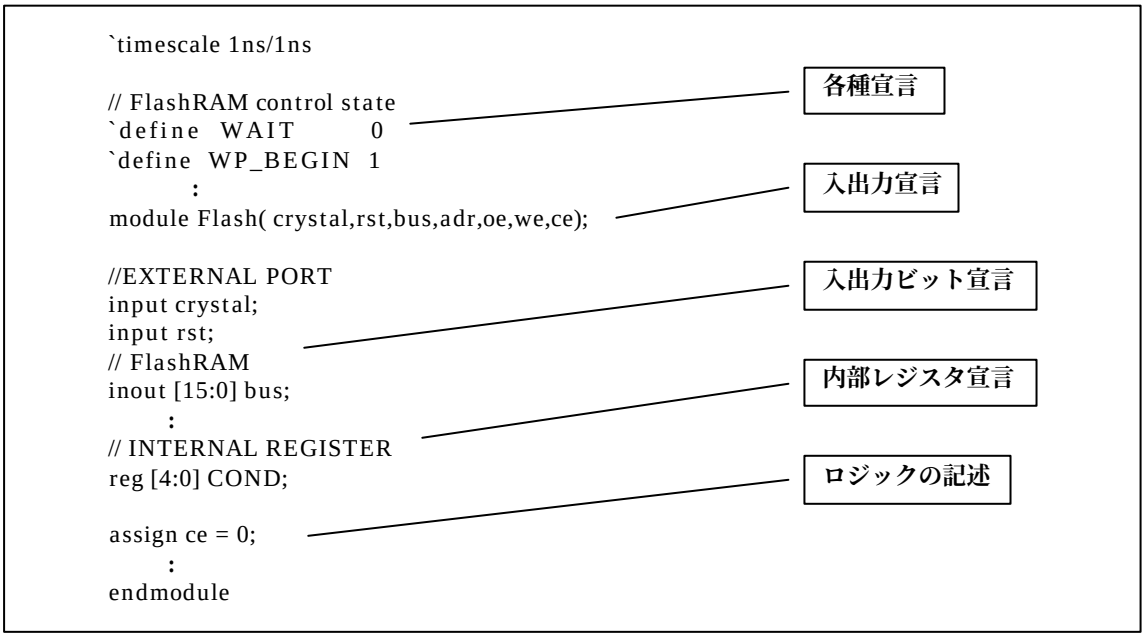

図 21：VerilogHDL による RTL 設計例

図 21 は FlashRAM 制御モジュールの一部である。VerilogHDL ではまず、ファイルの先頭に各種宣言を行う。次にモジュール名と入出力信号の宣言を行い、それらの入出力信号のビット幅を定義する。内部レジスタを定義する際は reg 宣言にてビット幅を指定する。

そしてロジック本体の記述を開始する。

付録に FlashRAM 制御モジュールのソースを載せる。

4.2.2 単体モジュールの実装

FPGA ボードコンピュータの構成モジュールである Board Sequencer は図 10 に示す 7 モジュールによって構成される。まず、7 モジュールそれぞれにおいてマッピングレベルシミュレーションを完了させる。図 22 に単体モジュールの実装フローを示す。

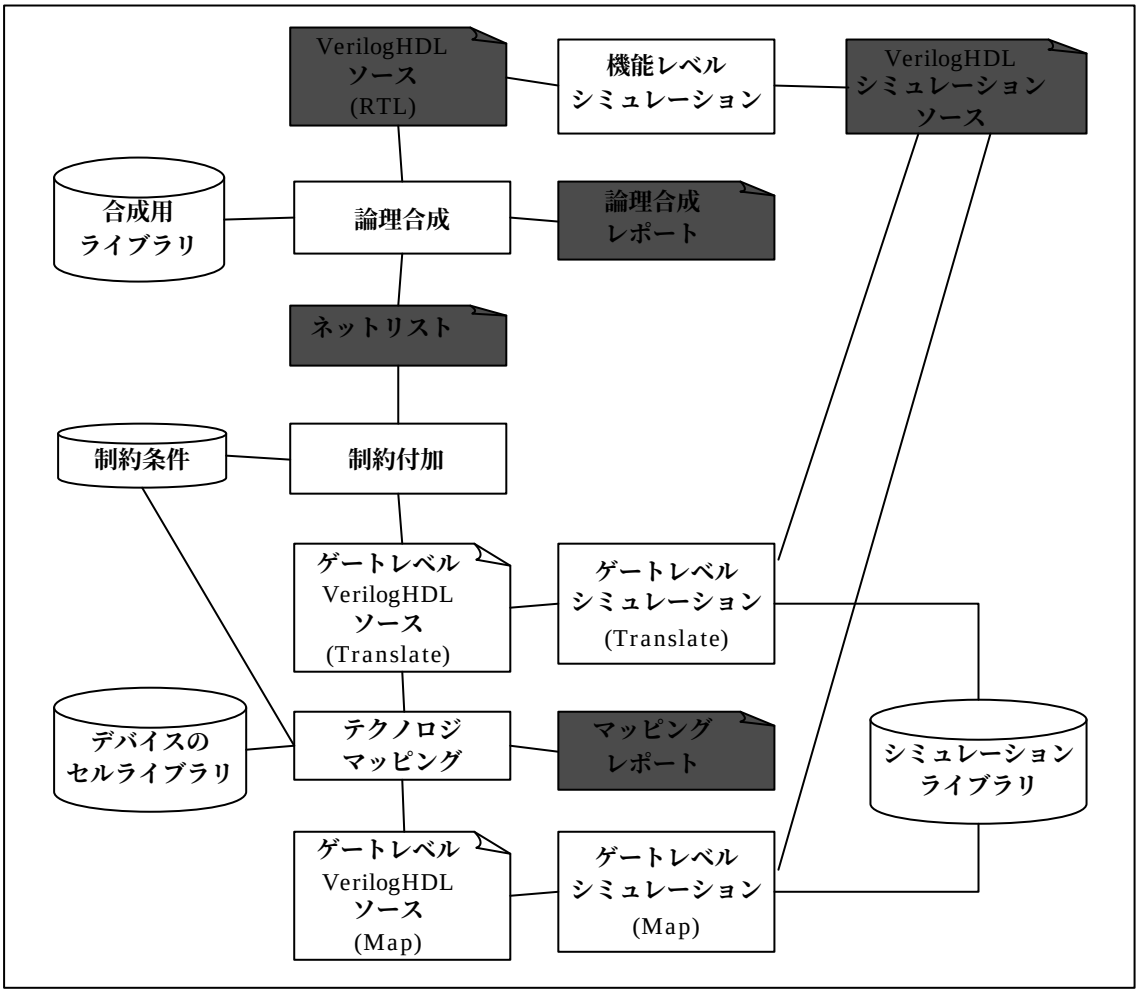

図 22：単体モジュールの実装フロー

まず、VerilogHDL を用いて RTL 設計を行う。そして、VerilogHDL にて別途作成したテストベンチを用いて機能レベルシミュレーションを行う。ここでは仕様通りの動作が得られているかを確認する。正常動作が得られれば論理合成にかける。論理合成レポートを確認し、ゲート規模や動作周波数を確認する。最適化が必要であれば再び RTL ソースを修正する。

