FPGAボード上での単一サイクル
マイクロプロセッサの設計と検証

学籍番号　：　２２１００１０１５６－８
氏　　名　：難波　　翔一朗
指導教員　：　山崎　勝弘教授

提　出　日　：　2005 年 2 月 21 日

内容梗概

本論文では、Verilog HDL を用いて単一サイクルマイクロプロセッサを設計し、FPGA ボード上で動作検証を行った。作成したプロセッサは単純な命令と構成を目標とし、SOAR(Simple Operation and Architecture RISC)と名づけた。SOAR プロセッサは 1 から独自に命令セットを考案し設計を行った。SOAR 上で動かすプログラムの開発環境としては、SOAR 命令セット専用のアセンブラ、仮想シミュレータも作成し、それらを用いることで、プロセッサに最適なプログラミングを行える環境を提供した。次に、本研究室のプロジェクトの 1 つであるハード／ソフト・コラーニングシステムのプロセスを利用して SOAR プロセッサを FPGA ボード上に実装を行った。独自のプロセッサをハード／ソフト・コラーニングシステム上で検証することで、システムの問題点を検討し、ハード／ソフト・コラーニングシステムの拡張案をまとめた。

ハード／ソフト・コラーニングシステムでは、本来学習用の基本命令セット MONI が用意されているが、独自に作成したプロセッサでも、インタフェースを MONI と同様に揃えることで、FPGA ボード上に搭載が可能である。この点を利用し、学習者が基本命令セットの MONI に独自の命令を追加できる、次のステップのハード／ソフト・コラーニングシステムを考案した。独自に設計したプロセッサを搭載できるシステムであれば、学習者にとって自由度の高い学習システムを提供でき、より積極的にハードウェアとソフトウェア協調学習が進められる。このような点を踏まえ、ハード／ソフト・コラーニングシステムの学習体系を拡張し、プロセッサデバッガと汎用アセンブラ／シミュレータを取り入れたシステムの検討を行った。

FPGA ボード上での実装には、ハード／ソフト・コラーニングシステムであらかじめ用意されている MONI システムと、独自に作成した周辺モジュールで構成するコンピュータシステムにおいて動作検証を行った。ハード／ソフト・コラーニングシステム上での検証では、インタフェースの調節のみの簡単な作業で実装ができ、プロセッサのみの実機上での検証を行った。独自に作成した周辺モジュールにおいては、システムの起動から、メモリ間データ転送、MPU の実行という一連の流れをシステム全体で検証し FPGA ボード上で動作確認を行った。
SOAR プロセッサの動作検証では、ランレングスエンコーディング、濃淡画像の二値化、各種ソートプログラムを FPGA ボード上で実行し、正しい結果が得られた。

目次

内容梗概 i

1 はじめに 1

1.1 研究背景 1
1.2 研究目的 4
1.3 研究概要 4
1.4 論文構成 4

2 単一サイクルマイクロプロセッサの設計 5

2.1 設計思想 5
2.2 命令セット 5
2.3 アーキテクチャ 8
2.4 Verilog HDLによる設計 14
2.5 HDLシミュレータ上での検証 15
2.6 プログラム開発用ツール 15

3 FPGA上でのハード／ソフト･コラーニングシステムを用いた実装 18

3.1 実装内容 18
3.2 SOARの接続 18
3.3 動作検証 19
3.4 考察 19

4 FPGA上での検証システムを用いた実装 21

4.1 システムシーケンサ 22
4.2 FlashRAM Moduleコントローラ 32
4.3 クロックカウンタ 32
4.4 7 セグメントデコーダ 33
4.5 システムバス 33
4.6 システムの性能評価 34
4.7 SOARプロセッサの動作検証 35
4.8 考察 35

5 プロセッサデバッガの検討 37

5.1 プロセッサデバッガ 37
5.2 汎用アセンブラ･汎用シミュレータ 37
5.3 ハード／ソフト･コラーニングシステムの拡張の検討 37
5.4 今後の課題 38

6 おわりに 40

謝辞 41

参考文献 42

図リスト

図 1:ハード/ソフト・コラーニングシステムの学習体系 ……… 3
図 2:SOARの命令形式 ……… 6
図 3:SOARのデータパス ……… 8
図 4:レジスタ間演算命令の実行過程 ……… 9
図 5:即値演算命令の実行過程 ……… 10
図 6:ディスプレースメント付き間接アドレッシングのロード命令(LDR)の実行過程 ……… 10
図 7:ディスプレースメント付き間接アドレッシングのストア命令(STR)の実行過程 ……… 11
図 8:即値アドレッシングのロード命令(LD)の実行過程 ……… 11
図 9:即値アドレッシングのストア命令(ST)の実行過程 ……… 12
図 10:LDLI/LDHI命令の実行過程 ……… 12
図 11:条件分岐命令の実行過程 ……… 13
図 12:無条件分岐命令の実行過程 ……… 13
図 13:仮想SOARシミュレータの実行例 ……… 17
図 14:MONI周辺システムとSOARの接続の様子 ……… 18
図 15:SOARコンピュータシステム ……… 21
図 16:システムシーケンサの状態遷移 ……… 22
図 17:INITIAL時の動作 ……… 23
図 18:LOAD時の動作 ……… 25
図 19:READY時の動作 ……… 25
図 20:RUN時の動作 ……… 26
図 21:BREAK1 時の動作 ……… 28
図 22:ERASE時の動作 ……… 28
図 23:BREAK2 時の動作 ……… 29
図 24:STORE時の動作 ……… 31
図 25:FINISH時の動作 ……… 31
図 26:FRMコントローラの状態遷移 ……… 32
図 27:入出力ポートとバスの接続 ……… 34
図 28:拡張後のハード/ソフト・コラーニングシステムの学習体系 ……… 38

表リスト

表 1:命令フィールドの意味 ……… 6
表 2:SOAR命令セットの一覧 ……… 7
表 3:SOARプロセッサの設計規模 ……… 14
表 4:SOARプロセッサの性能評価 ……… 14
表 5:HDLシミュレーションによるサンプルプログラムの実行結果 ……… 15
表 6:アセンブラの実行オプション一覧 ……… 16
表 7:7 セグメントLED表示用文字コード表 ……… 33
表 8:コンピュータシステムの回路規模 ……… 34
表 9:コンピュータシステムの性能評価 ……… 34
表 10:FPGAボード上でのSOARプロセッサ実行結果 ……… 35

1　はじめに

1.1 研究背景

半導体デバイスは 1950 年代に真空管からトランジスタへと移り替わり、その後、多くの電子回路システムは複数のトランジスタを集積した IC(Integrated Circuit)によって構成されるようになった。IC は半導体製造技術の発展に伴い、LSI(Large Scale Integration)、VLSI(Very LSI)へとその回路規模を拡大し、近年は 1 億ゲートを越える LSI も出現している[11]。このような回路規模の拡大を受けて、システム全体を 1 チップに収めるシステム LSI、もしくは、SoC(System on Chip)の設計が回路設計の中心となってきた。現在ではコンピュータや自動車、携帯電話、家電製品など、様々な製品にシステム LSI が搭載されるようになり、その性能が直接製品の性能を左右する重要な基幹部品となっている[12]。さらに、大規模なシステム LSI の開発には、90nm プロセスの超微細加工技術や高速論理合成アルゴリズム、ゲートレベルシミュレータなどの製造機器や設計支援ツールの発展は欠かせず、システム LSI 設計を取り巻く産業は今日の情報化社会の基盤となる分野であるといえる[13]。

LSI の設計規模が数百万ゲートを越える現在、設計期間の伸長がコストを増大させ、大きな問題となっている。そこでシステム LSI 開発者は、いかに設計期間を短縮するかという点に着目し、コンピュータによる回路設計の自動化を進めてきた。同時に、設計フローも、まずゲートレベル設計を行い最終的な製品ターゲットを作成するボトムアップデザインフローから、抽象度の高い設計仕様を作成後、機能記述を論理合成することで実際の回路情報に落とし込むトップダウンデザインフローに変化してきた。抽象度の高い HDL(Hardware Description Language)や、さらに上位のシステム記述言語である Handel C や System C などで設計を行うことで、早期の段階で容易に検証が行える。また、各段階で完成しているブロックを統合してシミュレーションを行えるので、設計の遅れているブロックに全体の設計期間が引っ張られず、全体として整合性がとれるといった利点もある[12]。

HDL(Hardware Description Language)は、動作レベルからゲートレベルにわたる幅広いレベルでハードウェアを記述できる言語である。機能レベル／RTL(Register Transfer Level)で表現した HDL 記述は、論理合成という処理を行うことで、ゲートレベルの情報に変換することができる。HDL を用いた LSI 設計は、従来の人の手による回路設計からコンピュータ上での設計に移行する過程で取り入れられてきた。もともと、設計ツールを制御する言語として開発されたが、論理合成ツールの技術向上によって広く普及し、現在では HDL による RTL 設計が主流となっている。中でも Verilog HLD と VHDL は、早期から IEEE で規格が標準化され、現在最も広く使用されている HDL である。

HDL の特徴として以下の点が挙げられる。

- ハードウェアに必要な並列処理や時間の概念が考慮されており、遅延の情報も扱える。
- プログラミング言語としての機能もあり、同じ HDL でシミュレーションの記述が可能である。
- 抽象度の高い動作レベルやゲート／スイッチレベルの記述を併用することで、設計効率のよい回路設計が行える。
- IEEE などで広く標準化が進んでおり、ターゲット回路や設計開発するマシン、また半導体ベンダに依存しない回路設計が可能である。
- 汎用性のある記述で、設計した回路の再利用が可能である。また、IP の普及により、CPU

や各種エンコーダが RTL で IP ベンダから提供されるようになり、高機能なハードウェアを容易に設計可能である。

FPGA(Field Programmable Gate Array)は、書き換え可能なハードウェアチップの 1 つである。回路の機能を実現する論理ブロックと、それらを接続するスイッチによって構成される。FPGA の構成要素は製造している各企業によって異なるが、主に SRAM で構成されていることが多い。SRAM を格子状に配置し、導線とスイッチによって結線を行っている。FPGA はメモリで構成されているため、回路の再構成を行うか、電源を落とさない限り 1 つの LSI チップとして動作する。回路構成を何度でも書き換えられるので、ハードウェア上での動作検証が可能であり、主に LSI のテストや、頻繁にアップデートが必要な機器に組み込まれて使用されている。

本研究の背景として、高性能計算研究室で研究を進めているハード／ソフト・コラーニングシステムについて紹介する。
近年の SoC 設計の時代背景から、LSI 設計現場ではハードウェアとソフトウェアの両方を理解できる人材が求められている。ハードウェア面では、レジスタ間の遅延やバスの共有を考慮した回路設計、要求された性能と制約を満たす設計空間探索能力、基本的なプロセッサアーキテクチャの理解などが挙げられる。また、ソフトウェア面では、ターゲットとするプロセッサアーキテクチャに最適なプログラミング、最適化コンパイラの設計、LSI 制御を行うプログラムの開発などが挙げられる。そこで、高性能計算研究室では、これらの能力を身に付ける体系的な学習環境としてハード／ソフト・コラーニングシステムを提案し、研究を進めてきた。
ハード／ソフト・コラーニングシステムとは、学習者がコンピュータシステムにおけるプロセッサアーキテクチャを、ソフトウェアとハードウェアの両側面から設計することによって、両者の理解を協調的に進める学習システムである。本システムの学習内容としては、プロセッサアーキテクチャを意識したアセンブリプログラミングを行うソフトウェア学習と、ソフトウェア学習で設計したプログラムを実行するプロセッサを設計するハードウェア学習のフローがある。ハード／ソフト・コラーニングシステムでは、複数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、スーパスカラ)をターゲットに、ソフトウェア学習とハードウェア学習を通じて双方の特徴を理解した高品質なシステムを設計できる人材をはぐくむことを目的とする。将来的には、本システムを学部生の演習や実験に導入し、実際の教育現場で活用できる学習システムとしたい。
本システムでは、現在までに基本命令セット MONI を中心とする学習環境を提供し、ハートとソフトの協調学習体系を実現している[1][3][4]。図 1に現在のハード／ソフト・コラーニングシステムの学習体系を示す。

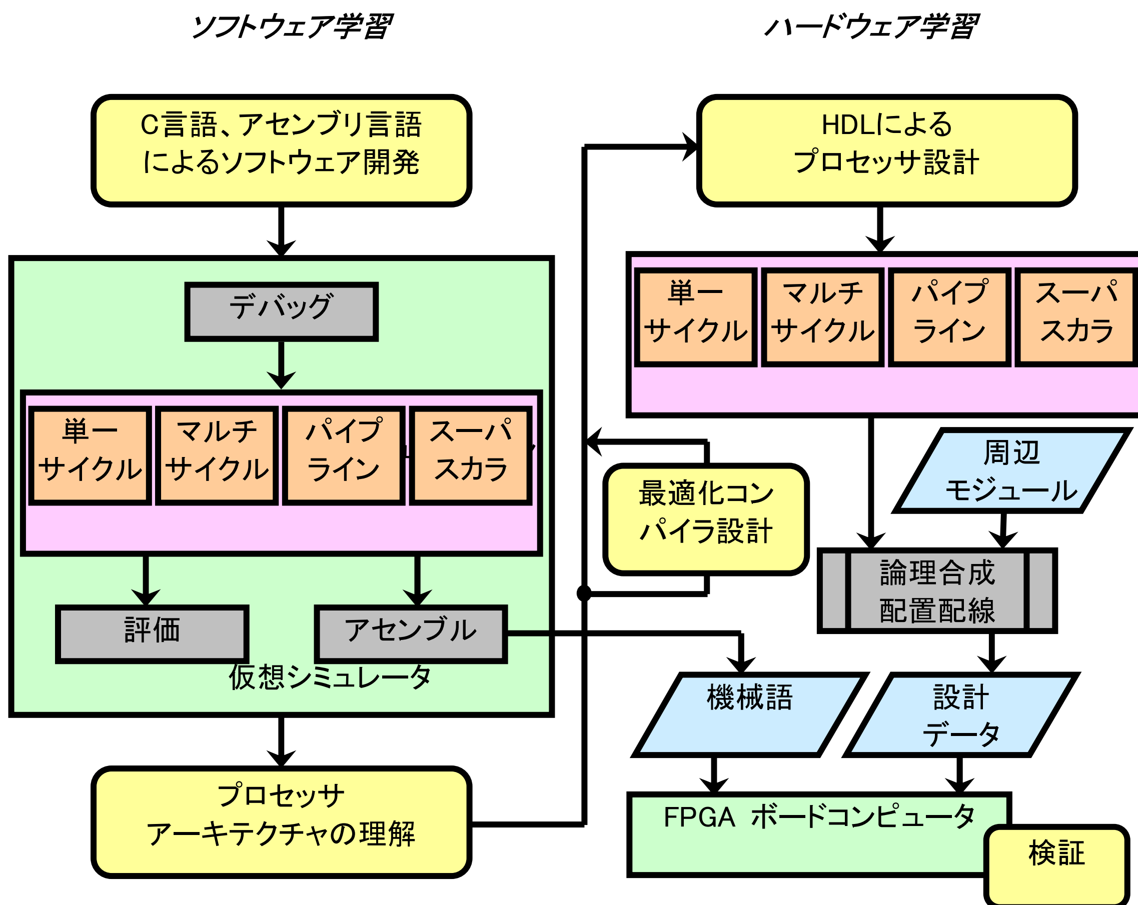

図1：ハード／ソフト・コラーニングシステムの学習体系

まず、学習者は、仮想シミュレータ上でアセンブリ言語によるプログラミングを行う。仮想シミュレータでは、シミュレーション方法として MONI 命令セットの 4 つのアーキテクチャ(単一サイクル、マルチサイクル、パイプライン、スーパスカラ)が選択可能で、それぞれの動作を、1 命令ごと、または 1 フェーズごとに確認しながらシミュレーションを行うことができる。これにより、MONI プロセッサの内部動作を理解しながらプログラミングができ、ハードウェアを意識したソフトウェアの設計手法が身に付けられる。プロセッサアーキテクチャを理解した後は、発展課題として、高級言語から MONI の各プロセッサアーキテクチャに最適な機械語を生成する、最適化コンパイラの設計を行う学習も用意している。ここでは、コンパイラの設計を通して、学習したプロセッサアーキテクチャの応用技術を学ぶ。次に、学習者は HDL を用いて実際に MONI プロセッサの設計を行う。HDL での設計を通じて、ハードウェア特有の性質である遅延や設計規模などを考慮する設計手法を理解する。その後、あらかじめ用意してある周辺モジュールと共に論理合成を行い FPGA 上にコンピュータシステムとして実装、プロセッサの動作検証を行う。動作検証には、ソフトウェア学習のフローで作成したプログラムを用いて、FPGA 上で実行する。

以上のように、研究背景として LSI の規模の拡大に伴うシステム LSI の開発環境の変化やハード／ソフト・コラーニングシステムを挙げたが、その他の LSI 技術として、Handel C や System C などのシステム設計言語と高位合成技術を用いたシステム設計の発展、IP(Intellectual Property)の普及による設計資産の再利用、処理プロセスにあわせて自身のハードウェアをリアルタイムに再構成できる動的再構成可能なマルチコンテキスト型のハードウェアチップの出現[16][17][18][19][20][21][22][23][24][25]、システム設計の段階から性能やコストが最適になるように、ハードとソフトの分割を決定するハード／ソフト協調設計などが挙げられる。

