修士論文

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

氏名：大八木 睦
学籍番号：6124020041-3
指導教員：山崎勝弘教授
提出日：2004 年 2 月日

立命館大学大学院 理工学研究科 情報システム学専攻

内容梗概

本論文では、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の協調学習を行うハード/ソフト・コラーニングシステムの構成要素である、MONI プロセッサとアーキテクチャ可変なプロセッサシミュレータの設計について述べる。

プロセッサの設計は、MIPS のサブセットとして独自に定義した 16 ビット命令セット（MONI）に基づき、VerilogHDL によるマルチサイクルプロセッサ、及びパイプラインプロセッサの設計を行った。FPGA を対象としたゲートレベルシミュレーションにおいて性能比較を行った結果、動作周波数ではマルチサイクルの方が速くなり、回路規模ではパイプラインの方が大きくなった。これは、パイプラインのデータパスに問題があったためであり、パイプラインプロセッサの動作周波数の向上が今後の課題である。

アーキテクチャ可変なプロセッサシミュレータは、単一サイクル、マルチサイクル、及びパイプラインからプロセッサアーキテクチャを任意に選択可能であり、データパス、レジスタ、及びメモリの内容を逐一変化させたり、複数の実行モードを用意したりすることで、アセンブリプログラムがプロセッサでどのように実行されるのかを可視化した。レジスタやメモリの内容を表示するデバッグ機能のほかに、プログラム解析として総実行命令数、総クロック数、CPI、ストール回数、フォワーディング回数などを表示できるようにし、パフォーマンスデバッガとしても使用できる。また、同研究室の方に試作したシミュレータを使用してもらい、プロセッサの動作が良く分かるなどの意見を頂いてシミュレータの目的を達成した一方、画面構成、操作性、及び実行モードなどに改善の余地があることを確認できた。

目次

図目次

表目次

1　はじめに

1.1 研究の背景と目的

近年、LSI の大規模化により、マイクロプロセッサ、メモリ、ASIC、アナログ回路などをワンチップ化し、システムを構築するシステム LSI が多くの電子機器に組み込まれている。システム LSI は特に、マイクロプロセッサを搭載するようになったことが大きな特徴であり、これはつまりソフトウェアを組み込むことができるということである。システム LSI を搭載した商品サイクルが短くなる中で、開発にかけられる期間もこれまで以上に短くなる。製品毎に固有の LSI を開発するのではなく、プロセッサ上で動くソフトウェアを組み込み、機能に柔軟性を持たせることはこれからの LSI 開発にとって大切な技術である[1]。

システム LSI に搭載するプロセッサは PC や WS などのプロセッサのように汎用性を持たせる必要はなく、組込みシステムに特化した専用プロセッサであることが多い。上述したように、機能に柔軟性を持たせる設計が重要視される中、専用プロセッサを合成する技術は目覚しいものがある。ベースとなる命令セットを持つプロセッサに新たに命令を追加することで、プロセッサの機能を自由に変更できる Tensilica 社の XTENSA[2]、東芝の MeP[3][4]、また専用のプロセッサ仕様言語を用いてプロセッサの合成を行う Co-Ware 社の LISA[5]などがその例であり、それらはデジタル家電などに搭載され効果をあげている[6]。

このようにシステム LSI 開発においては、プロセッサとソフトウェアは切り離せない関係にあり、開発に携わる技術者にとってハードウェアとソフトウェア両方の知識は必要不可欠である。ハードウェア面ではプロセッサアーキテクチャの理解が大前提であり、それを基に実機を対象とした HDL によるハードウェア設計技術などが求められる。また、ソフトウェア面ではハードウェアと密接に関わりを持ったアセンブリレベルでのプログラミングや、高級言語によるプロセッサを意識したプログラミングの能力が必要である。そのためには大学教育において、ハードウェアとソフトウェアの関係を密にした教育が重要である。このような背景から、我々はプロセッサをテーマとしたハードウェアとソフトウェアの協調学習システムの開発を進めている。本研究は学習システムの構成要素である、アーキテクチャが可変なプロセッサシミュレータの設計を行う。