論理合成によってネットリストが出力される。このネットリストをユーザが定義した制

約ファイル(UCF)を用いて、制約条件を付加する。その結果 Translate レベルの VerilogHDL が出力される。

Translate レベル VerilogHDL を用いて、ゲートレベルシミュレーションを行う。ここでは主に、RTL 記述が正しくゲートに変換できているかを確認する。仮に、制御線の初期化が行われておらず不定値が流れる場合、このゲートレベルシミュレーションでは、正しく動作しない。この段階は FPGA のテクノロジに依存せず、AND や XOR などのロジックレベルでのシミュレーションである。

続いてテクノロジマッピングを行う。ここで対象とする FPGA の資源へと割り付けられる。FPGA のアーキテクチャが異なると、マッピング結果も異なる。本システムでは Spartan2FPGA のアーキテクチャへと変換される。図 23 に Board Sequencer 内の dma_connection_control モジュールのマッピングレポートの一部を示す。

```
Target Device  : x2s200
Target Package : fg456
Target Speed   : -5
Mapper Version : spartan2 -- $Revision: 1.4 $
Mapped Date    : THU 12 FEB 23:17:39 2004
Design Summary
--------------
Number of errors:              0
Number of warnings:            0
Logic Utilization:
Number of Slice Flip Flops:     24 out of 4,704    1%
Number of 4 input LUTs:         98 out of 4,704    2%
Logic Distribution:
Number of occupied Slices:      55 out of 2,352    2%
Number of Slices containing only related logic: 55 out of 55     100%
Number of Slices containing unrelated logic:    0 out of 55       0%
*See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:         104 out of 4,704          2%
     Number used as logic:                  98
     Number used as a route-thru:            6
   Number of bonded IOBs:          33 out of 284   11%
   Number of GCLKs:                 1 out of 4      25%
   Number of GCLKIOBs:              1 out of 4      25%
```

図 23：dma_connection_control モジュールのマッピングレポート

マッピングレポートでは、論理合成レポートよりも正確な規模見積もりが出力される。dma_connection_control はマッピングレポートより、フリップフロップ数は全要素 4,704 個中 24 個、ロジックである 4 入力 LUT 数は全要素 4,704 個中 98 個である。マッピングによって出力される Map レベル VerilogHDL を用いてマッピングレベルシミュレーションを行う。単体テストでは、配置配線にかける意味がないため、シミュレーションが完了すれ

ば、単体モジュールとしての設計を完了とする。

Board Sequencer 内の 7 モジュールにおいて、マッピングレベルシミュレーションを終えると、Board Sequencer のトップモジュールにおいて、モジュールの結合を行う。そして同様にマッピングレベルシミュレーションまで行い、Board Sequencer 単体の実装を完了させる。

本システムでは、Board Sequencer 以外の機能モジュールとして DMA Controller、BUS Controller、MPU、命令メモリ、データメモリがある。それぞれの機能モジュールにおいても同様にマッピングレベルでの動作確認を終える。

4.2.3 機能モジュールの統合と実装

各々の機能モジュールにおいて、マッピングレベルでの動作が保証できると、次に Board Computer として、システムの統合を行う。システム全体の実装フローを図 24 に示す。