1.2 研究目的

以上のような背景を踏まえ、本研究では大きく分けて次の二点を研究目的とする。

第一に、マイクロプロセッサを中心とする FPGA コンピュータシステムの設計と実装を行い、システム LSI として動作検証を行う。組み込みシステムには欠かせないマイクロプロセッサを実際に命令セットから考案し HDL を用いて設計することによって、プロセッサの基本的なアーキテクチャの理解、HDL によるシステム設計の手法の習得を目的とする。さらに、設計した MPU を FPGA ボード上に実装し、動作検証を行うことで、システム LSI としての完成を目指す。加えて、設計したプロセッサ上で動作するプログラムの開発環境としてアセンブラと命令レベルシミュレータを提供することで、プロセッサのソフトウェアインタフェースの理解も深める。

第二に、マイクロプロセッサの設計を通じて、ハード／ソフト・コラーニングシステムの問題点を考察する。そして、コラーニングシステムの次のステップとして、プロセッサデバッガの必要性を検討し、プロセッサデバッガを取り入れたシステム全体の構想をまとめる。これまでのコラーニングシステムでは、基本命令セットである MONI の命令レベルシミュレーションと、ハードウェア設計を行うことでハードとソフトの協調学習を進めるスタイルであった[1][3][4]。今回の検討では、MONI の命令セットを拡張することでハードとソフトを理解する、より発展した学習スタイルを提案する。MONI を拡張する学習スタイルを採用することで、まず拡張前の基本的なプロセッサの命令セットアーキテクチャの理解を深め、その後、アセンブリプログラミングを通してターゲットとするプログラムに最適な命令セットを熟慮できる。また、拡張した命令セットをハードウェア化する段階でも容易に修正できるように、汎用アセンブラ・シミュレータとプロセッサデバッガを連動させたツールを提供する。これにより、ハードとソフトの協調設計という本質部分の学習を体系的に進められるシステムとすることを目標とする。

1.3 研究概要

本研究で作成したプロセッサは 3 オペランド命令形式の 16bit-RISC プロセッサである。本研究室の教育用システム、ハード／ソフト・コラーニングシステムで用いる MONI 命令セットを参考に、プログラムの静的・動的命令数が小さくなるよう独自に命令セットを定義する。ターゲットとした FPGA ボードは Celoxica 社の RC100 ボードである。また、設計用の HDL として、本研究では Verilog HDL を用いる。検証方法としては、ハード／ソフト・コラーニングシステムの周辺回路を用いた検証と、システムシーケンサ、Flash RAM コントローラを新たに作成し独自に設計したコンピュータシステムでの検証を行う。

1.4 論文構成

本論文では、第 2 章で作成した単一サイクルマイクロプロセッサのアーキテクチャについて詳しく説明する。第 3 章でハード／ソフト・コラーニングシステム上での実装について述べ、第 4 章では FPGA ボード上での動作検証と性能評価について述べる。第 5 章ではプロセッサデバッガの有用性と活用方法の検討を行い、ハードとソフトをより体系的に学習できる拡張ハード／ソフト・コラーニングシステムの構想をまとめる。

2 単一サイクルマイクロプロセッサの設計

2.1 設計思想

本研究において設計したマイクロプロセッサは、16bit 単一サイクルの RISC プロセッサである。命令セットは、ハード／ソフト・コラーニングシステムで用いる基本命令セット MONI を参考に、4 命令形式、25 命令を独自に定義した。プロセッサ内部のマイクロアーキテクチャは、高速化のためできるだけ単純な構成とし、命令セットは分岐命令を工夫することで実行命令数が最小になるよう設計した。以下、この単一サイクルマイクロプロセッサを、その設計思想から SOAR(Simple Operation and Architecture RISC)と呼ぶ。SOAR は、一般的な組み込みシステム向けプロセッサの機能を備える汎用プロセッサとして設計を行った。また、ハード／ソフト・コラーニングシステムの拡張を考察するためのサンプルプロセッサとしても設計を行った。これらを踏まえ、次のような特徴を持たせた。

- すべての命令長は 16bit 固定。
- 1 命令 1 クロックで実行。
- 4 つの命令形式を用意。
- 3 オペランド方式の命令。
- 8 個の汎用レジスタを搭載。
- 演算は汎用レジスタ間のみで行うレジスターレジスタマシン。
- 命令とデータのアドレスを指定するポートは、クロック信号のレベルセンシティブ値(Low/High)で切り替えて共有。つまり、クロックの立ち上がりで命令をフェッチ、クロックの立ち下がりでデータメモリにアクセスする。
- 条件演算と分岐命令数の削減を目的にステータスレジスタを設置。
- op フィールドの上位 2bit を命令形式ごとに割り当て、命令デコードの高効率化を図った。
- 連続したデータ(配列など)に容易にアクセスできるように、ロード／ストア命令にはディスプレースメント付きレジスタ間接アドレッシングを採用。
- 56 命令まで拡張が可能。

2.2 命令セット

SOAR には主に Register, Immediate, Transfer, Jump の 4 つの命令形式を用意した。図 2に SOAR の命令形式を示す。

Instruction Format	Field				
	5	2	3	3	3
Register	op	fn	rs	rt	rd
Immediate	op	imm		rt	rd
Transfer	op	imm			rd
Jump	op	imm			

図2:SOAR の命令形式

命令を表記するフィールドの組み合わせを命令形式という。同じ命令形式で表現された命令では、同じフィールドのデータはすべて同様に処理される。このように同じ命令形式の各フィールドを同様に扱えることは、命令のデコードの際にハードウェアを単純化できるメリットがある[8]。SOAR の命令形式では 5 つのフィールドを区切り、4 つの命令形式を定義した。表 1に各フィールドの意味を示す。

表1:命令フィールドの意味

フィールド	意味	bit 幅	用途
op	OPeration	5	命令を識別するフィールド。
fn	FuNction	2	Register 形式の命令に対して、その詳細な種類を指定するフィールド。
rs	Register Source	3	演算元のレジスタを指定するフィールド。
rt	Register Target	3	演算先のレジスタを指定するフィールド。
rd	Register Destination	3	演算結果を格納するレジスタを示すフィールド。
imm	IMMediate	5～11	即値演算やデータ転送の定数。データメモリのアドレス、ジャンプ先の命令メモリのアドレス、演算のオペランドなどとして指定される。

Register 形式では、レジスタ間の演算を行う命令を定義している。fn フィールドにより演算の種類を詳細に指定できる。演算の種類は、ADD, SUB, AND, OR ,XOR, NOT, SLL, SRL, SRA である。

Immediate 形式では、主にレジスタの値と imm フィールドの値を用いて演算を行う命令を定義している。また、ディスプレースメント付きレジスタ間接アドレッシングモードで指定したメモリとレジスタ間のデータ転送を行う命令も定義している。Immediate 形式の命令は、ADDI, SUBI, SLLI, SRLI, SRAI, LDR, STR である。imm フィールドが 5bit であることから、演算命令(ADDI, SUBI, SLLI, SRLI, SRAI)では 0～31 の範囲、データ転送命令(LDR, STR)では-16～15 の範囲の即値しか指定できない。このため、即値命令では、ループ変数の加減算や bit アクセスのための固定長シフト用

に定義した。

Transfer 形式は、メモリ・レジスタへのデータ転送と条件分岐を行う命令を定義している。imm フィールドが 8bit であるためデータ転送命令(LD，ST，LDLI，LDHI)では 0～255 の範囲でのみアドレスを指定できる。条件分岐命令(BC，BNC)では-128～127 の範囲を指定できる。条件分岐命令では、PC 相対アドレッシングで分岐先を設定する。このため-128～127 以上のアドレスに分岐するためには無条件分岐命令を使用する。

Jump 形式では、無条件分岐命令と内部割込み命令を定義している。無条件分岐命令では 0～4095 の範囲の命令に分岐できる。内部割込み命令としては NOP，HALT 命令を定義している。これらはそれぞれ、空白命令とプログラムの終了命令である。

表 2に SOAR 命令セットの一覧を示す。

表2：SOAR 命令セットの一覧

	op	mean	tp		op			fn		rs	rt	rd	behavior
Jump	NOP	No Operation	0	0	0	0	0	xxxxxxxxxxx					PC++
	HALT	HALT	0	0	0	0	1	xxxxxxxxxxx					exit
	JUMP	JUMP	0	0	0	1	0	imm					PC=imm
Register	ADD	ADD	0	1	0	0	0	0	0	rs	rt	rd	(*rd)=(*rt)+(*rs)
	SUB	SUBstruct	0	1	0	0	0	0	1	rs	rt	rd	(*rd)=(*rt)−(*rs)
	AND	AND	0	1	0	0	0	1	0	rs	rt	rd	(*rd)=(*rt)&(*rs)
	OR	OR	0	1	0	0	0	1	1	rs	rt	rd	(*rd)=(*rt)|(*rs)
	XOR	eXclusive OR	0	1	0	0	1	0	0	rs	rt	rd	(*rd)=(*rt)^(*rs)
	NOT	NOT	0	1	0	0	1	0	1	rs	xxx	rd	(*rd)=~(*rs)
	SLL	Shift Left Logically	0	1	0	0	1	1	0	rs	rt	rd	(*rd)=(*rt)<<(*rs)
	SRL	Shift Right Logically	0	1	0	0	1	1	1	rs	rt	rd	(*rd)=(*rt)>>(*rs)
	SRA	Shift Right Arithmetically	0	1	0	1	0	0	0	rs	rt	rd	(*rd)=(*rt)>>(*rs)
Immediate	ADDI	ADD Immediate	1	0	0	0	0	imm			rt	rd	(*rd)=(*rt)+imm
	SUBI	SUBtruct Immediate	1	0	0	0	1	imm			rt	rd	(*rd)=(*rt)−imm
	SLLI	Shift Right Logically Immediate	1	0	0	1	0	imm			rt	rd	(*rd)=(*rt)<<imm
	SRLI	Shift Left Logically Immediate	1	0	0	1	1	imm			rt	rd	(*rd)=(*rt)>>imm
	SRAI	Shift Right Arithmetically Immediate	1	0	1	0	0	imm			rt	rd	(*rd)=(*rt)>>imm
	LDR	LoaD from Relatives address	1	0	1	0	1	imm			rt	rd	(*rd)=DM[imm+(*rt)]
	STR	STore to Relative address	1	0	1	1	0	imm			rt	rs	DM[imm+(*rt)]=(*rd)
Transfer	LDLI	LoaD Immediate to Low 8bit	1	1	0	0	0	imm				rt	(*rd)=imm*(2^0)
	LDHI	LoaD Immediate to High 8bit	1	1	0	0	1	imm				rt	(*rd)=imm*(2^8)
	LD	LoaD from absolute address	1	1	0	1	0	imm				rd	(*rd)=DM[imm]
	ST	STore to absolute address	1	1	0	1	1	imm				rs	DM[imm]=(*rd)
	BC	Branch on Condition	1	1	1	0	0	imm				rd	if(STATE==rd) PC+=imm
	BNC	Branch on Not Condition	1	1	1	0	1	imm				rd	if(STATE!=rd) PC+=imm

2.3　アーキテクチャ

図 3に SOAR のデータパスアーキテクチャを示す。

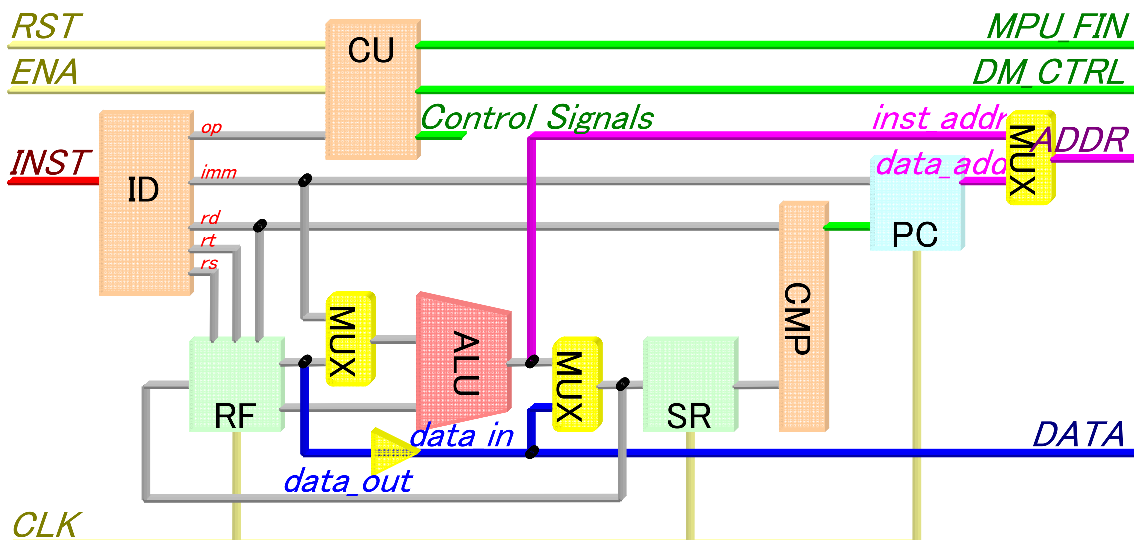

図3：SOAR のデータパス

SOAR は次の要素から構成される。

- ID(Instruction Decoder)
  命令メモリから読み出した命令を解読し、オペレーション、即値、レジスタ参照の 3 つの制御信号を出力する。
- CU(Control Unit)
  ID からオペレーションの種類を示す信号を受け取り、MPU 内の各モジュールに制御信号を出力する。
- RF(Register File)
  汎用レジスタ 8 個から構成される。ソースレジスタ 2 つと、デスティネーションレジスタ 1 つを同時に指定できる。ソースレジスタはアドレスを指定することでクロックに非同期で読み出しが可能。デスティネーションレジスタはクロックに同期して指定したアドレスにデータを書き込む。
- ALU(Arithmetic Logic Unit)
  2 入力 1 出力の算術論理演算装置。算術演算、論理演算、分岐先アドレス計算、データメモリアドレス計算を行う。
- SR(Status Register)
  1 命令前の演算結果の状態を格納するレジスタ。状態はゼロフラグとサインフラグを立てることで 4 種類が表現可能。すべての命令の実行ごとに必ず更新される。
- CMP(Comparator)
  BC もしくは BNC で指定された状態フラグと、SR の値を比較する。比較の結果、完全にマッチすれば PC に対して分岐先のアドレスにジャンプする信号を送信する。
- PC(Program Counter)
  現在実行中の命令メモリのアドレスを指し示す。

- MUX(Multiplexer)
  複数の入力信号から 1 つの出力信号を選択する。
- 3 ステートバッファ
  入出力ポートの信号が競合しないように制御する。図 3に示すように SOAR では 3 ステートバッファは Data の入出力の管理を行っている。入出力の切り替えには 1bit の制御信号が必要である。SOAR では CU が制御信号を送信している。

SOAR プロセッサは、RST を Low にすることですべてのモジュールにリセットがかかる。SOAR プロセッサの動作を開始するには、RST を Low から High に切り替え、ENA を High にセットする。次のクロックの立ち上がりから命令の読み出しが始まり、プログラムの実行が開始される。初めに実行される命令は常に 0 番地のアドレスである。命令は、クロックが Low のときに PC の値を ADDR ポートに流し、次のクロックの立ち上がりでメインメモリにアクセスすることで読み出す。読み出された命令は、ID、CU でデコードされ各モジュールの動作を制御して実行し、データのロードや演算、分岐先の計算などを行う。すべての命令において、PC、SR の値は自動的に更新される。以下に各命令の実行過程を図に示しながら解説する。

(1) レジスタ間演算命令(ADD, SUB, AND, OR, XOR, NOT, SLL, SRL, SRA)
    レジスタ間演算命令の場合は、RF から rs と rt が指すソースレジスタの値を取り出し、ALU で演算を行う。次のクロックの立ち上がりで rd が指す RF のデスティネーションレジスタに演算結果を書き込む。SR には演算結果の状態が保存される。図 4に演算命令の場合の流れを示す。

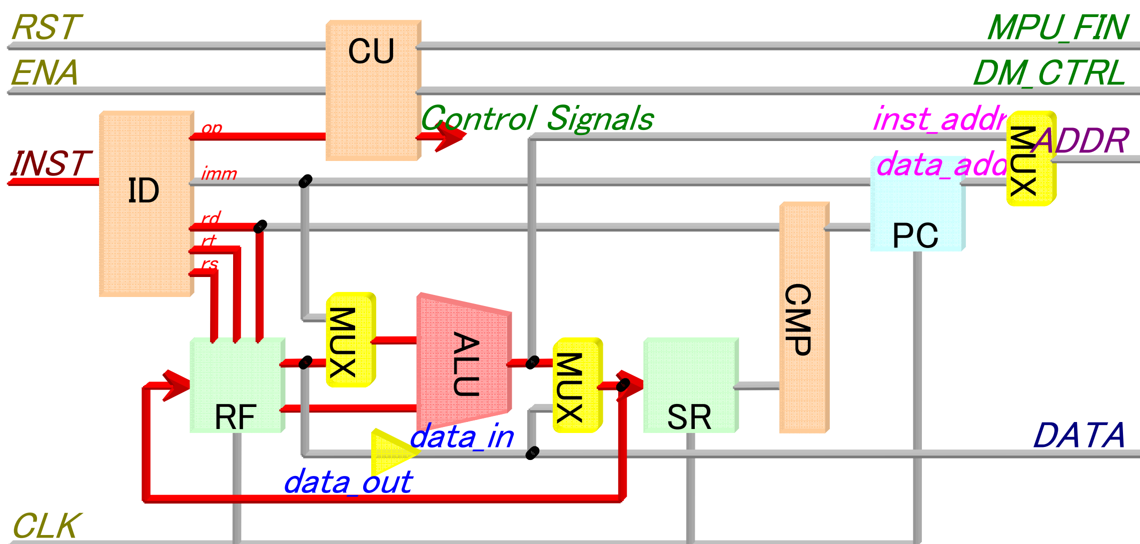

図4:レジスタ間演算命令の実行過程

(2) 即値演算命令(ADDI, SUBI, SLLI, SLRI, SRAI)
    即値演算命令の場合も同様に、rt が指す RF のソースレジスタの値と命令コードの即値 imm を ALU で演算し、次のクロックの立ち上がりで rd が指す RF のデスティネーションレジスタに結果を書き込む。SR には演算結果の状態が保存される。図 5に演算命令の場合の流れを示す。