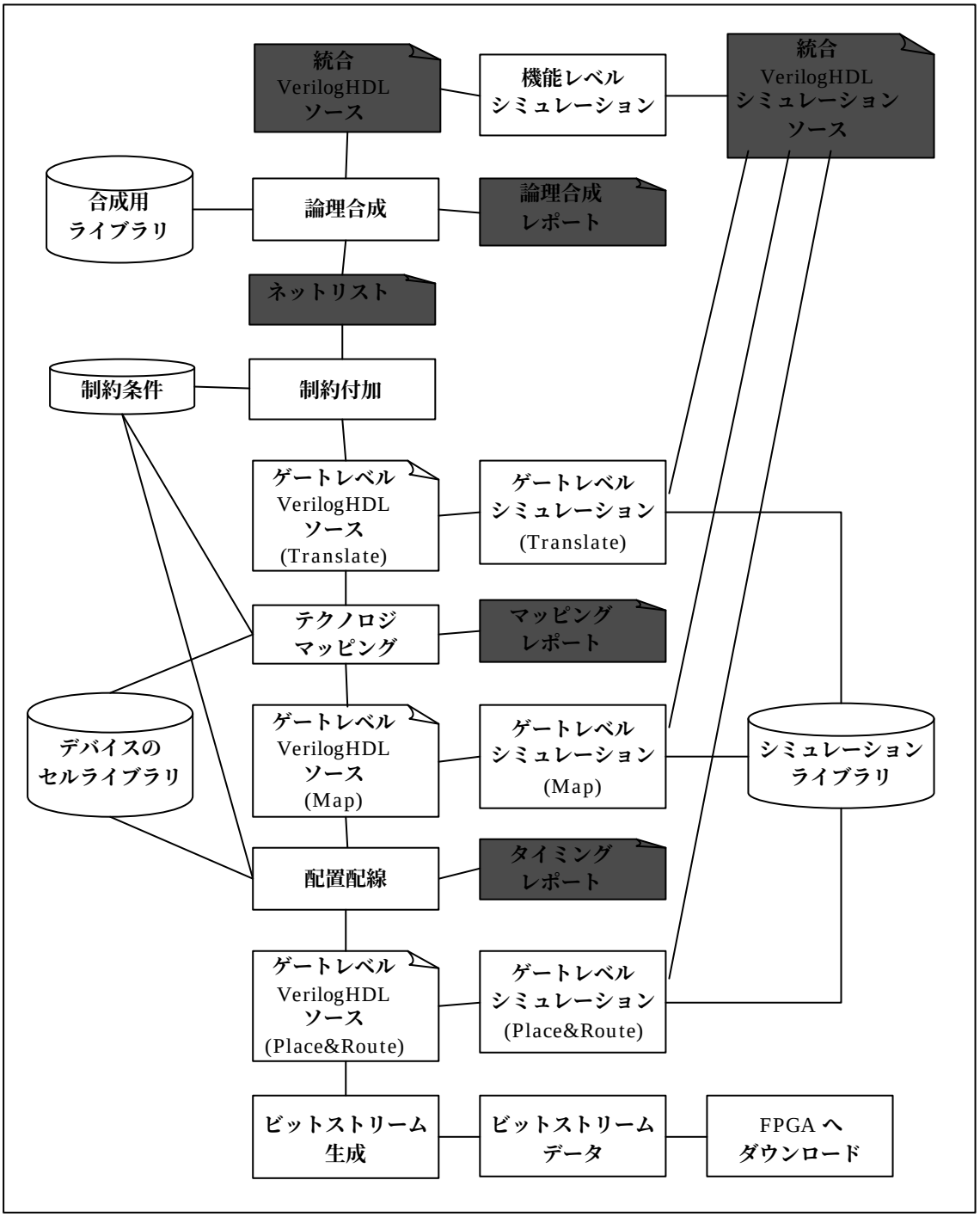

図 24：システム全体の実装フロー

マッピングレベルまでは、単体モジュールと同様のテスト手順を踏む。機能モジュール単体の動作は保証されているため、モジュール間のインタフェースとタイミング同期に関しての検証を行う。システムレベルになると、配置配線を行い、FPGA 内の配線遅延を見積もる。出力されるタイミングレポートを確認し、タイミング違反がないことを確認する。

違反がなければ、配置配線によって出力された配置配線レベル VerilogHDL を用いて最後のシミュレーションを行う。

配置配線シミュレーションを正常に終えると、FPGA の構成データであるビットストリームの生成を行う。そして出力されるビットストリームデータを用いて FPGA へダウンロードを行い、システムを実装する。

4.3 開発期間

本システムの開発期間を表 4 に示す。

表 4：FPGA ボードコンピュータ開発期間

ステップ	開発期間(単位=日)
仕様の策定	20
仕様書の作成	20
コーディングと単体テスト	15
統合テスト	10
実機検証とデバッグ	25

FPGA ボードコンピュータの仕様の策定に始まり、まず仕様書を作成した後、コーディングを開始した。機能モジュールのコーディングと、単体テストにはあまり時間を要さなかった。本システムをデバッグする上での問題点は、RC100 上の FlashRAM をどれだけの精度でシミュレーションできるかがポイントであった。仮想 FlashRAM を VerilogHDL によって模擬することは可能であるが、FlashRAM 内の WSM(Write State Machine)は記述することができないため、正確なシミュレーションが行えない。そのため、システム全体の正確なシミュレーションを行うことができないまま実機検証へ移るしかなかった。次節において我々が行った RC100 ボード上でのデバッグ手法を述べる。

4.4 デバッグ手法

専用ハードウェアの設計において、FPGA はプロトタイプとしての動作検証に用いられることが多い。その際は、入力に対する出力が期待値と一致するかを確認するだけでよい。しかし、本研究では RC100 ボード全体で 1 つのシステムを構築しているため、FPGA 上での実機検証がそのままシステム検証になる。今回、FlashRAM の制御が課題であったため、まず FlashRAM の制御コントローラのみを FPGA に実装し、動作検証を行った。そしてコントローラの動作確認を終えた後、ボードコンピュータ内の DMA Controller モジュールに FlashRAM コントローラを組み込みシステムの動作検証を行った。

システム全体のデバッグは、RC100 ボード上の 7 セグメントディスプレイを用いて行った。具体的には、確認したいデータや線を Board Sequencer 内の Display Control モジュールに引っ張り出し、7 セグメントディスプレイに表示する。このデバッグのために、

Display Control モジュール内に 4 ビットレジスタを 200 個ほど用意し、外部スイッチから制御を与えてやることで、レジスタの中身を順次表示していく。図 25 にディスプレイへの表示方法を示す。

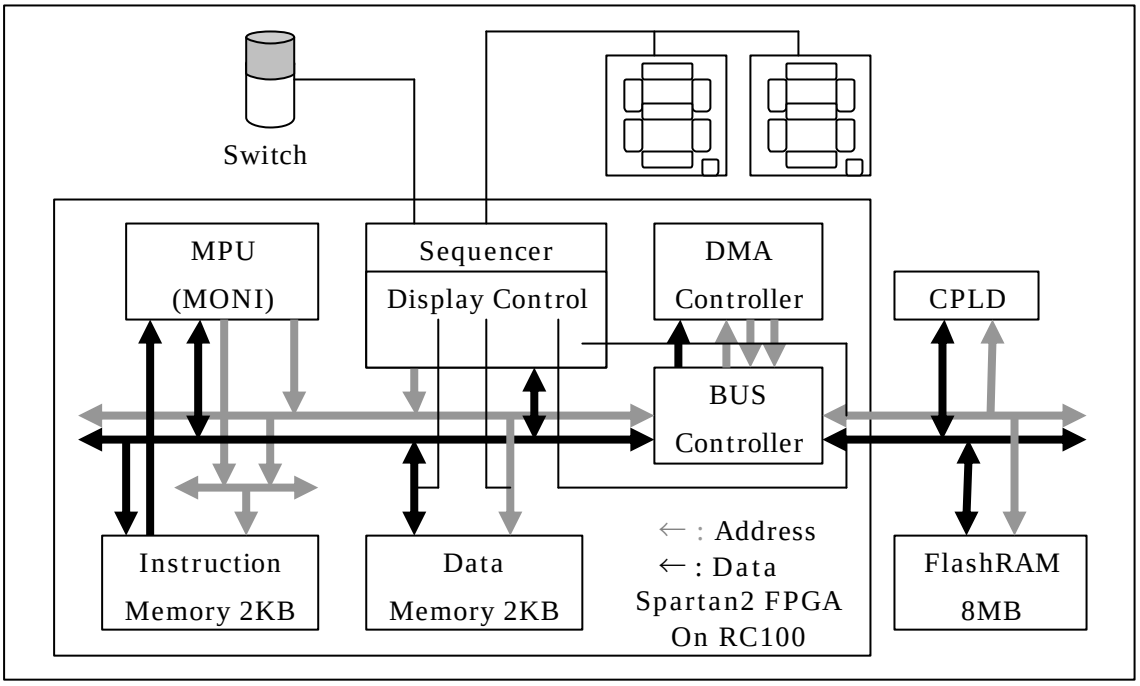

図 25：7 セグメントディスプレイを用いたデバッグ

図 25 では DMA 転送において、FlashRAM からデータメモリへと確実にデータが書かれているかを検査する場合を表している。

4.5 機能モジュールの実装規模と動作周波数

主要機能モジュール(図 5 参照)における単体での実装規模、動作周波数を表 5 に示す。ここで、MPU アーキテクチャは単一サイクルである。

表 5：主要機能モジュールの実装規模と動作周波数

	Registers	LUTs	System Gates	動作周波数(MHz)
Board Sequencer	257	495	30,000	57.683
DMA Controller	209	751	38,000	43.833
BUS Controller	94	370	16,000	24.752
MPU	161	915	44,000	37.883

表 5 において Registers は、モジュール実装に要した記憶素子の数(FF と Latch 回路の合計)、LUTs は組み合わせロジックの数、そして System Gates は、Registers と LUTs の合計をシステムゲート換算した際の数であり、実装規模に等しい。動作周波数はモジュー

ルの最大動作周波数で単位は MHz である。

実装規模と動作周波数の関係は、各モジュールの機能に相応しい結果であった。Board Sequencer は TDI(64 ビットレジスタ)を 3 個持つため、レジスタ数は 4 モジュール中最大であった。しかし、あまり複雑なロジックを持たないため動作周波数は最大である。一方、BUS Controller はレジスタ数、ロジック数ともに最小であるが、動作周波数は最も低い。これは、各モジュールからのバス信号が集中するため、遅延が大きくなったことが原因である。本システムの要求動作周波数は、FlashRAM の書き込みに依存し、10MHz であるため、個々のモジュールにおいて要求を満たしていることを確認できる。

4.6　ボードコンピュータシステムの実装規模と動作周波数

ボードコンピュータの機能モジュールを結合し、システムとして動作させた際の実装規模と動作周波数を表 6 に示す(MPU は単一サイクルアーキテクチャ)。

表 6：ボードコンピュータシステムの実装規模と動作周波数

	Registers	LUTs	System Gates	使用率(%)	動作周波数(MHz)
BC System	819	2,281	130,000	62	21,030

表 6 において使用率(%)は、SpartanⅡFPGA におけるリソース使用率である。MPU コアを単一サイクルアーキテクチャとして実装した際、62%のリソースを使用したことになる。動作周波数も 21,030MHz であることから、FlashRAM の要求動作周波数を満たしている。

4.7　実機検証

MONI 命令セットの MPU コアとそのソフトウェアを動作させることで FPGA ボードコンピュータの実機検証を行った。実機検証に用いたプログラムはに示したものである。以下に実機検証手法を述べる。

4.7.1　実行プログラムの作成と FPGA へのダウンロード方法

本システムにおいて、プログラムを実行するには FlashRAM の特定番地にデータや命令列などを格納しておく必要がある。図 26 に実行プログラムの生成と FlashRAM の格納先を示す。

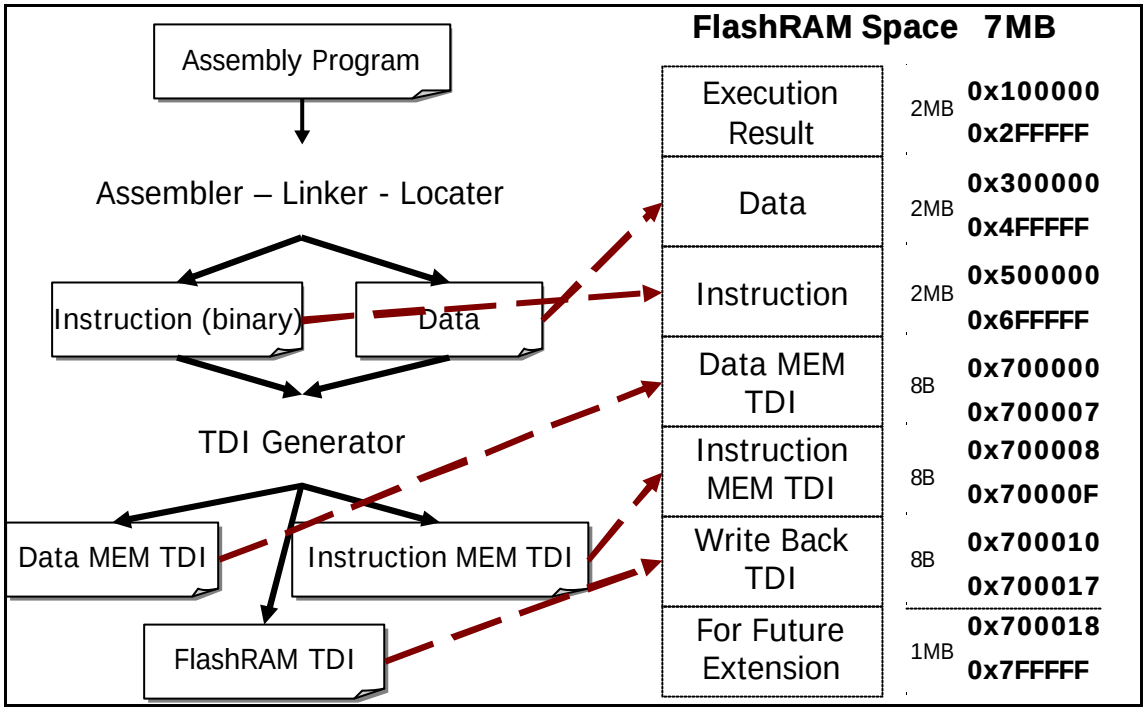

図 26：実行プログラムの生成と FlashRAM 格納先

1. **実行プログラムの生成**

設計したアセンブリプログラムをアセンブラに通し、命令群とデータを生成する。次に TDI Generator **を起動し、**3 **個の** TDI **を作成する。それらを** RC100 **ボードに付属のファイル転送ツール** FTU2 **を用いて** FlashRAM **の特定番地に格納する(データは** 0x300000~**、命令は** 0x500000~**、**TDI **は** 0x700000~**)。**

2. FPGA **コンフィギュレーション**

FoundationISE **を用いて** FPGA **構成データである** BIT **ストリームファイルを生成する。そして生成されたファイルを、**FTU2 **を用いてホスト** PC **から** RC100 **ボード上の** FPGA **へダウンロードする。ダウンロードが終了すると、ボードコンピュータシステムは初期状態(図** 7 **参照)となる。そして外部からリセット信号を与えることで、システムが起動する。**MPU **が演算を終えると、**DMA **転送により** FlashRAM **の** 0x100000 **番地に結果が格納される。**