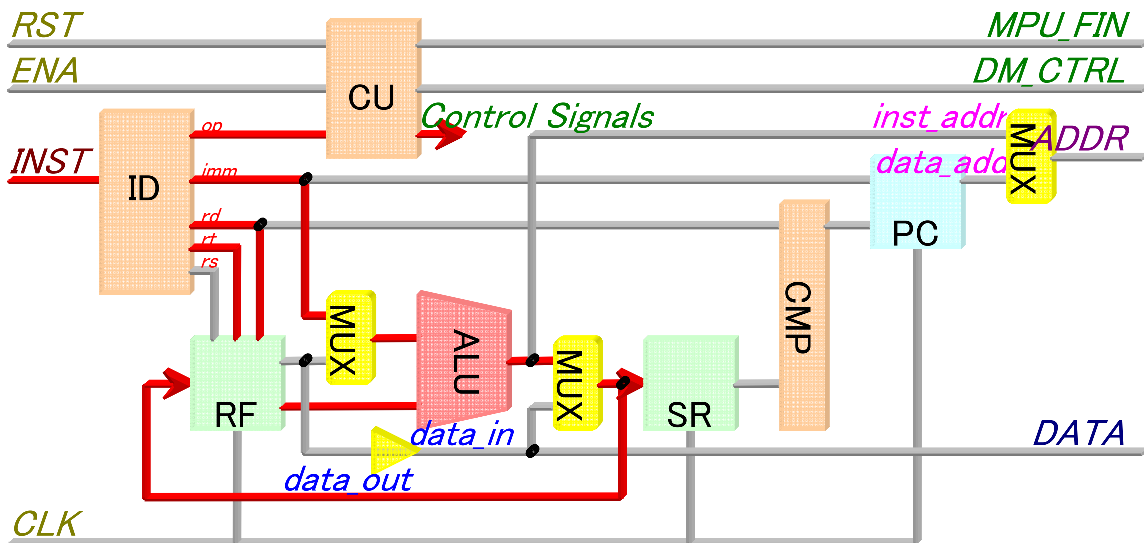

図5：即値演算命令の実行過程

(3)　ディスプレースメント付き間接アドレッシングロード／ストア命令(LDR, STR)

ディスプレースメント付き間接アドレッシングのロード／ストア命令では、rt が指す RF のソースレジスタの値と即値 imm の値を ALU で演算し、アクセスするデータメモリのアドレス計算を行う。演算結果は MUX を通してシステムのアドレスバスに流される。ロードの場合、データメモリから出力されたデータは MPU の DATA ポートより入力される。DATA ポートに入力されたデータは、次のクロックの立ち上がりで rd が指す RF のデスティネーションレジスタに保存される。ストアの場合は、rs が指す RF のソースレジスタのデータをデータメモリに転送する。SR には転送されたデータの状態が保存される。図 6と図 7にディスプレースメント付き間接アドレッシングのロード／ストア命令の流れを示す。

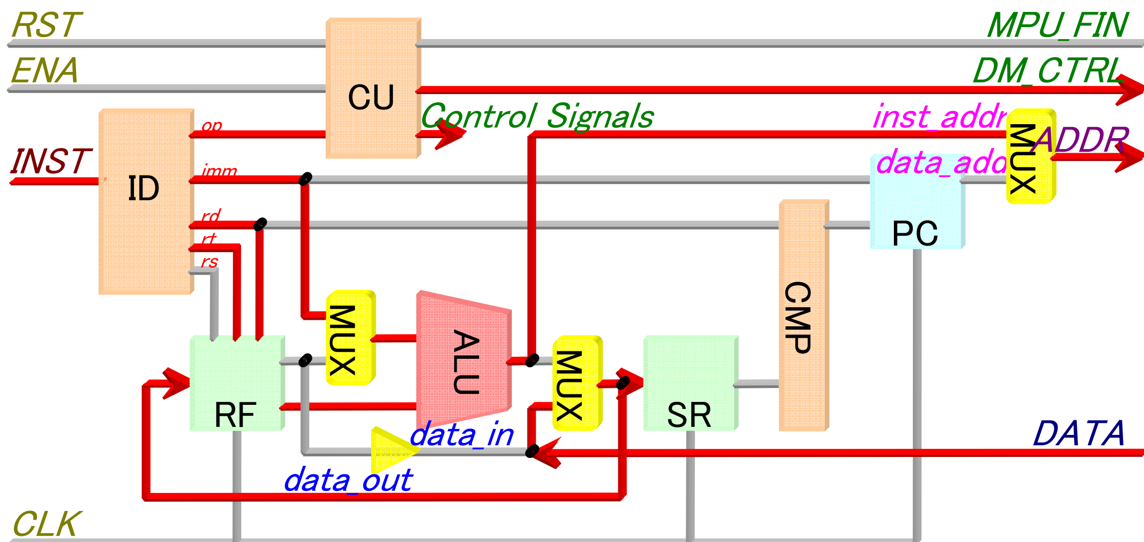

図6：ディスプレースメント付き間接アドレッシングのロード命令(LDR)の実行過程

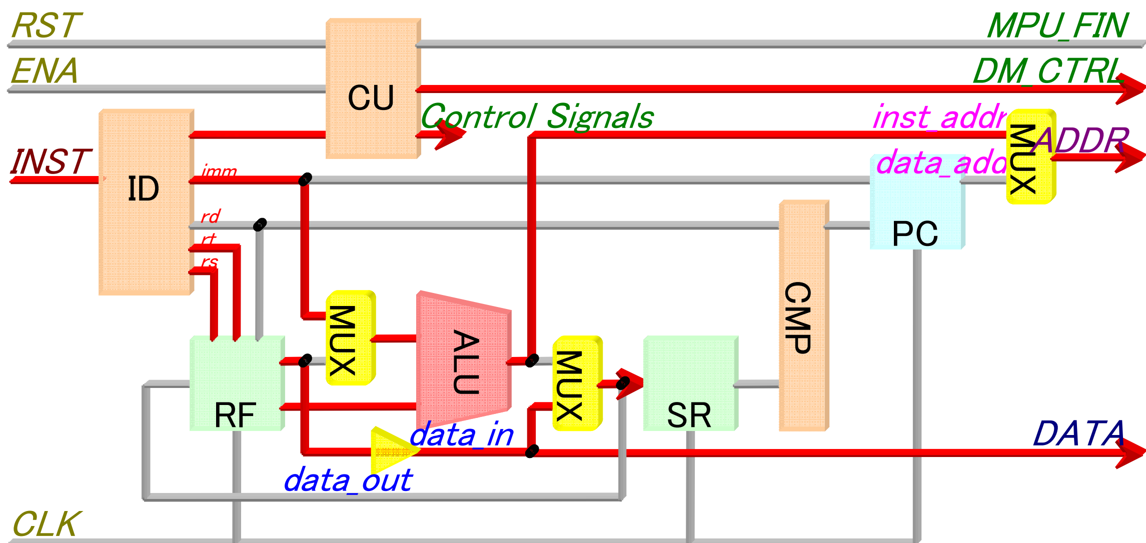

図7：ディスプレースメント付き間接アドレッシングのストア命令(STR)の実行過程

(4) 即値アドレッシングのロード／ストア命令(LD, ST)

即値アドレッシングのロード／ストア命令では、データメモリのアドレスを示す即値 imm が ALU を素通りし ADDR ポートへと流れる。データメモリから転送されてきたデータは、ディスプレースメント付き間接アドレッシングの LDR 命令と同様に rd が指す RF のデスティネーションレジスタに格納される。ST 命令の場合も同様に、rs が指す RF のソースレジスタの値を即値 imm が示すデータメモリに格納される。SR には転送されたデータの状態が保存される。図 8と図 9にディスプレースメント付き間接アドレッシングのロード／ストア命令の流れを示す。

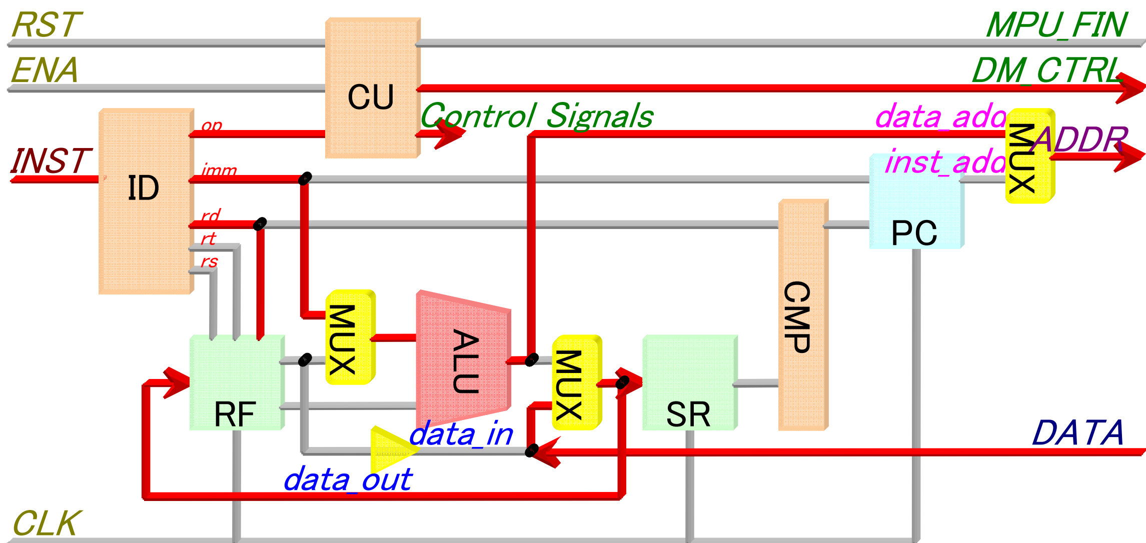

図8：即値アドレッシングのロード命令(LD)の実行過程

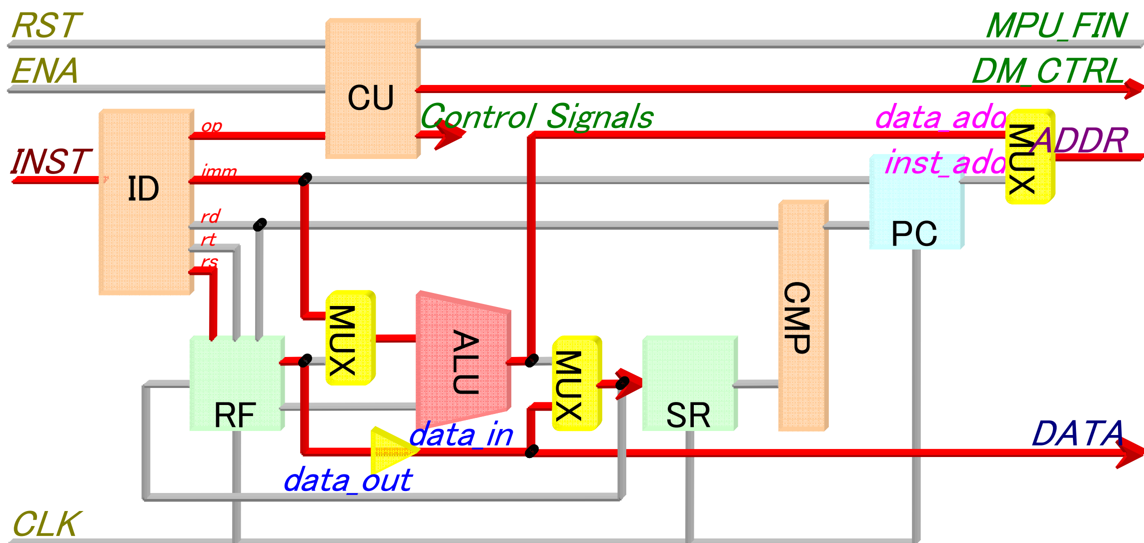

図9：即値アドレッシングのストア命令(ST)の実行過程

(5) 即値アドレッシングロード命令(LDLI, LDHI)

即値アドレッシングロード命令では、即値演算命令の流れとよく似ている。ただし、レジスタの指定が 1 つである。つまり rs でデスティネーションレジスタとソースレジスタ両方を同時に指定する。LDLI では下位 8bit を、LDHI では上位 8bit をロードし、再び rs が示す RF のデスティネーションレジスタに書き戻す。SR には、RF にロードされたデータの状態が格納される。図10に LDLI／LDHI 命令の実行過程を示す。

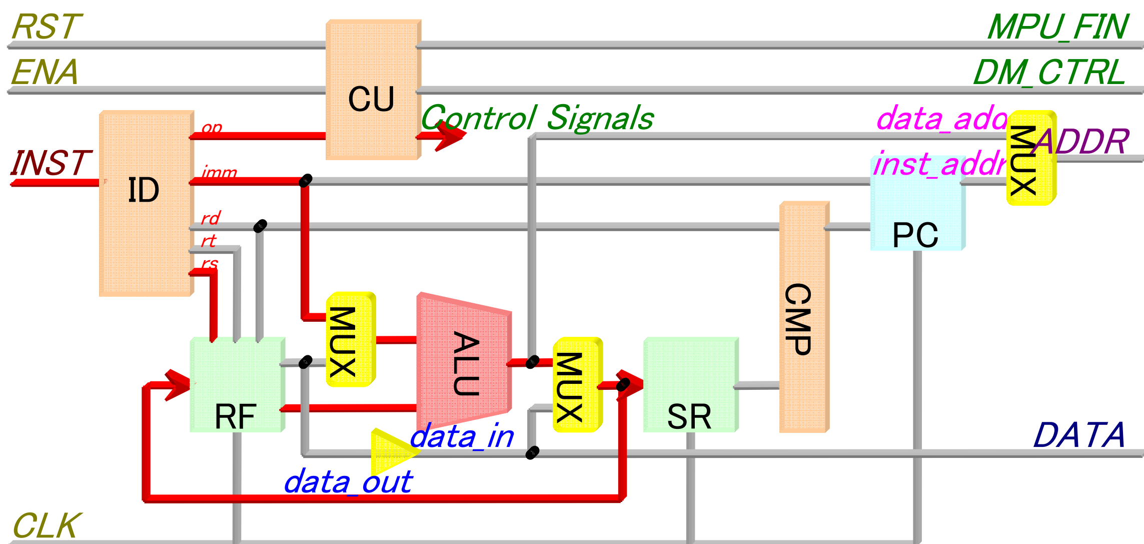

図10：LDLI／LDHI 命令の実行過程

(6) 条件分岐命令(BC, BNC)

条件分岐命令では、rd フィールドをフラグとして用いて SR と比較を行う。比較は CMP で評価され、直接 PC に分岐するかどうかの制御信号を送信する。分岐判定の信号を受信した PC は、現在の PC に即値の imm を加えた値を次の命令のアドレスとして出力する。図 11に条件

分岐命令の実行過程を示す。SR には imm 値の状態が格納される。

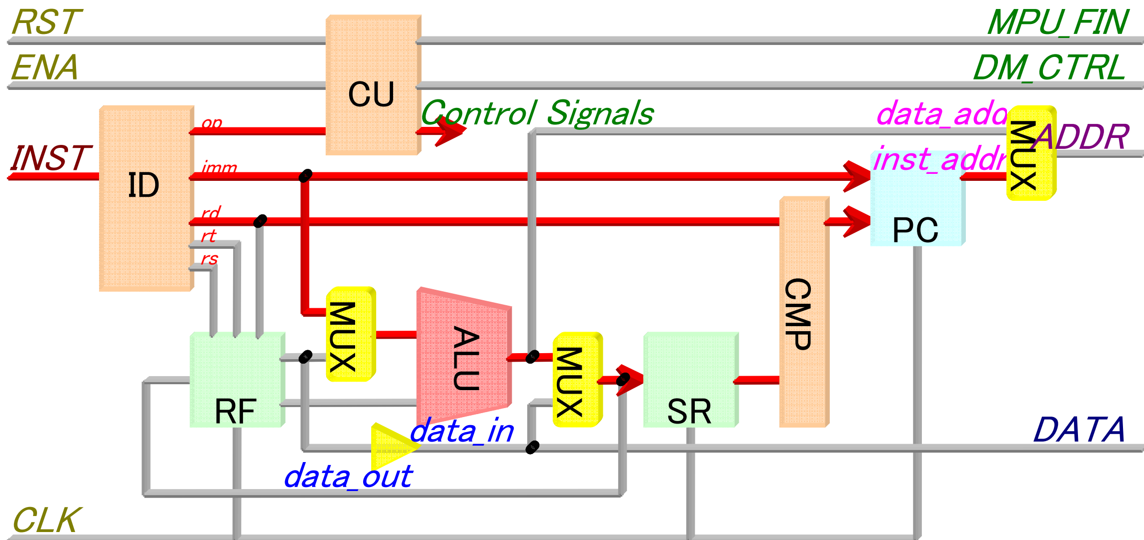

図11：条件分岐命令の実行過程

(7)　無条件分岐命令(JUMP)