3. **結果の確認**

FTU2 **を起動し、**FlashRAM **の** 0x100000 **番地からの値をホスト** PC **上のファイルに書き出し、結果を確認する。**

図 7 **に示したシーケンスが、どこまで進んだのかを** 7 **セグメントディスプレイを用いて表示させ、正常に動作しない場合は** 4.4 **節に示した方法により、異常個所を特定し、デバッグを行った。**

5. FPGA ボードコンピュータシステムの評価

FPGA ボードコンピュータシステムを RC100 ボード上に実装し、正常な動作を確認することができた。しかし、現在のシステムでは幾つかの制約が存在するため、汎用性が高いシステムとは言えない。システムの改良に向けた今後の課題を以下に示す。

① システムのマルチサイクル実行

② システムアドレス空間の柔軟な変更

③ マルチ MPU コア搭載

現在のシステムでは図 7 における初期状態から FlashRAM に結果を書き戻しが完了するまでの 1 サイクルを終えると、初期状態に戻り動作を停止する制御となっている。このため、命令メモリが 2KB とデータメモリが 2KB という制限の上でのみ、プログラムを実行可能である。この制限を破るための一手法として、DMA 転送時に使用する TDI を拡張し、次の TDI を指し示すポインタ領域を付加することが考えられる。図 27 にその例を示す。

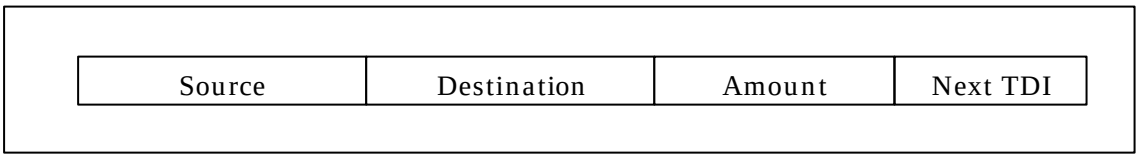

図 27：TDI の拡張

図 27 の構造を持つ TDI を用いることで、システムのマルチサイクル実行が可能となる。命令メモリ構成 TDI、データメモリ構成用 TDI、結果の書き戻し用 TDI それぞれに NextTDI 領域を付加し、次サイクル時の TDI アドレスを記憶する。それぞれの NextTDI が NULL となった場合、そのサイクルでシステムを停止する。このようにすることで、BlockRAM 領域 2KB の制限を破ることができ、簡単な仮想記憶による命令の実行が可能となる。

現在のシステムは図 6 に示すシステムアドレス空間で固定となっている。これでは、システムに汎用性があるとは言えない。そこでシステムアドレス空間を定義するテンプレートを用意することで、マルチプラットフォームに対応できるようなシステムの改良が考えられる。RC100 ボードには、FlashRAM 以外の記憶素子として 1MB の SSRAM が搭載されているため、今後システムを拡張する際は、汎用性を高めることを考慮に入れた改良を行ってもらいたい。更にモジュール間の構成要素を自由に定義できるようになれば、FPGA 内に複数の MPU コアを組み込み、専用ハードウェアをメモリマップドした上でシステムを合成することも可能となる。

6. おわりに

本研究では、現在のシステム LSI 設計に必要な技術を考察し、ハードウェアとソフトウェアの協調学習システムであるハード/ソフト・コラーニングシステムを提案し、設計した。また、学習システムの教材として用いられる FPGA ボードコンピュータを考案し、Celoxica(株)の RC100 ボード上に実装した。

今後の展望として、FPGA ボードコンピュータの第 5 章に述べた改良を行い、命令セットシミュレータと連携を密にすることで、提案だけでは終わらない、実際の教育現場で使用可能なシステムに発展させる。

ユビキタスという流行語が生じているように、コンピュータを中心とする VLSI はあらゆる産業基盤に対して急速に浸透しつつある。そして今後の日本の産業界の発展において VLSI 技術の一層の高度化と振興を図ることが必要不可欠である。2004 年 4 月に開設される立命館大学理工学部の新学科である電子情報デザイン学科は、このような産業界からの強い要請と VLSI 技術利用の社会的動向を踏まえ、人材育成、新たな研究開発の展開、産官学連携、技術者交流の促進等に寄与する目的で開設され、本研究室も配属されることとなった。今後は本研究室の学生が、並列処理とハード/ソフト・コデザインを柱として、より活発な研究活動が行われることを願って止まない。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導をいただきました山崎勝弘教授、小柳滋教授に深く感謝いたします。

また、本研究の共同研究者である中村浩一郎氏、大八木睦氏、本研究に関して貴重なご意見をいただきました Tran So Cong 氏、EDA ツールに関する数々の質問に答えて頂いた東京エレクトロンデバイス㈱の松原哲雄氏、及び色々な面で励ましを下さった高性能計算研究室の皆様に心より深く感謝いたします。

参考文献

[1] 三木良雄:システム LSI 設計特論講義資料,第 1 章システム LSI とは,立命館大学大学院スターク寄附講座,2003.

[2] 内山邦男:システム LSI 設計特論講義資料,第 3 章 LSI 構成要素Ⅰ,立命館大学大学院スターク寄附講座,2003.

[3] 三木良雄: システム LSI 設計特論講義資料,第 10 章設計事例と今後の課題,立命館大学大学院スターク寄附講座,2003.

[4] 鈴木敬:高位言語ベースデザイン設計特論講義資料,第 4 章アーキテクチャレベルコンポーネント生成技術 3,立命館大学大学院スターク寄附講座,2003.

[5] 今井正治:高位言語ベースデザイン設計特論講義資料,第 4 章アーキテクチャレベルコンポーネント生成技術 2,立命館大学大学院スターク寄附講座,2003.

[6] 若林一敏:高位言語ベースデザイン設計特論講義資料,第 4 章アーキテクチャレベルコンポーネント生成技術 1,立命館大学大学院スターク寄附講座,2003.

[7] 清尾克彦:高位言語ベースデザイン設計特論講義資料,第 1 章システムレベル設計手法の概要,立命館大学大学院スターク寄附講座,2003.

[8] 清尾克彦:高位言語ベースデザイン設計特論講義資料,第 7 章システム LSI の実際,立命館大学大学院スターク寄附講座,2003.

[9] 若林一敏:システム LSI 設計特論講義資料,第 5 章機能・論理検証,立命館大学大学院スターク寄附講座,2003.

[10] GONZALEZ R E: Xtensa: A Configurable and Extensible Processor, IEEE Micro ,Vol.20, No.2, PP60-64,65-70 ,2000.03.

[11] 東芝 Mep:http://www.mepcore.com/

[12] HOFFMANN A, KOGEL T, NOHL A, BRAUN G, SCHLIEBUSCH O, WAHLEN O, WIEFERINK A, and MEYR H :A Novel Methodology for the Design of Application-Specific Instruction-Set Processors(ASIPs) Using a Machine Description Language, IEEE Trans Computer-Aided Design Integrated Circuits System ,Vol.20, No.11, PP1338-1354 ,2001.11.

[13] Xtensa ベンチマーク:http://www.tensilica.com/html/eembc_optimized.html

[14] 立命館大学 MELPEC:http://www.ritsumei.ac.jp/se/Melpec/

[15] 西村, 額田, 天野:教育用パイプライン処理マイクロプロセッサ PICO｀2´の開発 電子情報通信学会技術研究報告,Vol.99, No.532(CPSY99 106-116), pp 61-68 ,2000.01.12.

[16] 高橋, 児島, 上土井, 吉田:マイクロコンピュータ設計教育環境 City-1 FPGA コンピュータの自由な設計と製作,情報処理学会研究報告, Vol.97, No.17(DA-83), pp 41-48 ,1997.02.14.

[17] 田中,久我,末吉,小羽田:教育用マイクロプロセッサ KITE とその開発支援環境,情報処理学会研究報告,Vol.93, No.49(ARC-100), pp.59-66 ,1993.06.

[18] 土江,佐々木,弘中,児島:教育研究用スーパースカラ・プロセッサ・シミュレータ Mikage の概要,情報処理学会研究報告, Vol.96, No.80(ARC-119), pp 107-112 , 1996.08.27.
[19] 井上, 中垣, 大内, 末吉:教育用 RISC 型マイクロプロセッサ DLX-FPGA とそのラピッドシステムプロトタイピング,電子情報通信学会技術研究報告,Vol.95, No.25(ICD95 11-22), pp.71-78 ,1995.04.28.
[20] 今井, 古川, 井面, 白木, 石川:計算機システム教育のためのビジュアルシミュレータ VisuSim,情報処理学会研究報告, Vol.2001, No.34-(CE-59), pp 77-84, 2001.03.23.
[21] 越智, 沢田, 岡田, 浜口, 上嶋, 神原, 安浦: 計算機工学・集積回路工学教育用マイクロプロセッサ KUE-CHIP2, 情報処理学会研究報告,Vol.92, No.82(ARC-96), pp 93-100 ,1992.10.22.
[22] 岩井原, 山家, 中川, 国貞, 斎藤, 永浦, 池兼, 中村, 安浦:教育用計算機 QP-DLX の開発と開発環境, 情報処理学会研究報告, Vol.93, No.111(ARC-103 DA-69), pp 95-102 ,1993.12.16.
[23] 桜井, 長沢, 宮内, 石川:教育用 RISC 型マイクロプロセッサ MITEC-II を用いた演習環境の開発及び MITEC-II を用いた演習の実施,情報処理学会研究報告 ,Vol.2001, No.101(CE-61), pp47-54 ,2001.10.19.
[24] 下川, 西野, 早川,システムソフトウェア教育支援環境「港」における FPGA を利用した演習環境の開発 電子情報通信学会技術研究報告,Vol.102, No.697(ET2002 95-119), pp7-12 ,2003.03.07.
[25] 原野,塩見:教育用マイクロプロセッサ SE4 を用いた設計演習の提案,情報処理学会全国大会講演論文集,Vol.65th, No.4,pp4.277-4.278,2003.03.25.
[26] 大八木,池田,山崎,小柳:ハード/ソフト・コラーニングシステムにおけるアーキテクチャ選択可能なプロセッサシミュレータの設計,情報処理学会 第 66 回全国大会論文集,2004.
[27] 池田,中村,大八木,Tuan,山崎,小柳:ハード/ソフト・コラーニングシステムにおける FPGA ボードコンピュータの設計,情報処理学会 第 66 回全国大会論文集,2004.
[28] 大八木, 池田, 山崎 : HDL による RISC プロセッサの設計経験(Ⅰ)-命令セットアーキテクチャと設計-, FIT2002, c-7, 2002.
[29] 池田,大八木,山崎:HDL による RISC プロセッサの設計経験(Ⅱ)-性能評価と考察-, FIT2002, c-8, 2002.
[30] 小林真輔:コンフィギャラブル・プロセッサを理解する-コンパイラ検証のための基礎知識,Design Wave, pp 65-72, 2003,12.
[31] 池田,中村,Tran:設計仕様書 品種名:RC100 を用いたボードコンピュータ,2004.
[32] 大八木睦:ハード/ソフト・コラーニング上でのアーキテクチャ可変なプロセッサシミュレータ(MONI シミュレータ)の使用法,2004/02/08.
[33] 大八木睦: ハード/ソフト・コラーニングシステム上でのアーキテクチャ選択可能なプ

ロセッサシミュレータの設計と試作,立命館大学大学院理工学研究科,修士論文,2004.
[34] 中村浩一郎: FPGA ボードコンピュータの開発,立命館大学理工学部情報学科,卒業論文,2004.