無条件分岐命令では、即値 imm で指定したアドレスが PC に保存され、そのまま次の命令のアドレスとなる。SR には imm 値の状態が格納される。図 12に無条件分岐命令の実行過程を示す。

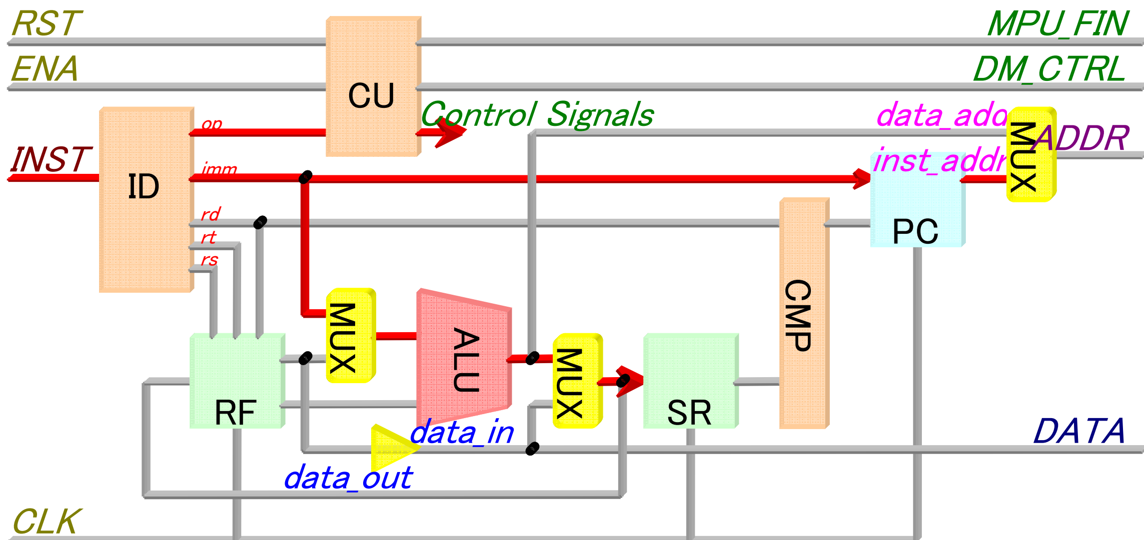

図12：無条件分岐命令の実行過程

2.4　Verilog HDL による設計

SOAR プロセッサの設計には Verilog HDL を用いた。開発環境として、Xilinx 社の統合開発環境ツールの ISE6.3i を使用した。論理合成ツールは XST、遅延の計算には Timing Analyzer を用いた。表 3と表 4に SOAR プロセッサの各モジュールの設計規模と性能試算値を示す。

表3：SOAR プロセッサの設計規模

モジュール	HDL 記述量(行)	スライス数	LUT 数	ゲート数
ID	94	18	35	267
CU	72	19	38	255
RF	18	32	64	4096
ALU	30	166	289	2172
SR	17	4	5	46
CMP	19	2	3	18
PC	30	14	19	248
MUX	11	2	16	18
SOAR	120	461	799	6822
ALL	436	－	－	－

表4：SOAR プロセッサの性能評価

モジュール	パス (元→先)	最大動作周波数 (MHz)	最大遅延 (ns)	ゲート遅延 (ns)	配線遅延 (ns)
ID	INST→SRC	72.622	13.770	5.992 (43.5%)	7.778 (56.5%)
CU	OP→DM_CTRL	73.932	13.526	7.636 (56.5%)	5.890 (43.5%)
RF	RA2→RD2	91	10.989	5.992 (54.5%)	4.997 (45.5%)
ALU	SRC→DST	41.217	24.262	11.383 (46.9%)	12.879 (53.1%)
SR	DIN1→STATE1	93.8	10.661	3.511 (32.9%)	7.150 (67.1%)
CMP	DIN2→SEL_BRN	87.078	11.484	6.540 (56.9%)	4.944 (43.1%)
PC	SEL_OFST→cnt	120.55	8.295	3.701 (44.6%)	4.594 (55.4%)
MUX	SEL→DOUT	84.353	11.855	5.992 (49.3%)	6.054 (50.7%)
SOAR	INST→ADDR	28.26	35.386	12.434 (35.1%)	22.952 (64.9%)
	DATA→RF	123.99	8.065	2.950 (36.6%)	5.115 (63.4%)
ALL	INST→ADDR→RF	23.014	43.451	15.384 (35.41%)	28.067 (64.59%)

表 3には、プロセッサの設計規模として、HDL の記述量と FPGA に実装した際の回路規模の試算値を示している。SOAR プロセッサの Verilog HDL での記述量は、各モジュールとヘッダファイルを合わせ計 436 行であった。また、SOAR プロセッサ全体の使用ゲート数は 8676 システムゲートと試算され、Spartan 2 FPGA チップの全ゲート数(20 万システムゲート)の約 2%である。

表 4には、Timing Analyzer で計算した回路の性能評価である。パスの列では、指定した入力から出力まで、もしくは、内部レジスタ間のパスのうち最も遅延が大きいものを示している。最大遅延のフィールドは AND や OR で発生するゲート遅延とゲート間を結ぶ配線遅延の合計である。表 4の ALL 項目では、SOAR プロセッサの最大遅延をディスプレースメント付き間接アドレッシングのロード命令の実行時と想定し、命令が入力されてからデータアドレスを出力するまでの遅延時間と、データが入力されてから RF に書き込むまでの時間を合計して最大遅延を算出した。前述したように、ディスプレースメント付き間接アドレッシングのロード命令は、データメモリのアドレスの計算をし

た後、メモリからそのアドレスのデータを読み出す。このため、アドレス計算とデータ転送の 2 つのパス遅延を考慮している。この結果、SOAR プロセッサの最高動作周波数は約 23MHz であった。

2.5 HDL シミュレータ上での検証

Veriog HDL で設計した SOAR は、Model Technology 社／Mentor Graphics 社の ModelSim XE II/Starter 5.8c を用いて HDL シミュレーションを行った。HDL シミュレーションでは、テストベンチ内に仮想的なメモリを記述し、実行前に mem ファイルからプログラムをロードしておく。SOAR プロセッサはこのテストベンチの仮想的なメモリにアクセスすることで、順次命令を読み出し、プログラムの実行シミュレーションを行う。テストベンチでは、N までの加算、最大値検索、バブルソートのサンプルプログラムを実行した。表 5にサンプルプログラムの実行結果を示す。

表5：HDL シミュレーションによるサンプルプログラムの実行結果

プログラム	静的命令数	動的命令数	実行クロック数	データ内容
N までの和	7	34	34	10 までの和
最大値検索	13	57	57	10 個のデータの最大値
バブルソート	18	599	599	10 個のデータのソート

HDL シミュレーションでのテストプログラム(N までの和、最大値検索、バブルソート)の実行結果は、それぞれ正常な値が得られた。動的命令数と、実行クロック数は同じ値となった。これは、CPI(clock cycle per instruction)が 1 であることを示している。つまり、MPU で命令を実行し始めてから HALT 命令を受け取り、実行が終了するまでに余分なクロックなどが混在せず、1 命令 1 クロックで実行できていることを示している。

2.6 プログラム開発用ツール

SOAR プロセッサの動作検証を行うに当たり、プログラムの設計効率を上げるためにアセンブラ asm と命令レベルの仮想 SOAR シミュレータ sim を作成した。どちらも C 言語で設計し、コマンドラインから実行する。asm は出力ファイルをオプションで指定することで、バイナリ、16 進、2 進、ModelSim のシミュレーション用の coe ファイル、mem ファイルなどさまざまな形式のファイルを出力できる。また、sim では、シミュレート後に実行命令数などを確認が可能である。

(1) アセンブラ

作成したアセンブラ asm は、次のプログラムファイルより構成している。

- asm.c：トップのソースファイル。79 行。
- asm.h：asm のヘッダファイル。55 行。
- Error.c：エラー出力を一括して管理するプログラム。25 行。
- LabelTable.c：ラベルの登録、参照、クリアを行うプログラム。ラベルとアドレスの情報をリスト構造の識別子表で管理し、ラベルをインデックスとしてテーブル参照を行うことでアドレスを得ることができる。40 行。
- Option.c：コマンドラインから渡されたオプションを解析するプログラム。107 行。
- OutputCode.c：アセンブル結果をファイルに出力するプログラム。オプションによってバイナリや ASCII 形式など、さまざまな形式で出力できる関数を持つ。224 行。

- Parser.c：構文解析を担当するプログラム。1 命令ずつ機械語に置き換える。243 行。
- Preprocess.c：前処理を担当するプログラム。コメントの除去、ラベルの登録、命令数のカウントなどを行う。128 行。
- Scanner.c：字句解析を行うプログラム。呼び出すごとに次の字句を切り出す ScanToken()と、切り出した字句を識別する ClassfyToken()の関数を含む。113 行。

全部で 9 ファイル、28 関数、1014 行のプログラムである。表 6に指定オプションの一覧を示す。

表6：アセンブラの実行オプション一覧

オプション	内容	指定条件
−b [bin ファイル]	バイナリファイルの指定出力。	
−c [coe ファイル]	FPGA 内部に組み込んだ Xilinx の BlockRAM 初期化ファイル(coe ファイル)の指定(出力)。	−n と−d の指定
−d [dm ファイル]	ASCII 表記でデータメモリを表現したファイルの指定(入力)。仮想 MPU シミュレータ、もしくは、coe／mem ファイル用のデータメモリとして使用する。	
−h [hex ファイル]	16 進表記の hex ファイルの指定(出力)。	
−m [mem ファイル]	Verilog HDL で記述したテストベンチ内のメモリの初期化に用いる mem ファイルの指定(出力)。	−n と−d の指定
−n [size]	coe／mem に保存するメモリファイルのデータ数。命令メモリとデータメモリそれぞれのメモリサイズとして設定される。	0<size
−s	仮想 SOAR シミュレーションの実行。	−d の指定

(2)　仮想 SOAR シミュレータ

仮想 SOAR シミュレータは、コンピュータ上に仮想的な SOAR プロセッサを実現し、1 命令ごとに命令レベルのシミュレーションを行うツールである。1 命令ごとのレジスタの変化が標準出力に表示されるので、プログラムのデバッグに活用できる。仮想 SOAR シミュレータは C 言語で設計し Simulate.c にまとめた。設計規模は 372 行であった。仮想 SOAR シミュレータはアセンブラ asm の処理過程で呼び出し実行する。asm のコマンドラインで−s オプションを指定することで、アセンブリソースのアセンブル後に実行される。図 13に asm の実行例を示す。

```
% ../asm Power.as -m Power.mem -b Power_2048.bin -c Power_2048.coe -n 2048 -d Data_dm.txt -s
> Analyzing asm options.
> Pre-processing.
> Assembling program.....................
******** 21 instructions are assembled.
> Loading data...........
******** 11 datas are loaded.
> Simulator run.
 No. /  PC : INST   A    B    C    D    E    F    G    H   SR tar src dst imm
0000 / 0000: LD   000a 0d68 5fbf ef84 0000 0001 ef78 2e37  2   x   x   0    0
0001 / 0001: LDHI 000a 0d68 00bf ef84 0000 0001 ef78 2e37  2   x   x   2    0
0002 / 0002: LDLI 000a 0d68 0001 ef84 0000 0001 ef78 2e37  2   x   x   2    1
0003 / 0003: LD   000a 0002 0001 ef84 0000 0001 ef78 2e37  2   x   x   1    1
0004 / 0004: BC   000a 0002 0001 ef84 0000 0001 ef78 2e37  2   x   x   6   15
0005 / 0005: SUBI 000a 0001 0001 ef84 0000 0001 ef78 2e37  2   1   x   1    1
                                         (中略)
0060 / 0013: JUMP 000a 0000 0064 0000 0000 0001 ef78 2e37  2   x   x   x   10
0061 / 0010: SUBI 000a 0000 0064 ffff 0000 0001 ef78 2e37  6   3   x   3    1
0062 / 0011: BC   000a 0000 0064 ffff 0000 0001 ef78 2e37  6   x   x   6   -6
0063 / 0005: SUBI 000a ffff 0064 ffff 0000 0001 ef78 2e37  6   1   x   1    1
0064 / 0006: BC   000a ffff 0064 ffff 0000 0001 ef78 2e37  2   x   x   6   13
0065 / 0019: ST   000a ffff 0064 ffff 0000 0001 ef78 2e37  2   x   x   2    2
0066 / 0020: HALT 000a ffff 0064 ffff 0000 0001 ef78 2e37  2   x   x   x    0
******** 67 instructions are simulated.
> Output data file.
******** Output 11 datas, successfuly.
> Output mem file.
11_010_00_000_000_000   //            LD      A       0
11_001_00_000_000_010   //            LDHI    C       0
11_000_00_000_001_010   //            LDLI    C       1
11_010_00_000_001_001   //            LD      B       1
11_100_00_001_111_110   //            IF(BC)  NEG     END
10_001_00_001_001_001   //    MULTI:  SUBI    B       B       1
11_100_00_001_101_110   //            IF(BC)  NEG     END
10_000_00_000_010_011   //            ADDI    D       C       0
11_100_00_000_110_110   //            IF(BC)  NEG     MINUS
01_000_01_000_000_010   //            SUB     C       A       A
10_001_00_001_011_011   //    LP1:    SUBI    D       D       1
11_100_11_111_010_110   //            IF(BC)  NEG     MULTI
01_000_00_000_010_010   //            ADD     C       C       A
00_010_00_000_001_010   //            JUMP    LP1
01_000_01_000_000_010   //    MINUS:  SUB     C       A       A
01_000_01_000_010_010   //    LP2:    SUB     C       C       A
10_000_00_001_011_011   //            ADDI    D       D       1
11_100_11_111_110_110   //            IF(BC)  NEG     LP2
00_010_00_000_000_101   //            JUMP    MULTI
11_011_00_000_010_010   //    END:    ST      C       2
00_001_00_000_000_000   //            HALT
******** Output machine code, successfuly.
> Output coe file.
******** Output COE file, successfuly.
> Output bin file.
******** Output BIN file, successfuly.
> Post-processing.
```

ソースファイル

シミュレーションオプション

シミュレーション内容

図13：仮想 SOAR シミュレータの実行例

3　FPGA 上でのハード／ソフト･コラーニングシステムを用いた実装

3.1　実装内容

ハード／ソフト･コラーニングシステムでは、MONI プロセッサを FPGA に実装するための周辺モジュール(以下 MONI 周辺システム)があらかじめ用意されている。この MONI 周辺システムを利用することにより、MONI 以外のプロセッサを容易に FPGA へ実装することが可能であると考えられる。

実装した FPGA ボードは、Celoxica 社の RC100 ボードである。RC100 ボードには、Xilinx 社の 20 万システムゲートの Spartan II FPGA、同じく Xilinx 社の CoolRunner CPLD、また、FPGA 外部のメモリとして intel 社の 64 Mbit StrataFlash Flash memory を搭載している。

今回の実装では、MONI 周辺システムを利用することで、第一に、SOAR プロセッサ単体の動作確認を行うことを目的とする。すでに完成している MONI 周辺システムを用いるので、周辺モジュールの動作は保証されており、実機上でのデバッグを周辺モジュールとインタフェースと SOAR プロセッサ内部に限定することができる。第二に、ハード／ソフト･コラーニングシステム上で MONI 以外の異なるプロセッサの実装が可能であるかを検証する。検証により、MONI 周辺システムとの接続や実機上での実装が容易であることが分かれば、独自に設計したプロセッサの検証に MONI 周辺モジュールを用いることや、ハード／ソフト･コラーニングシステムにおいて MONI 以外のプロセッサを設計し得る。

3.2　SOAR の接続

今回の FPGA 上での動作検証では、MONI 周辺モジュールとの接続方法が重要となる。SOAR プロセッサのインタフェースを MONI プロセッサと同じように整え、MONI 周辺システムと接続し、RC100 の FPGA へ実装を行った。図 14に SOAR を MONI 周辺システムに接続した概観を示す。

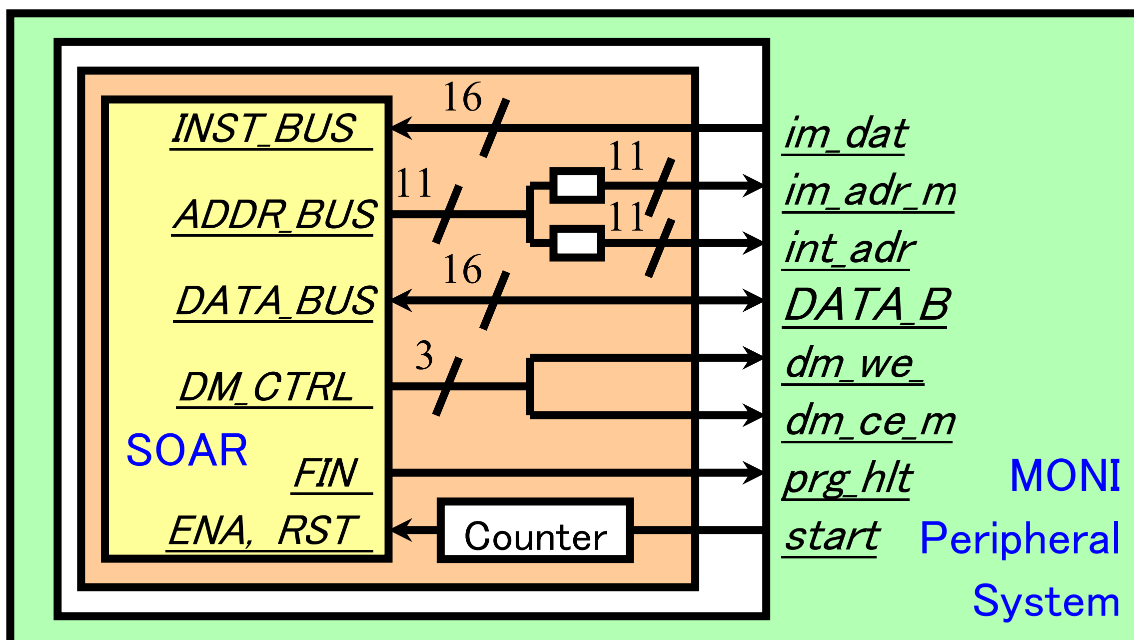

図14：MONI 周辺システムと SOAR の接続の様子

MONI プロセッサの仕様では、メモリのアドレス指定は命令用とデータ用で別々のポートが用意されている。SOAR プロセッサでは、アドレスの指定は ADDR_BUS の 1 ポートのみを使用し、クロックのレベルセンシティブ値(Low/High)で命令用とデータ用を切り替えて指定しているので、MONI

周辺システムに SOAR プロセッサを実装するには調節が必要である。SOAR プロセッサの ADDR_BUS ポートの出力に 2 つのレジスタを並列に接続し、クロックのエッジセンシティブ制御(立ち上がり／立ち下がり)ででそれぞれに値を書き込むことで、命令メモリアドレスとデータメモリアドレスの分離を図った。2 つのレジスタを MONI 周辺システムのそれぞれのアドレスポートと結び、アドレスの指定を行う。

また、データメモリへの制御信号も、コントロールイネーブル信号とライトイネーブル信号が、別々に用意されている。SOAR プロセッサでは、DM_CTRL がコントロールイネーブル、ライトイネーブル、リードイネーブルの 3 つの信号を出力しているので、その中のコントロールイネーブル、ライトイネーブル信号のみを MONI 周辺システムと結んだ。

最後に、動作開始のタイミング調節が必要である。MONI プロセッサでは、MONI 周辺システムから start ポートを通して動作開始の信号を受信し 6 クロック後にメモリへのアクセスを開始する。これは、メモリを含む MONI 周辺システムとのタイミングを合わせるための動作である。SOAR では、ENA がアクティブになると、次のクロックからメモリのアクセスを開始して命令を実行するので、MONI 周辺システムと間に 5 クロックを計測するカウンタを設け、6 クロック後に SOAR の ENA へアクティブ信号を送信する。

3.3 動作検証

MONI 周辺システムを用いて、RC100 ボード上で SOAR プロセッサを動作させ、正しく実行することを確認した。

検証には、N までの和、バブルソート、最大値検索のプログラムをテストプログラムとして用いた。検証で用いた各プログラムは、asm でアセンブルし sim でシミュレーションを行って設計したプログラムである。ボード上での SOAR プロセッサの実行方法は MONI プロセッサのボード上での実行手順と同様に行った[5]。これにより、MONI 以外のプロセッサを MONI 周辺システムに組み込むことで、FPGA 上で簡単に検証が行えることが確認された。

3.4 考察

第一の目的であった SOAR プロセッサの実機上での検証は、MONI 周辺システムを用いることで比較的容易に行えた。考察点として、実機検証を通じて理解した HDL 上でのシミュレーションとの違いを挙げる。

一点目として、動作開始時にメモリから読み出すアドレスや、プログラムを終了したことを示す FIN 信号を安定させておく必要あることを述べる。HDL シミュレータ上では、一定の間隔を置いてプログラム開始の信号をアクティブにしていたので、プロセッサは正常に動作を開始できていた。しかし、実機上では手動のスイッチによる入力であったため、クロックに同期しないランダムなタイミングでの信号となり、次のクロックまでに信号が安定せず、結果プロセッサが動作しないケースがあった。また、プログラムの終了を示す FIN 信号も、組み合わせ回路の出力としていたために、ゲート遅延によって発生した不正なアクティブ信号が出力されている現象が発生した。この解決策として、クロックに同期して信号の値を保持するレジスタを設けた。本来 FPGA は SRAM によって構成されているため、同期回路の設計を前提としたチップである。このため、FPGA に組み込む回路はできるだけクロック同期のものが望ましい。

二点目として、シミュレーションの重要性を認識した。実機上での検証は回路内部のレジスタ値が観測できない。このため、不正な動作を起こした場合はそのデバッグが非常に手間がかかる。このため、HDL シミュレーションを通して、些細なバグなどは早期に取り除いておく必要がある。HDL シミュレーションでの検証にはある程度時間がかかるが、実機での生じたバグの検証にはさらに時間がかかるので、より上流の設計過程での検証を再認識した。

第二の異なるプロセッサの検証では、MONI 以外のプロセッサを MONI 周辺システムに組み込

むことで、FPGA 上で簡単に検証が行えることが確認された。これにより、独自に定義し作成したプロセッサを、容易に FPGA ボード上に実装できることが分かった。今回行った動作検証手法をハード／ソフト・コラーニングシステムにおいても用いることで、ハードウェア学習に新しいフローが定義できる。すなわち、ユーザが独自にプロセッサを設計し検証を行う学習フローである。ユーザが独自にプロセッサを設計することで、今までのハード／ソフト・コラーニングシステムより、発展した協調学習が可能である。

問題点もいくつか見られた。まず、MPU の動作のタイミングである。プロセッサと MONI 周辺システムとのインタフェースに、カウンタを挿入することで解決できるが、プロセッサの動作開始時における命令読出しのタイミングを誤る可能性があるため、MONI 周辺システムに組み込む方が汎用的であると思われる。次に、データや命令、アドレスのバス化である。一般的なプロセッサではデータのやり取りは LSI 内部のシステムバスを用いることが多い。MONI システムでも同じようにバスアーキテクチャを採用し、FPGA 内のほかのモジュールとは、システムバスを使用してのデータのやり取りを行っている。バスを使用することで、システム内のデータのやり取りをシーケンサが一括して管理でき、モジュール間の同期が取りやすくなる。このような観点から、MONI 周辺システムのインタフェース部分でも、データの入力、出力ポートを 1 つにまとめ、データバスにつながるポートとするべきである。

4　FPGA 上での検証システムを用いた実装

SOAR プロセッサを中心とするコンピュータシステムを実現するために、独自に周辺モジュールを設計し FPGA ボード上で動作検証を行った。MONI 周辺システムでの実機検証により SOAR プロセッサ単独での動作は確認したため、今回の実装では、FPGA 外部からのデータの転送、システム全体の制御、共有バスアーキテクチャなど、新たなモジュールを含めたコンピュータシステム全体での動作検証を行う。実装環境は MONI 周辺システムでの実装時と同じ RC100 ボードを用いた。図 15に SOAR コンピュータシステムの全体図を示す。

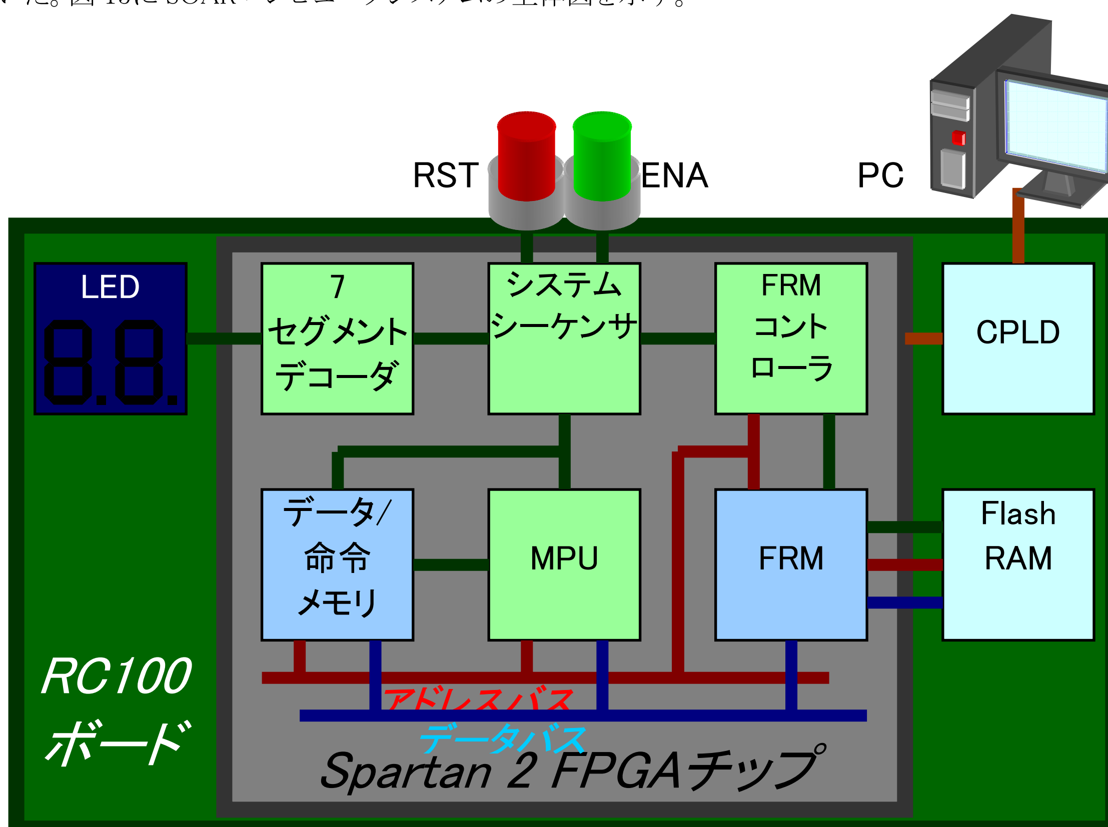

図15：SOAR コンピュータシステム

使用した RC100 ボードには以下のモジュールと装置が搭載されている[5]。今回のシステムでは、これらの機能も利用し動作検証を行った。

- CPLD
  ホストコンピュータの PC から FPGA チップに構成する回路情報をダウンロードする際に利用。
- Flash RAM
  FPGA ボードコンピュータで実行するプログラムとデータを格納しておく RAM。FPGA を構成する前にホストコンピュータの PC から書き込みを行っておく。
- 7 セグメント LED
  7 つの LED を点灯させることによって、デジタルな数値を表示可能。現在のシステム内部の状態を表示するために利用する。
- スイッチ
  RC100 ボード外部から、FPGA 内部を制御するためのスイッチ。アクティブ High の ENA スイッチとアクティブ Low の RST スイッチがある。

次に FPGA チップ上に展開するコンピュータシステムのモジュールを示す。

- システムシーケンサ
  システムに電源が投入され FPGA が構成されてから、プログラムを実行し、Flash RAM に実行結果を格納するまでの手順を管理するモジュール。内部にシステムの状態を表すレジスタを備え、その値によって各モジュールの制御信号を決定し送信する。
- MPU
  プログラムの実行を行う SOAR プロセッサ。
- FRM(Flash RAM Module)
  FPGA 外部の Flash RAM にアクセスし、内部のデータの読み出し、書き込みや、128KByte(1ブロック)の消去を行うモジュール。本研究室の中谷氏より提供[4]。
- FRM コントローラ
  必要なメモリのデータ量分だけ、FRM に読み書きさせる制御モジュール。
- データ／命令メモリ(メインメモリ)
  汎用プロセッサのメインメモリの役割を担っているモジュール。Xilinx が提供する Block RAM の IP コアによって構成されている。命令とデータのメモリをデュアルポートのメモリで実現した。メモリサイズは命令／データメモリそれぞれ 16bit×2048 語ずつ用意し、システム全体で 8KByte のメモリが使用可能である。
- 7 セグメントデコーダ
  7 セグメント LED に表示するデータをデコードするモジュール。

4.1 システムシーケンサ

システムシーケンサは、9 つの状態を持つ有限状態機械として実現した。また、今回設計したシーケンサは、単相クロックの立ち上がり同期で状態の遷移を行う。システムシーケンサは各状態において、ある特定の信号を受信することでクロックに同期して次の状態に遷移し、コンピュータシステム内の全モジュールに現在の状態での動作を制御する信号を送信する。図 16にシステムシーケンサの状態遷移図を示す。

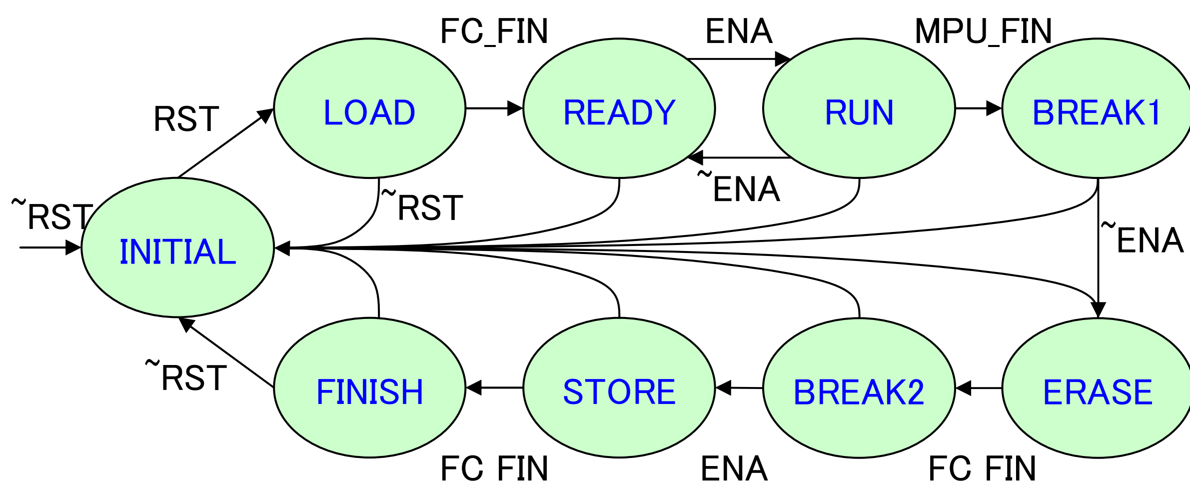

図16:システムシーケンサの状態遷移

システムシーケンサが示す状態は、内部のレジスタの値によって表現され FPGA コンピュータ全

体の動作状況を示したものである。以下に、システムシーケンサの状態遷移に沿って各状態でのシステム全体の様子を図示しながら詳しく説明する。

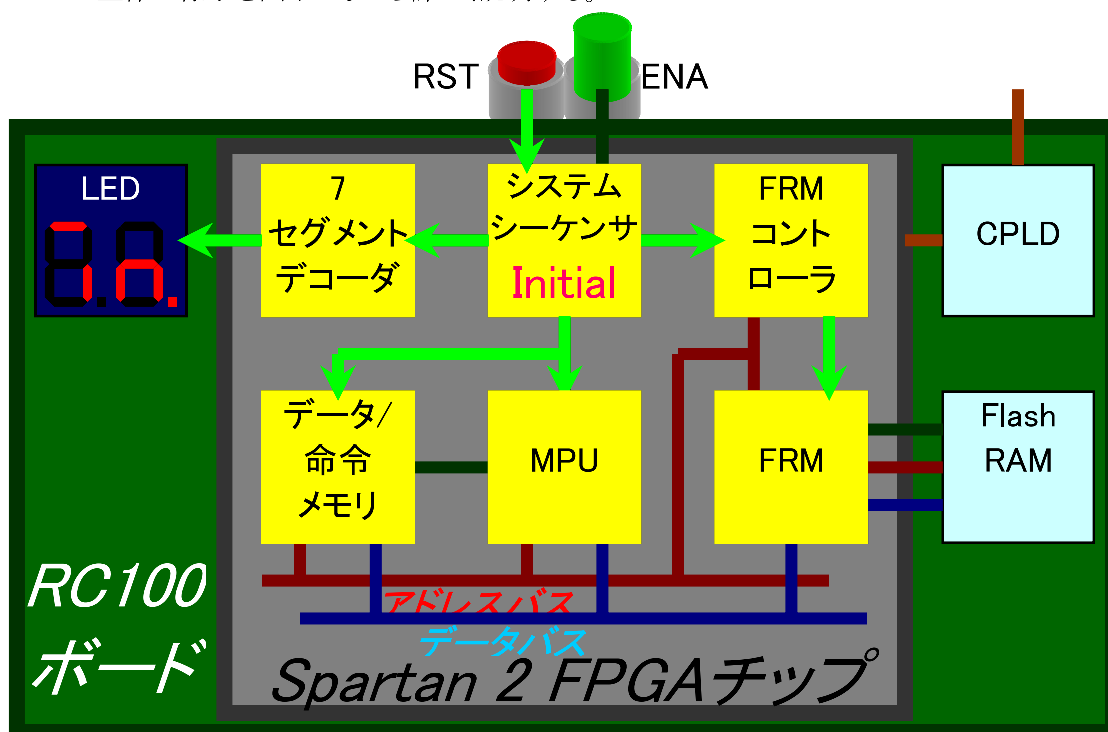

図17:INITIAL 時の動作

- INITIAL 状態
  システム全体にリセットがかかっている状態。すべてのモジュールにおいて、内部のレジスタが初期化される。FPGA 外部の RST スイッチを Low にすることで、クロックに同期して全状態から遷移が可能である。図 17に外部スイッチの RST が Low となり、システム内部の書くモジュールにリセット信号が配信される様子を示す。

- LOAD 状態
  FPGA 外部の Flash RAM から FPGA 内部の Block RAM にデータを転送している状態。INITIAL 状態時に RST スイッチがノンアクティブになることで、クロックに同期してこの状態に切り替わる。