付録

・FlashRAM コントローラの VerilogHDL 記述

```
`timescale 1ns/1ns

//
// FlashRAM control state
`define    WAIT                    0
// word program
`define    WP_BEGIN                1
`define    WP_WRITE_40H            2
`define    WP_WRITE_DATA           3
`define    WP_WAIT_STS             4
`define    WP_READ_STS             5
`define    WP_CHECK_SR7            6
`define    WP_GET_SRD              7
`define    WP_WRITE_FFH            8
`define    WP_END                  9
// block erase
`define    BE_BEGIN                10
`define    BE_WRITE_20H            11
`define    BE_WRITE_CON            12
`define    BE_WAIT_STS             13
`define    BE_READ_STS             14
`define    BE_CHECK_SR7            15
`define    BE_GET_SRD              16
`define    BE_WRITE_FFH            17
`define    BE_END                  18
// clear lock bit
`define    CL_BEGIN                19
`define    CL_WRITE_60H            20
`define    CL_WRITE_CON            21
`define    CL_WAIT_STS             22
`define    CL_READ_STS             23
`define    CL_CHECK_SR7            24
`define    CL_GET_SRD              25
`define    CL_WRITE_FFH            26
`define    CL_END                  27

//
// FlashRAM WRITE DATA AND ADDRESS
`define    PD        16'h5678// PROGRAM DATA
`define PA           24'h000002        // PROGRAM ADDRESS
`define    WC        16'h40            // WORD COMMAND
`define SR7          7
// BLOCK ERASE ADDRESS
```

```
`define    BA        24'h000000        // BLOCK ADDRESS
`define    EC        16'h20            // ERASE COMMAND
`define    CON       16'hD0            // ERASE CONFIRM
// CLEAR LOCK BIT
`define    CC        16'h60            // CLEAR LOCK BIT COMMAND

module
FlashRAM(  crystal,rst,bus,adr,oe,we,ce,byte,sts,enable0,enable1,seg0,seg1,FP_COM,FP_PARP
ROT_MASTER );

        // EXTERNAL PORT
        input     crystal;
        input     rst;
        // FlashRAM
        inout     [15:0]     bus;
        output    [23:0]     adr;
        output    oe;
        output    we;
        output    ce;
        output    byte;
        input     sts;
        // 7 SEG DISPLAY
        output    enable0;
        output    [7:0]      seg0;
        output    enable1;
        output    [7:0]      seg1;
        // CPLD
        output    [2:0]      FP_COM;
        output    FP_PARPROT_MASTER;

        // REG
        reg       [15:0]     data;
        reg       [23:0]     adr;
        reg       we;
        reg       oe;
        reg       ce;
        reg       [7:0]      seg0;
        reg       [7:0]      seg1;

        // INTERNAL REGISTER
        reg       [4:0]      COND;

        //
        // CONSTANT SIGNAL
        // FlashRAM
        assign    byte = 1;          //
        // 7 SEG DISPLAY
        assign    enable0 = 1;       //
        assign    enable1 = 1;       //
        // CPLD
        assign    FP_COM = 7;
        assign    FP_PARPROT_MASTER = 1;
```

```
// INOUT CONTROL
assign     bus = (oe)? data: 16'hz;

// wire
wire crystal;
wire clkin,CLK0_W,clk0,clkdv,LOCKED,clk;
wire CLK0_W2,clk2,clkdv2,LOCKED2,clk_2x;

//
// CLKDLL            GENERATION
// clk
IBUFG IBUFG(.I(crystal),.O(clkin));

CLKDLL CLKDLL (.CLKIN(clkin), .CLKFB(CLK0_W), .RST(0),
.CLK0(clk0), .CLK90(), .CLK180(), .CLK270(),
.CLK2X(), .CLKDV(clkdv), .LOCKED(LOCKED));

// CLOCK FEED BACK
BUFG BUFG1 (.I(clk0), .O(CLK0_W));

// BUFG 2
BUFG BUFG2 (.I(clkdv), .O(clk));

// clk_2x
CLKDLL CLKDLL2 (.CLKIN(clkin), .CLKFB(CLK0_W2), .RST(0),
.CLK0(clk2), .CLK90(), .CLK180(), .CLK270(),
.CLK2X(), .CLKDV(clkdv2), .LOCKED(LOCKED2));

//CLOCK FEED BACK
BUFG BUFG3 (.I(clk2), .O(CLK0_W2));

//BUFG 4
BUFG BUFG4 (.I(clkdv2), .O(clk_2x));

//
// we
always @( negedge clk_2x )
if( rst )
   we <= 1;
else if( ~rst )
begin
   if( clk )
   begin
      if(     COND==`WP_WRITE_40H     ||     COND==`WP_WRITE_DATA     ||
COND==`WP_WRITE_FFH ||COND==`BE_WRITE_20H || COND==`BE_WRITE_CON ||
COND==`BE_WRITE_FFH ||COND==`CL_WRITE_60H || COND==`CL_WRITE_CON ||
COND==`CL_WRITE_FFH )
         we <= 0;
   else
         we <= 1;
   end
```

```
        else if( ~clk )
            we <= 1;
        end
        else
            we <= 1;

        //
        // ce
        always @( negedge clk_2x )
        if( rst )
                ce <= 1;
        else if( ~rst )
        begin
          if( clk )
          begin
            if(  COND==`WP_WRITE_40H  ||  COND==`WP_WRITE_DATA  ||
COND==`WP_WRITE_FFH  ||COND==`WP_READ_STS  ||  COND==`WP_CHECK_SR7
||COND==`BE_WRITE_20H || COND==`BE_WRITE_CON || COND==`BE_WRITE_FFH
||COND==`BE_READ_STS  ||  COND==`BE_CHECK_SR7  ||  COND==`CL_WRITE_60H
|| COND==`CL_WRITE_CON || COND==`CL_WRITE_FFH ||COND==`CL_READ_STS
|| COND==`CL_CHECK_SR7 )
              ce <= 0;
            else
              ce <= 1;
            end
              else if( ~clk )
              begin
                if(  COND==`WP_READ_STS  ||  COND==`WP_CHECK_SR7
||COND==`BE_READ_STS || COND==`BE_CHECK_SR7 ||COND==`CL_READ_STS ||
COND==`CL_CHECK_SR7 )
                  ce <= 0;
              else
                ce <= 1;
              end
            end
            else
              ce <= 1;

        //
        // oe
        always @( negedge clk_2x )
        if( rst )
          oe <= 1;
        else if( ~rst )
        begin
        if(  COND==`WP_READ_STS  ||  COND==`WP_CHECK_SR7
||COND==`BE_READ_STS || COND==`BE_CHECK_SR7 ||COND==`CL_READ_STS ||
COND==`CL_CHECK_SR7 )
            oe <= 0;
        else
            oe <= 1;
        end
```

```
//
// COND
always @( posedge clk )
if( rst )
   COND <= `WAIT;
else if( ~rst )
begin
   if( COND==`WAIT )
            COND <= `CL_BEGIN;
            // WORD PROGRAM
            else if( COND==`WP_BEGIN )
                     COND <= `WP_WRITE_40H;
            else if( COND==`WP_WRITE_40H )
                     COND <= `WP_WRITE_DATA;
            else if( COND==`WP_WRITE_DATA )
                     COND <= `WP_WAIT_STS;
            else if( COND==`WP_WAIT_STS )
            begin
                     if( ~sts )
                              COND <= `WP_WAIT_STS;
                     else if( sts )
                              COND <= `WP_READ_STS;
            end
            else if( COND==`WP_READ_STS )
                     COND <= `WP_CHECK_SR7;
            else if( COND==`WP_CHECK_SR7 )
            begin
                     if( bus[`SR7] )
                              COND <= `WP_GET_SRD;
                     else if( ~bus[`SR7] )
                              COND <= `WP_CHECK_SR7;
            end
            else if( COND==`WP_GET_SRD )