- 図 18に LOAD 状態での FPGA 外部の Flash RAM から FPGA 内部の Block RAM にデータが転送される様子を示す。

- READY 状態
MPU の実行が可能なことを示す状態。MPU のイネーブル信号をノンアクティブにすることで MPU の動作を停止させている。LOAD 状態時に Flash RAM への書き込みが終了したことを示す FC_FIN 信号がアクティブになることで、クロックに同期して遷移する。図 19 では READY 状態でのシステム全体の様子を示す。

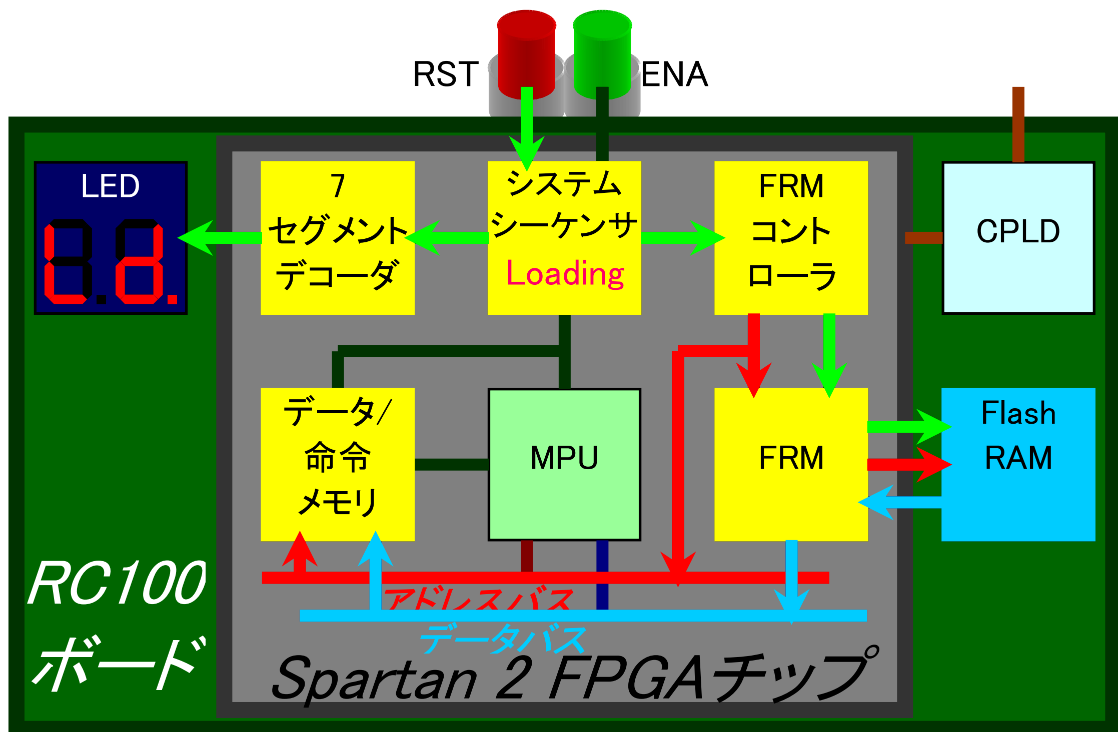

図18：LOAD 時の動作

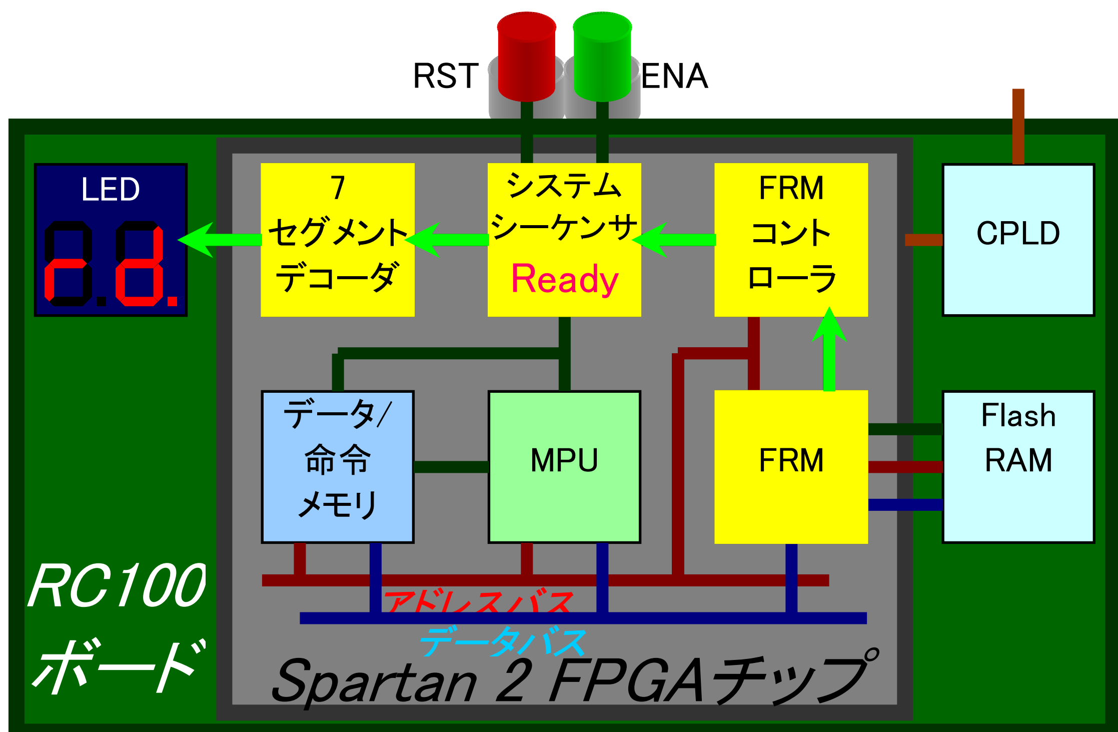

図19：READY 時の動作

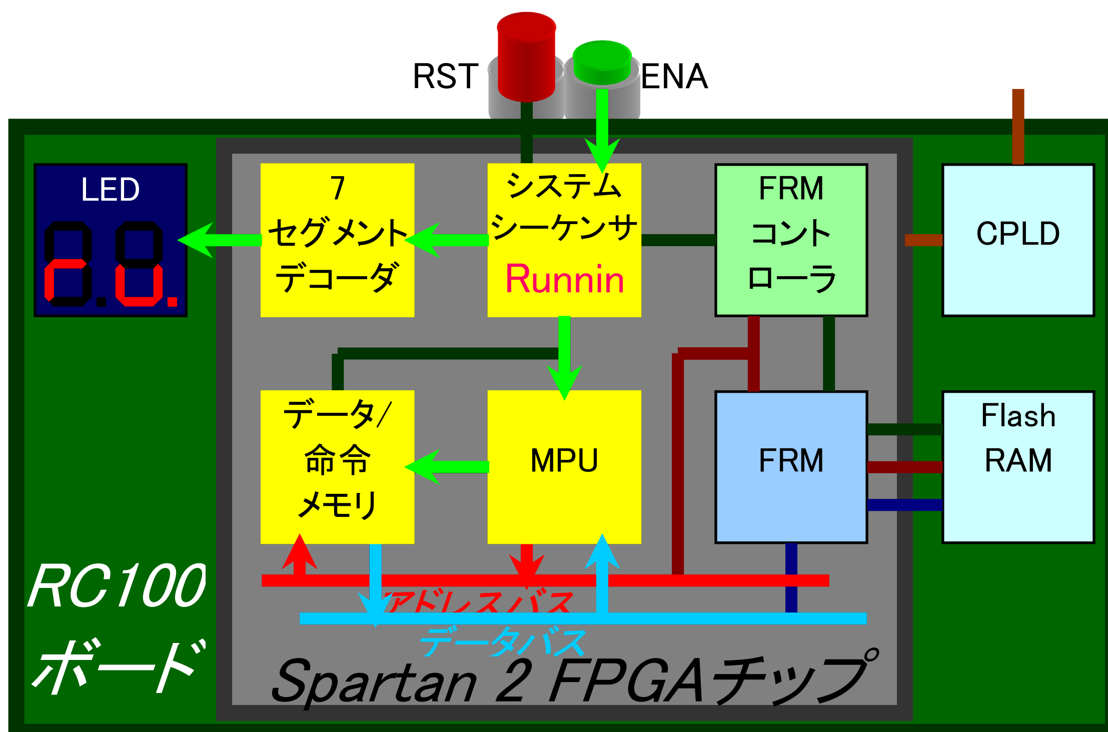

図20：RUN 時の動作

■ RUN
MPU が動作中であることを示す状態。READY 状態時に FPGA 外部の ENA スイッチをアクティブにすると、クロックに同期してこの状態に移り MPU の実行が始まる。MPU の実行が終了する前に、FPGA 外部の ENA スイッチをノンアクティブにすることで、クロックに同期して MPU の実行を一時中断することも可能である。このとき、システムは READY 状態に遷移する。図 20ではアドレスバスとデータバスを使用して MPU とメインメモリがやり取りし、プログラムを実行している様子を示している。

■ BREAK1
MPU の実行後にメモリの内部を観察するために設けた状態。実機上でのデバッグで使用した。RUN 状態の時に、MPU の実行終了を示す MPU_FIN 信号を受信すると、次のクロックでこの状態に移る。

- 図 21に BREAK1 状態でのシステム全体の様子を示す。

- ERASE
  FPGA 外部のメモリである Flash RAM 内のデータを消去している状態。Flash RAM にデータを書き込む際に、一度 Flash RAM 内のデータを消去しなければならない。BREAK1 状態時に FPGA 外部の ENA スイッチを Low にすることで、クロックに同期してこの状態に遷移する。図 22には ERASE 状態でのシステム全体の動作模様を示している。

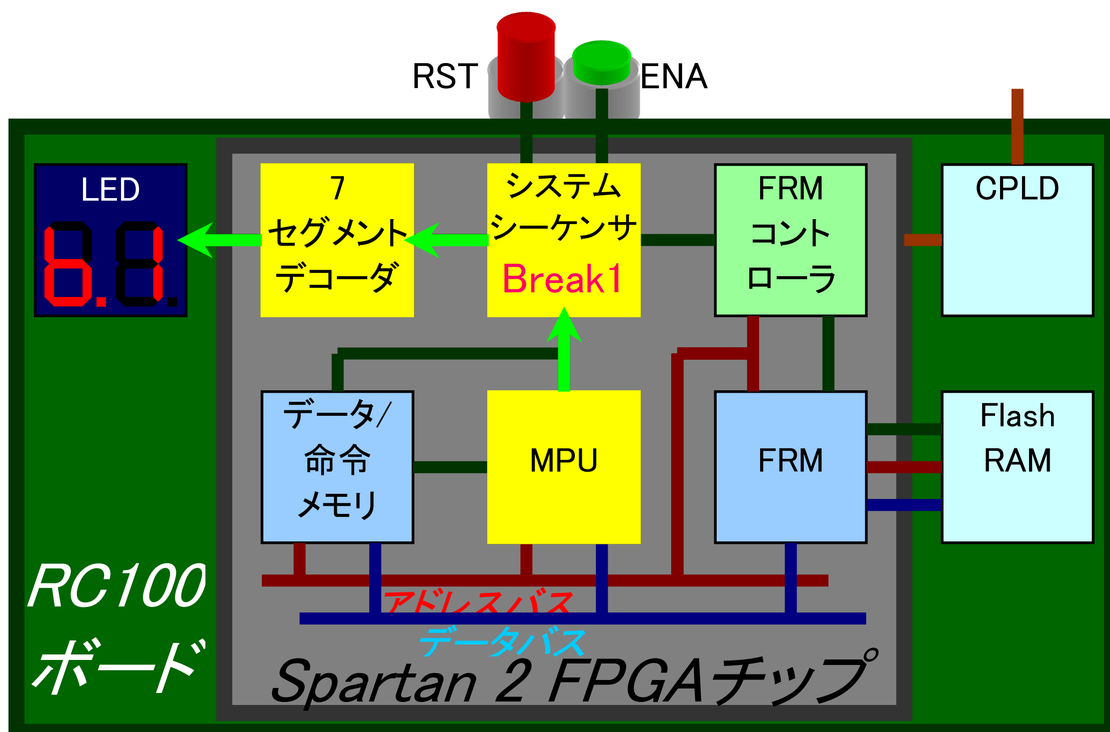

図21：BREAK1 時の動作

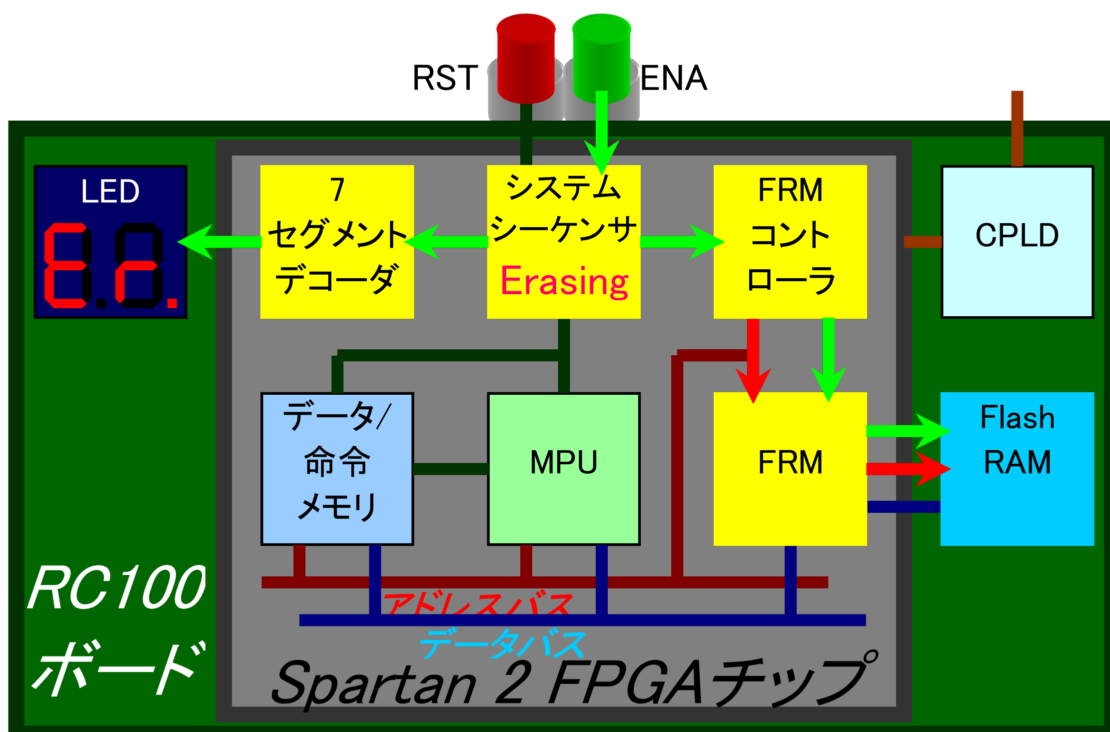

図22：ERASE 時の動作

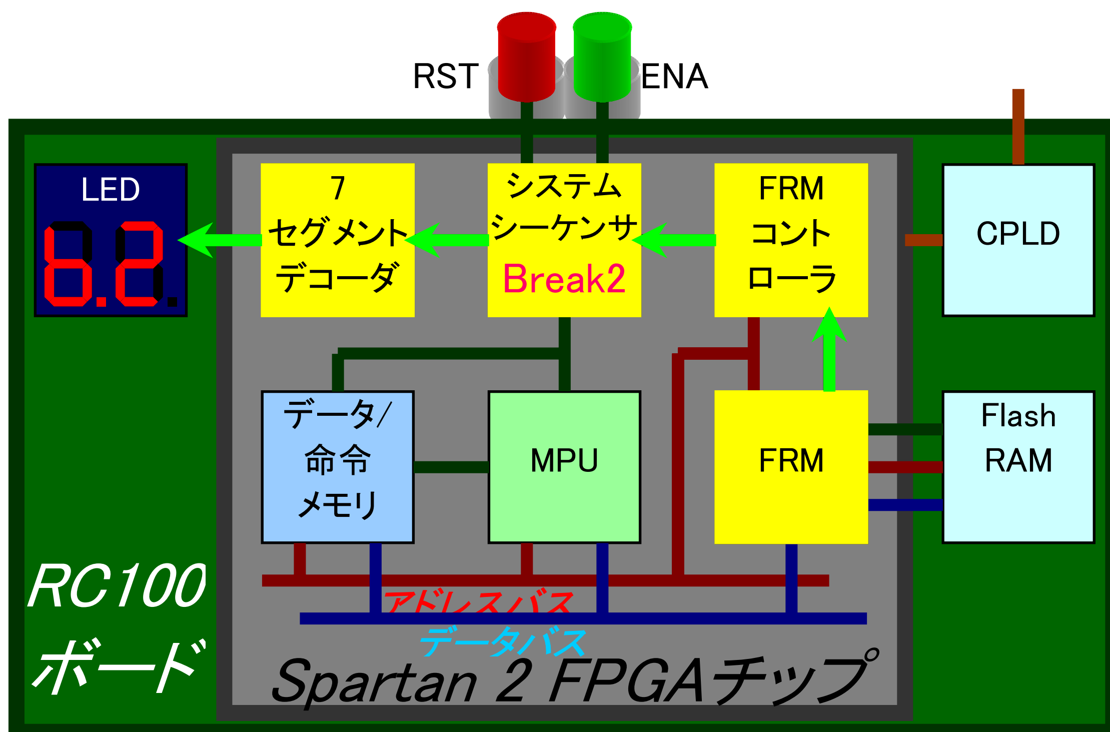

図23：BREAK2 時の動作

■ BREAK2
Flash RAM を消去後、一定時間待機するための状態。ERASE 状態のときに、データの消去が完了すれば FC_FIN がアクティブになり、次のクロックでこの状態に遷移する。この状態では、Flash RAM にデータを書き戻す準備をする。図 23 では、Flash RAM 内部のデータを消去し BREAK2 状態に遷移した様子を示している。

■ STORE
Flash RAM にデータを書き戻している状態。BREAK2 状態時に FPGA 外部の ENA スイッチを High にすることで、クロックに同期してこの状態に遷移する。

- 図 24 では、FPGA 内部の Block RAM から FPGA 外部の Flash RAM へデータを書き込んでいる様子を表している。

- FINISH
システムの動作が終了したことを示す。STORE 状態時にデータの書き戻しが完了すれば FC_FIN がアクティブになり、次のクロックでこの状態に遷移する。FPGA 外部の Flash RAM から命令とデータを読み込み、プログラムを実行して、再び Flash RAM にデータを書き戻すという手順を終了したことを意味する。図 25に FINISH 状態でのシステム全体の様子を示す。

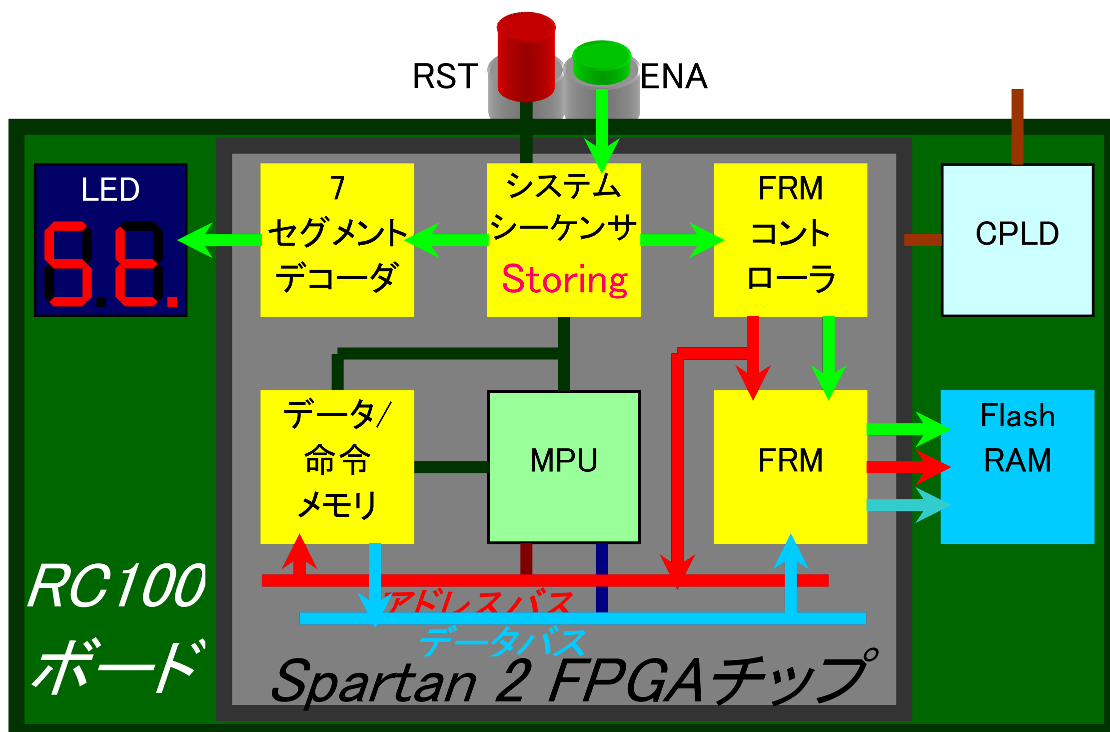

図24：STORE 時の動作

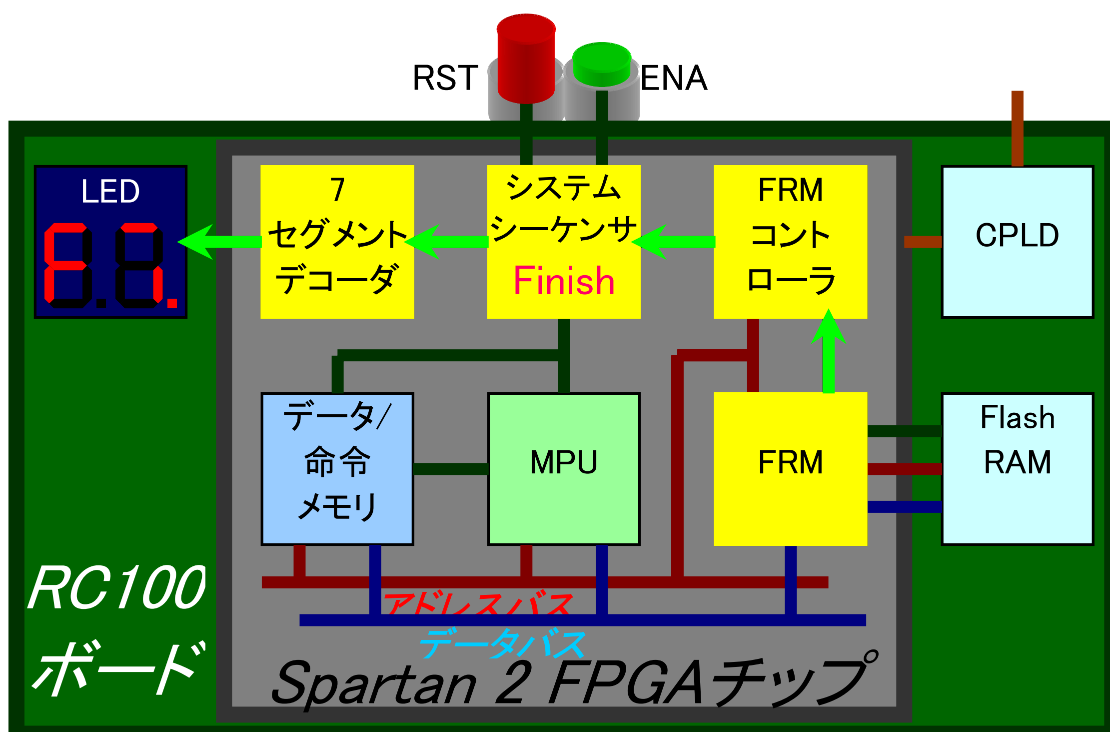

図25：FINISH 時の動作

4.2　FlashRAM Module コントローラ

Flash RAM Module(以下 FRM)コントローラは、FRM の制御を専門に担当するモジュールである。FRM は、本研究室の中谷氏によって作成されたモジュールで、read、write、erase の各命令を受信することで Flash RAM にアクセスし 1 データの読み出し、書き込み、もしくは 1 ブロックの消去を行う。本システムでは FRM を利用して FPGA 内部のシステムデータバスと外部の Flash RAM とのデータ転送を実現した。

Flash RAM 内のデータを操作する手順を、読み出しを例に述べる。

まず、システムシーケンサが FRM コントローラに対して Flash RAM からメモリへ必要データをコピーする命令を送信する。次に、この信号を受信した FRM コントローラが、命令の内容を自身の命令レジスタに保存し FRM に対して Flash RAM から 1 データを読み出す命令を発する。FRM は、データ読み出しの命令を実行し、成功すると FRM コントローラに読み出し完了の信号を送信する。読み出されたデータはシステムバスに一定時間流され、FPGA 内部のメインメモリに確実に保存されるのを待つ。ここで、メインメモリからの書き込み完了の信号は送信されない。最後に、FRM からデータ読み出し完了信号を受信した FRM コントローラは、次のアドレスのデータを転送するため、FRM コントローラにコマンドを送信する準備をする。実際のデータのやり取りは、このような FRM と FRM コントローラの対話的なやり取りにより、メインメモリのアドレス分、繰り返して実行される。

FRM コントローラもシステムシーケンサと同様にクロック同期式の有限状態機械で実現した。図 26に FRM コントローラの状態遷移図を示す。

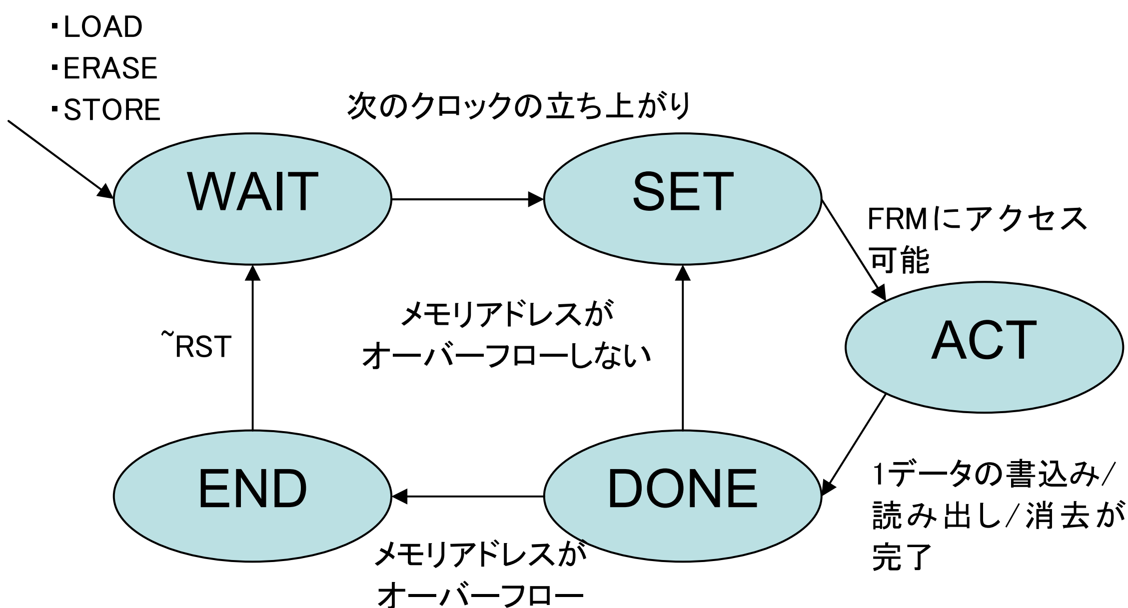

図26：FRM コントローラの状態遷移

4.3　クロックカウンタ

システムシーケンサ内に設置した MPU の動作クロック数をカウントするモジュール。MPU に動作開始の信号を送信してから、MPU からの終了信号を受信するまでのクロック数をカウントする。MPU の動作終了後 BREAK1 の状態時に 7 セグメント LED にカウントしたクロック数を表示する。このモジュールの機能により、正確な動的実行命令数のカウントが可能となる。

4.4　7 セグメントデコーダ

7 セグメントデコーダは、FPGA 外部にある 7 セグメント LED にデータを表示するための制御信号を生成するモジュールである。デコード可能な文字として、0～9 までのアラビア数字と 34 種類のアルファベットがある。これらの文字は、本システム内で専用の文字コードとして定義した。表 7に文字コード表を示す。

表7:7 セグメント LED 表示用文字コード表

文字	文字コード定数	LED 表示信号	文字	文字コード定数	LED 表示信号
0	SIGNAL0	011_1111	i	SIGNALi	000_0101
1	SIGNAL1	000_0110	J	SIGNALJ	001_1110
2	SIGNAL2	101_1011	j	SIGNALj	000_1101
3	SIGNAL3	100_1111	K	SIGNALK	111_0101
4	SIGNAL4	110_0110	k	SIGNALk	111_1000
5	SIGNAL5	110_1101	L	SIGNALL	011_1000
6	SIGNAL6	111_1101	l	SIGNALl	011_0000
7	SIGNAL7	010_0111	M	SIGNALM	011_0111
8	SIGNAL8	111_1111	n	SIGNALn	101_0100
9	SIGNAL9	110_1111	o	SIGNALo	101_1100
A	SIGNALA	111_0111	P	SIGNALP	111_0011
b	SIGNALb	111_1100	q	SIGNALq	110_0111
C	SIGNALC	011_1001	r	SIGNALr	101_0000
c	SIGNALc	101_1000	S	SIGNALS	110_0100
d	SIGNALd	101_1110	t	SIGNALt	111_1000
E	SIGNALE	111_1001	U	SIGNALU	011_1110
e	SIGNALe	111_1011	u	SIGNALu	001_1100
F	SIGNALF	111_0001	V	SIGNALV	111_0010
G	SIGNALG	011_1101	W	SIGNALW	111_1110
H	SIGNALH	111_0110	X	SIGNALX	011_1111
h	SIGNALh	111_0100	y	SIGNALy	110_1110
I	SIGNALI	000_1111	Z	SIGNALZ	101_1111

4.5　システムバス

本 FPGA システムではシステムバスとしてデータバスとアドレスバスを用意した。それぞれ、FPGA チップ内に実装し、モジュール間でデータのやり取りを行う際に利用する。データバスではメインメモリ、MPU、FRM 間のデータ、命令の伝送路として用い、アドレスバスは、メインメモリのアドレス指定のために用いる。FPGA 内に実装したシステムバスは、HDL 記述においてモジュールとしてではなく、配線として実装した。このため、システムバスに接続するモジュールでは 3 ステートバッファを通した出力と入力を接続し、バス用の入出力ポートを用意する必要がある。図 27に入出力ポートとバスの接続の様子を示す。

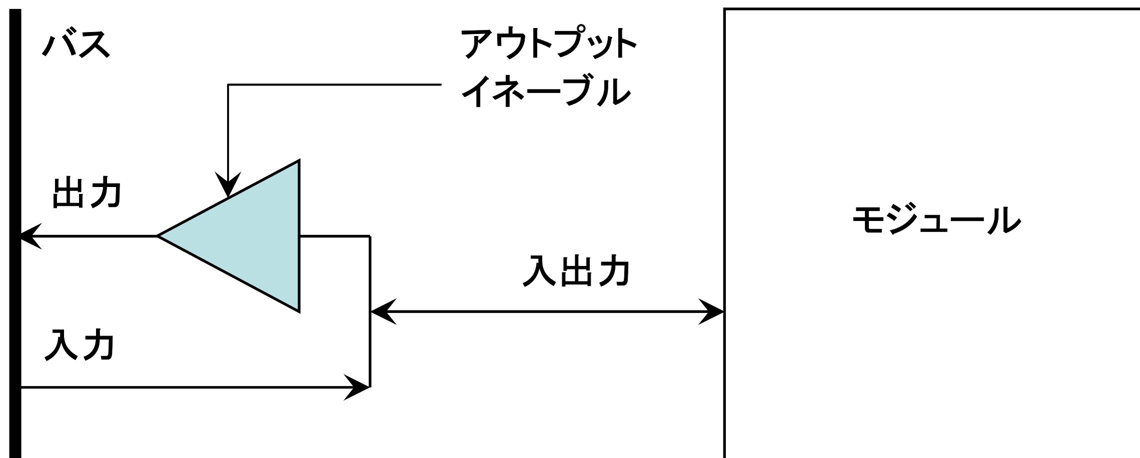

図27：入出力ポートとバスの接続

図 27に示すように、モジュールからデータの出力がない場合にはアウトプットイネーブルを Low にセットし、3 ステートバッファによってシステムバスに余分なデータが放出されないようにする。システムバスには、他のモジュールも接続されているため、モジュール自身が動作していない場合は、アウトプットイネーブルは Low にセットし、データの衝突を避ける。

4.6 システムの性能評価

作成したコンピュータシステムの回路規模と性能評価を、表 8と表 9に示す。表 8と表 9では、第 2 章に示した SOAR プロセッサの性能評価も、システム内の 1 モジュールとして項目に載せた。また、表 8と表 9には、今回作成したモジュールである、システムシーケンサ、FRM コントローラ、7 セグメントデコーダ、クロックカウンタ、SOAR プロセッサ、ALL について触れている。

表8：コンピュータシステムの回路規模

モジュール	HDL 記述量(行)	スライス数	LUT	ゲート数	FPGA 使用率
システムシーケンサ	122	19	36	308	0.154%
FRM コントローラ	249	52	69	1088	0.544%
7 セグメントデコーダ	32	4	7	42	0.021%
クロックカウンタ	12	1	16	448	0.224%
SOAR MPU	120	461	799	6822	3.411%
ALL		677	1052	155911	77.956%

表9：コンピュータシステムの性能評価

モジュール	パス (元→先)	最大動作周波数 (MHz)	最大遅延 (ns)	回路遅延 (ns)	配線遅延 (ns)
システムシーケンサ	STATE→FC_ENA	145.391	6.878	3.272 (47.6%)	3.606(52.4%)
FRM コントローラ	ENA→FRM_ADDR	77.845	12.846	3.731 (29.0%)	9.115 (71.0%)
7 セグメントデコーダ	CHR→SIG	103.029	9.706	5.992 (61.7%)	3.714 (38.3%)
クロックカウンタ	CNT→CNT	183.016	5.464	4.592 (84.0%)	0.872 (16.0%)
SOAR MPU	INST→ADDR→RF	23.014	43.451	15.384 (35.41%)	28.067 (64.59%)