                     COND <= `WP_WRITE_FFH;
            else if( COND==`WP_WRITE_FFH )
                     COND <= `WP_END;
            else if( COND==`WP_END )
                     COND <= `WP_END;
            // BLOCK ERASE
            else if( COND==`BE_BEGIN )
                     COND <= `BE_WRITE_20H;
            else if( COND==`BE_WRITE_20H )
                     COND <= `BE_WRITE_CON;
            else if( COND==`BE_WRITE_CON )
                     COND <= `BE_WAIT_STS;
            else if( COND==`BE_WAIT_STS )
            begin
                     if( ~sts )
                              COND <= `BE_WAIT_STS;
                     else if( sts )
                              COND <= `BE_READ_STS;
```

```
        end
        else if( COND==`BE_READ_STS )
                COND <= `BE_CHECK_SR7;
        else if( COND==`BE_CHECK_SR7 )
        begin
                if( bus[`SR7] )
                        COND <= `BE_GET_SRD;
                else if( ~bus[`SR7] )
                        COND <= `BE_CHECK_SR7;
        end
        else if( COND==`BE_GET_SRD )
                COND <= `BE_WRITE_FFH;
        else if( COND==`BE_WRITE_FFH )
                COND <= `BE_END;
        else if( COND==`BE_END )
                COND <= `WP_BEGIN;
        // CLEAR LOCK BIT
        else if( COND==`CL_BEGIN )
                COND <= `CL_WRITE_60H;
        else if( COND==`CL_WRITE_60H )
                COND <= `CL_WRITE_CON;
        else if( COND==`CL_WRITE_CON )
                COND <= `CL_WAIT_STS;
        else if( COND==`CL_WAIT_STS )
        begin
                if( ~sts )
                        COND <= `CL_WAIT_STS;
                else if( sts )
                        COND <= `CL_READ_STS;
        end
        else if( COND==`CL_READ_STS )
                COND <= `CL_CHECK_SR7;
        else if( COND==`CL_CHECK_SR7 )
        begin
                if( bus[`SR7] )
                        COND <= `CL_GET_SRD;
                else if( ~bus[`SR7] )
                        COND <= `CL_CHECK_SR7;
        end
        else if( COND==`CL_GET_SRD )
                COND <= `CL_WRITE_FFH;
        else if( COND==`CL_WRITE_FFH )
                COND <= `CL_END;
        else if( COND==`CL_END )
                COND <= `BE_BEGIN;
        else
                COND <= `WAIT;
end

//
// data & adr
always @( negedge clk_2x )
```

```
if( rst )
begin
        data <= 16'hz;
        adr <= 24'hz;
end
else if( clk )
begin
        if( COND==`WAIT )
        begin
                data <= 16'hz;
                adr <= 24'hz;
        end
        // WORD PROGRAM
        else if( COND==`WP_BEGIN )
        begin
                data <= 16'hz;
                adr <= 24'hz;
        end
        else if( COND==`WP_WRITE_40H )
        begin
                data <= `WC;
                adr <= `PA;
        end
        else if( COND==`WP_WRITE_DATA )
        begin
                data <= `PD;
                adr <= `PA;
        end
        else if( COND==`WP_WAIT_STS )
        begin
                data <= 16'hz;
                adr <= 24'hx;
        end
        else if( COND==`WP_READ_STS )
        begin
                data <= 16'hz;
                adr <= 24'hx;
        end
        else if( COND==`WP_CHECK_SR7 )
        begin
                data <= 16'hz;
                adr <= 24'hx;
        end
        else if( COND==`WP_GET_SRD )
        begin
                data <= 16'hz;
                adr <= 24'hx;
        end
        else if( COND==`WP_WRITE_FFH )
        begin
                data <= 16'hFF;
                adr <= 24'hx;
```

```
end
else if( COND==`WP_END )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
// BLOCK ERASE
else if( COND==`BE_BEGIN )
begin
        data <= 16'hz;
        adr <= 24'hz;
end
else if( COND==`BE_WRITE_20H )
begin
        data <= `EC;
        adr <= `BA;
end
else if( COND==`BE_WRITE_CON )
begin
        data <= `CON;
        adr <= `BA;
end
else if( COND==`BE_WAIT_STS )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`BE_READ_STS )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`BE_CHECK_SR7 )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`BE_GET_SRD )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`BE_WRITE_FFH )
begin
        data <= 16'hFF;
        adr <= 24'hx;
end
else if( COND==`BE_END )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
```

```
// CLEAR LOCK BIT
else if( COND==`CL_BEGIN )
begin
        data <= 16'hz;
        adr <= 24'hz;
end
else if( COND==`CL_WRITE_60H )
begin
        data <= `CC;
        adr <= 24'hx;
end
else if( COND==`CL_WRITE_CON )
begin
        data <= `CON;
        adr <= 24'hx;
end
else if( COND==`CL_WAIT_STS )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`CL_READ_STS )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`CL_CHECK_SR7 )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND ==`CL_GET_SRD )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
else if( COND==`CL_WRITE_FFH )
begin
        data <= 16'hFF;
        adr <= 24'hx;
end
else if( COND==`CL_END )
begin
        data <= 16'hz;
        adr <= 24'hx;
end
// ELSE
else
begin
        data <= 16'hz;
        adr <= 24'hx;
end
```

```
        end

        //
        // seg0
        always @( posedge clk )
        if( rst )
                seg0 <= 8'b10111111; // 0
        else if( ~rst )
        begin
                if( COND==`WAIT )
                        seg0 <= 8'b10111111; // 0
                else if( COND==`CL_BEGIN )
                        seg0 <= 8'b10000110; // 1
                else if( COND==`CL_WRITE_60H )
                        seg0 <= 8'b11011011; // 2
                else if( COND==`CL_WRITE_CON )
                        seg0 <= 8'b11001111; // 3
                else if( COND==`CL_WAIT_STS )
                        seg0 <= 8'b11100110; // 4
                else if( COND==`CL_READ_STS )
                        seg0 <= 8'b11101101; // 5
                else if( COND==`CL_CHECK_SR7 )
                        seg0 <= 8'b11111101; // 6
                else if( COND==`CL_GET_SRD )
                        seg0 <= 8'b10000111; // 7
                else if( COND==`CL_WRITE_FFH )
                        seg0 <= 8'b11111111; // 8
                else if( COND==`WP_END )
                        seg0 <= 8'b11101111; // 9
                else
                        seg0 <= 8'b10000000;
        end

        // seg1
        always @( rst or sts )
        if( rst )
                seg1 <= 8'b10111111; // 0
        else if( ~rst )
                seg1 <= sts;

endmodule
```