4.7　SOAR プロセッサの動作検証

複数のサンプルプログラムを用いて、FPGA 上で SOAR プロセッサの動作検証を行った。プログラムの実行結果を表 10に示す。検証に用いたプログラムは 4 種類のソートプログラム、乗算、圧縮プログラムのランレングスエンコーディング、グレースケール画像を白黒画像に変換する二値化プログラムなど 10 数のテストプログラムである。これらは、いずれも、asm と sim によりアセンブルと命令レベルシミュレートを行い作成した。動的命令数は、システムシーケンサに内蔵したクロックカウンタによって実機上で計測した値である。

表 10に示すプログラムは、すべて FPGA 上で実行し正しい結果を得た。静的命令数ではランレングスエンコーディングが 140 で最大となった。また、動的命令数では、累乗算の 118,175 が最大である。実行内容で注目すべき項目は、二値化のプログラムである。今回の検証ではデータメモリとして 2KByte の Block RAM しか実装できず、50×34 という小さい画像でしか実行できなかった。

表10：FPGA ボード上での SOAR プロセッサ実行結果

プログラム	静的命令数	動的命令数	使用レジスタ数	実行内容
バブルソート	18	599	7	10 データの並び替え
選択ソート	22	415	7	
挿入ソート	17	149	8	
シェルソート	26	226	8	
最大値検索	13	57	5	10 データの最大値
N までの和	7	34	2	10 までの和
最大公約数	15	34	3	288 と 54 の最大公約数
乗算(自然数限定)	9	211	3	12×15
乗算(整数)	13	212	3	
累乗算	21	118175	4	3 の 10 乗
ランレングスエンコーディング	140	1958	8	5Byte のデータの圧縮
二値化	34	16936	7	50×34 の pgm 画像

4.8　考察

表 8を注目すると、システムを構成するそれぞれのモジュールの使用率は非常に低いが、SOAR プロセッサを含むシステム全体のゲート数は 15 万ゲートと Spartan II FPGA チップの約 8 割を使用している。これは、FPGA 内部に実現した IP の Block RAM が多くのゲート数を消費しているためであると考えられる。Block RAM は Xilinx の IP を使用して FPGA 内部に 4KByte のデュアルポートメモリとして実装した。

表 9に示す性能評価では、SOAR プロセッサの 23MHz を下回る最高動作周波数はないことから、コンピュータシステムがシステムクロック数を低下させることは回避できている。今後は、SOAR プロセッサを最適化し動作周波数を上げることで、コンピュータシステム全体のシステムクロックを向上させることが課題として挙げられる。

表 10では、作成したプログラム規模とその実行内容について示している。バブルソート、N までの和では、静的命令数が非常に小さく抑えられた。これは、分岐命令にステータスレジスタを用いるアーキテクチャを採用したためである。静的命令数では、ランレングスエンコーディングが 140 行と最大であったが、これは、SOAR 命令セットに関数呼び出しを実現する命令が存在しないため、同じ処理をプログラム内に複数記述したためである。しかし、関数呼び出しの命令を用意しても、これ以上のアセンブリプログラムは、人間の手によるデバッグが困難となる。命令レベルのシミュレーションを用いても、1 命令ずつレジスタをトレースする手法では非常に設計効率は悪い。このため、

より、高度なプログラムの設計には C コンパイラなどを作成し、高級言語での設計を行う必要がある。動的命令回数では、累乗算のプログラムが群を抜いて多くなった。これは、乗算の計算を累乗算分繰り返した結果である。二値化の画像データ用のメモリが小さい点では、IP を使用して FPGA 内部にメモリを実装する手法では、メモリ容量がチップのシステムゲート数に依存するため、今後は FPGA 外部に搭載されているメモリモジュールを利用するか、より容量の大きい FPGA を搭載する FPGA ボードを対象に実装することが課題となる。

5　プロセッサデバッガの検討

今回の実験では独自のプロセッサ SOAR を設計し FPGA ボード上に実装して動作検証を行った。SOAR プロセッサの検証方法として、ハード／ソフト・コラーニングシステムの MONI 周辺モジュールを用いた検証と、独自に作成した周辺モジュールとを組合せた検証を行った。これら 2 つの検証において、以下の問題点が浮かび上がった。

- FPGA ボード上でのデバッグが困難。LED でしか内部レジスタの確認ができない。
- スイッチの数が限られ、外部からの操作性が悪い。
- 論理合成に時間がかかる。少しのデバッグを行うときもシステム全体の論理合成が必要であり、動作確認に時間がかかる。
- 異なる命令セットアーキテクチャでのアセンブリプログラム開発環境が必要である。
- ハードウェアの設計では、HDL シミュレータでは波形での確認しかできず、直感的に理解しにくい。
- バス内のデータの流れが観測できない。不定値が流れた場合の動作が予測できない。

これらの問題点に対して、次節に示すプロセッサデバッガと汎用アセンブラ・汎用シミュレータの設計を検討する。これらのツールを提供することで、上記の問題点の解決を図る。

5.1　プロセッサデバッガ

実機上の FPGA 内部の信号を動作中に観測し、ある時刻でのすべてのレジスタの値が確認できるものとする。また、プロセッサに供給されるクロックをストップさせるブレーク回路を盛り込むことで、プロセッサのフェーズ実行や、1 命令ごとの実行、あるいは、停止が可能で、停止中にレジスタの値の変更もできるようにする。このような機能を持つことにより、ハードウェアデバッガの機能を持たせる。また、PC 上での命令シミュレータと協調して動作させることで、シミュレーション結果と実機での動作結果とを比較しながらデバッグを進められる。実機とシミュレーションの実行結果を比較することは、プログラムによるアルゴリズムのバグと、ハードウェアでの制約上でのバグとの分離が図られる。

5.2　汎用アセンブラ・汎用シミュレータ

ユーザが設計した命令セットの一覧表と対応する機械語、命令の動作を記述したテーブルを与えることにより、異なる命令セットアーキテクチャのアセンブルとシミュレーションを行う。これにより、ハード／ソフト・コラーニングシステム上で、ユーザが独自に MPU を設計し検証することが可能となり、より発展的な協調学習を行うことができる。

汎用アセンブラ・シミュレータでは、命令セットと機械語の一覧表を読み込むことで、専用のプロセッサようのアセンブリ記述ものアセンブルも可能とし、さらに、命令とその動作内容の一覧表を参照することで、汎用的な命令レベルシミュレーションの実行も可能としている。現段階では、詳細な仕様は決定していない。今後、関連研究を調査し、徐々に仕様を決定する。

5.3　ハード／ソフト・コラーニングシステムの拡張の検討

以上のような問題点を踏まえハード／ソフト・コラーニングシステムの拡張の検討を行った。図 28 に拡張後のハード／ソフト・コラーニングシステムの全体の学習体系を示す。

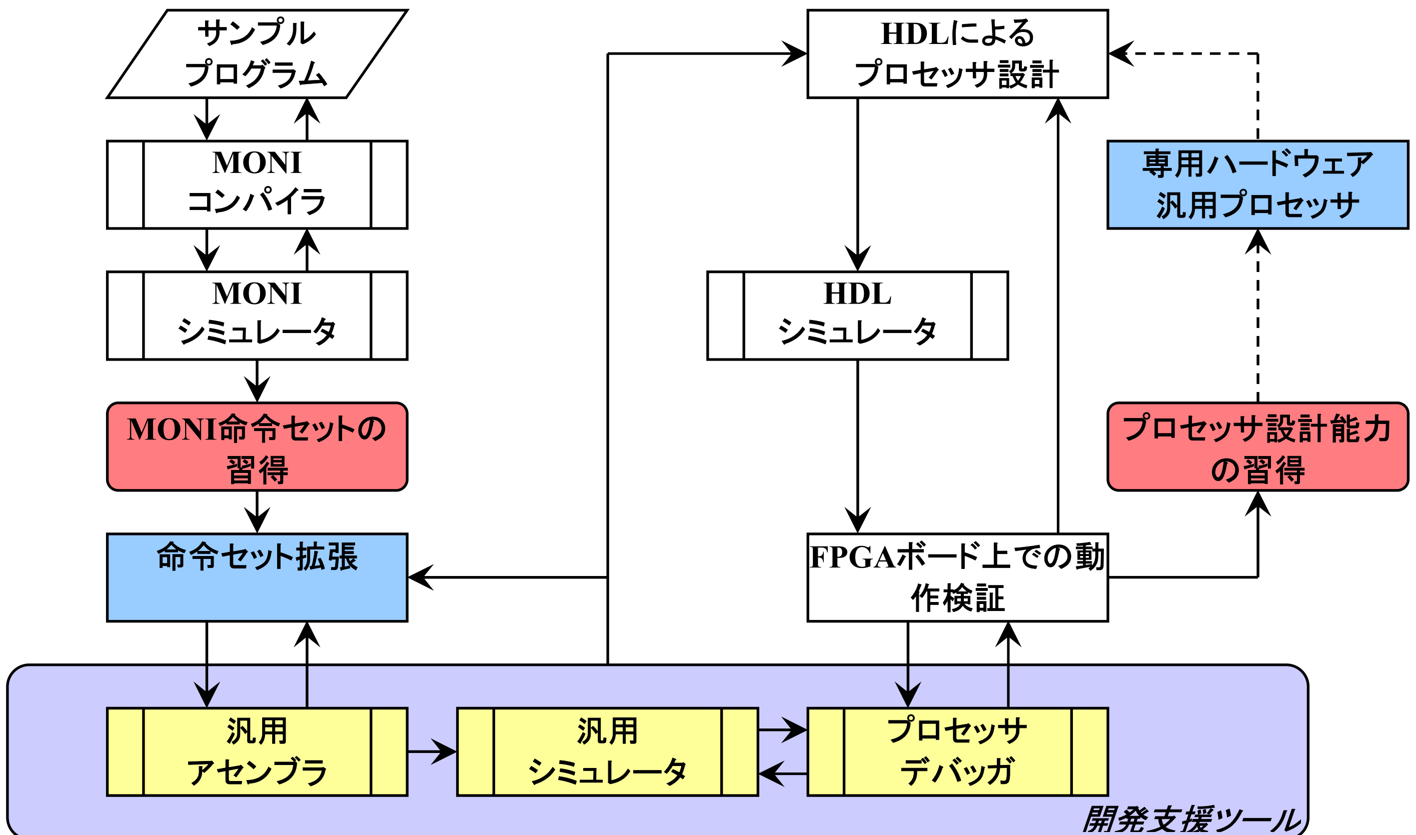

図28：拡張後のハード／ソフト・コラーニングシステムの学習体系

システムの特徴として以下の点を挙げる。

- ホストコンピュータ側から制御を可能とした学習支援ツールを提供する。
- 実装する FPGA 内部の状態を、提供する学習支援ツール上で観測できる。
- プロセッサデバッガと汎用シミュレータとを連動させることにより、アセンブリプログラミングとハードウェアデバッグを協調して進められる。また、シミュレーション結果と FPGA ボード上での実行結果を比較することで、ハードウェアの設計ミスとプログラムのバグを分離して構造的な設計手法により開発が進められる。基本命令セットの MONI は、必要最低限の命令のみを提供し、汎用アセンブラと汎用シミュレータを用いることによって独自に拡張できる。

5.4　今後の課題

上記の MPU の設計を通して、次のステップのハード／ソフト・コラーニングシステムでは、汎用アセンブラ・シミュレータの開発が必要であると考えられる。また、FPGA 上に実装してからのデバッグとシミュレーション機能も求められる。今後、これらの機能を開発し学習ツールとしてまとめることで、命令レベルのシミュレーションから、ハードウェア上でのデバッグまでを統合して学習できるシ

ステムの体系をまとめていく。

まず、FPGA ボード上からホストコンピュータへのデータの送受信を確立することが必要である。

6　おわりに

本論文では、単一サイクルマイクロプロセッサの設計とその動作検証を行った。設計は Verilog HDL による RTL 記述で行い、動作検証は RC100 ボードの FPGA 上に実装することで行った。動作検証の手法として、ハード／ソフト・コラーニングシステム上での検証と、独自のコンピュータシステムによる検証を行った。

ハード／ソフト・コラーニングシステム上での検証では、MPU のインタフェースを調節することで、比較的容易に実装できた。しかし、コラーニングシステム上では、MONI のアセンブラと命令レベルシミュレータしか用意されておらず、アセンブリプログラミングを行うためには MPU 専用のアセンブラと命令レベルシミュレータを作成しなければならないことが明らかになった。

独自のコンピュータシステムによる検証では、MPU の動作検証に、二値化やランレングスエンコーディング他、10 以上のプログラムを実行し、正常に結果が得られることを確認した。しかし、実機での検証には、ボード上の 7 セグメント LED でしか内部レジスタの値を確認する方法がないなど、デバッグ手法が限られ動作検証に非常に時間がかかった。

これらの点を踏まえ、ハード／ソフト・コラーニングシステムの次のステップとして、プロセッサデバッガと汎用アセンブラ・汎用シミュレータを提供することを提案した。これにより、MONI 命令セットの拡張を行うことでハードウェアとソフトウェアの協調学習を行い、今までよりも、さらに本質的な学習が可能となる。これらの学習体系を、拡張ハード／ソフト・コラーニングシステムとして考案し、本論文でまとめた。

謝辞

本研究の機会を与えてくださり、ご指導いただきました山崎勝弘教授、小柳滋教授に深く感謝します。また、本研究に関して貴重な助言、ご意見をいただきました、中谷氏及び、高性能計算研究室の皆様に心より感謝いたします。

最後に、本研究において多くのご指導、ご協力をいただきましたコンピュータシステム研究室の中村氏に深く感謝いたします。

参考文献

[1] 中村浩一郎：FPGA ボードコンピュータシステムの開発, 立命館大学理工学部卒業論文, 2004.

[2] 古川達久：マルチサイクル・パイプライン方式による教育用マイクロプロセッサの設計と検証, 立命館大学理工学部卒業論文, 2003.

[3] 池田修久：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, 立命館大学理工学研究科修士論文, 2004.

[4] 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可変なプロセッサシミュレータの設計と試作, 立命館大学理工学研究科修士論文, 2004.

[5] 池田修久, 中村浩一郎, Tran So Cong：RC100 を用いた FPGA ボードコンピュータ 設計仕様書 Ver0.7.3, 立命館大学理工学研究科 高性能計算研究室・コンピュータシステム研究室, 2004.2.5.

[6] 中谷嵩之：FPGA を用いたプロセッサ検証システムの設計と実装, 立命館大学理工学部卒業論文, 2005.

[7] 中谷嵩之：FlashRAM module manual, 立命館大学高性能計算研究室, 2004.

[8] John L. Hennessy, David A. Patterson 著, 成田光彰 訳：コンピュータの構成と設計(上)(下), 日経 BP 社, 1999.

[9] 中森章 著：マイクロプロセッサ・アーキテクチャ入門, CQ 出版, 2004.4.1.

[10] 小林優 著：改訂・入門 Verilog HDL 記述―ハードウェア記述言語の速習&実践, CQ 出版, 2004.

[11] 濱川圭弘 編著：半導体デバイス工学, オーム社, 200.12.20

[12] 社会人向け VLSI 設計セミナー レクチャー用マニュアル, 立命館大学 VLSI センター, 2004.

[13] 江田努：第 1 章 LSI の応用, LSI 応用工学 講義資料, 立命館大学 MELPEC 講座, 2004.

[14] Jan Gray: Hands-on Computer Architecture - Teaching Processor and Integrated System Design with FPGAs, Gray Research LLC, P.O. Box 6156, Bellevue, WA, 98008

[15] 細川晃平, 松本克裕, 中村裕一：FPGA エミュレータを利用したカスタムプロセッサ向け高機能デバッガ, 第 4 回リコンフィギャラシステム研究会論文集, pp.145-151, 2004.

[16] 児玉祐悦, 片下敏宏, 佐谷野健二：リコンフィギャラブルシステム REX への並列計算機 EM-X の実装, 情報処理学会計算機アーキテクチャ研究会, 2003.3.11

[17] 谷川一哉, 吉田哲生, 児島彰, 弘中哲生, 吉田典可：PARS アーキテクチャの詳細設計に関する考察, 情報処理学会計算機アーキテクチャ研究会 144-6, 2001.7.25

[18] 天野英晴:リコンリギャラブルシステム最新情報 Recent activites on Reconfigurable System, SACSIS, 2004.

[19] 天野英晴:マルチコンテキストデバイスを用いた動的適応型ハードウェアの提案, 情報処理学会研究報告, 2002-ARC-150, Vol.2002, No.112, pp.59-64, 2002.11.28.

[20] 北岡稔朗, 天野英晴, 安生健一朗:DRP 上での仮想ハードウェア実現のための構成情報の圧縮, 第 1 回リコンフィギャラブルシステム研究会 論文集, pp.213-218, 2003.9.

[21] 天野英晴, 犬尾武, 紙弘和:DRP 上での仮想ハードウェア機構の検討, 電子情報通信学会技術研究報告, VLD2003-122, Vol.103, No.578, pp.47-52, 2004.1.

[22] 出口勝昭, 山田裕, 天野英晴. DRP 上でのウェーブレットフィルタの実装, 電子情報通信学会技術研究報告, VLD2002-142, Vol.102, No.609, pp.73-78, 2003.1.29.

[23] 出口勝昭, 阿部昌平, 安生健一朗, 粟島亨, 天野英晴:DRP-1 上への JPEG2000 の離散ウェーブレット変換器と算術符号器の実装, 第 4 回リコンフィギャラブルシステム研究会 論文集, pp.9, 2004.9.

[24] Fredrik Dahlgren and Per Stenström: On Reconfigurable On-chip Data Caches, in Proc of 24th ACM/IEEE International Symposium on Microarchitecture, pp.189-198, 1991.11.

[25] Katherine Compton and Scott Hauck: Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Surveys, Vol. 34, No. 2, pp.171-210, June 2002.