1.2 関連研究

教育用プロセッサを用いた計算機工学教育環境の開発は多くの大学で行われている。

（1） 教育用プロセッサを用いたシステム

- PICO²[7]

  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[8]

  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[9][10]

  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。

その他、九州工業大学の DLX-FPGA[11]、武蔵工業大学の MITEC-Ⅱ[12]、九州大学の QP-DLX[13]などがある。

（2） プロセッサシミュレータ

- Mikage[14]

  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[15]

  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[16]

  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。

- SimpleScalar[17]

  スーパースカラシミュレータ。命令フェッチ、命令発行機構、メモリアクセス機構、演算機構、その他ハードウェア機構の動作を忠実にシミュレートすることができる。また、使用者が独自の機構を C 言語でコーディングすることも可能である。

（3） ハード/ソフト協調学習システム

- 港[18]

  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、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[19]

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 研究概要

我々は HDL によるプロセッサ設計と FPGA ボード上での実機検証、及びソフトウェアシミュレータによる動作理解を組み合わせたシステムの開発を進めている。このシステムはオリジナルプロセッサ（MONI）を用いたハードウェア設計演習と、可観測性を重視したプロセッサシミュレータによる演習を融合させ、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である。ソフトウェアシミュレータだけでは分からないことをハードウェア設計で学び、ボードコンピュータだけではわからないプロセッサの内部動作を、シミュレータを通して学ぶことができる。また、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）を選択可能にしており、最も初期段階のプロセッサアーキテクチャから、現在のスーパースカラに代表される高速なプロセッサへの変遷を学習し、プロセッサアーキテクチャの理解に繋げる。学習者は、アーキテクチャ可変なプロセッサシミュレータを用いたソフトウェア開発、最適化コンパイラの設計、及び HDL によるプロセッサ設計を通して、ハードウェアとソフトウェアを学ぶ。

本研究では、主に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を行った。上述の 4 種類のアーキテクチャを任意に選択することが可能で、それぞれのプロセッサにおいて複数の実行モードを用意し、データパス、レジスタ、及びメモリの中身をそのつど変化させることにより、プロセッサの動作を理解させることが目的である。アーキテクチャを自由に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。

1.4 論文の構成

本論文では、ハード/ソフト・コラーニングシステムの構成要素であるプロセッサの設計と、アーキテクチャ可変なプロセッサシミュレータの設計について述べる。先ず、2 章ではハード/ソフト・コラーニングシステムの全体像について述べ、3 章では VerilogHDL によるマルチサイクル、及びパイプラインプロセッサの設計について述べる。4 章ではアーキテクチャ可変なプロセッサシミュレータの要求仕様とその機能、5 章でプロセッサシミュレータの設計環境とモジュール構成、6 章では設計したプロセッサシミュレータのテストと評価について述べる。最後に、7 章で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

2.1 システム概要

ハード/ソフト・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 HDL によるプロセッサ設計、及び設計したプロセッサを FPGA ボードコンピュータ[23]に搭載し、周辺回路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。

本システムの特徴をまとめると以下の通りである。

- 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
  - ソフトウェアシミュレータだけでは分からないことを、ハードウェア設計を通して学べる
  - ボードコンピュータだけでは分からないプロセッサの内部動作をシミュレータで観測できる
- プロセッサアーキテクチャを段階的に学習
  - なぜ今スーパースカラや VLIW なのか
  - どのようにプロセッサが発展してきたのか
- アーキテクチャを意識した上でのプログラミング演習
  - プロセッサを高速に動作させるプログラミングを学習
- 同じプログラムを異なるアーキテクチャで動作させ評価可能
  - プロセッサアーキテクチャの評価

本システムは 16 ビット教育用 MONI プロセッサ、命令セットシミュレータ、最適化コンパイラから構成される。次節ではそれら構成要素について述べる。

2.2 システム構成要素

2.2.1 MONI プロセッサ

MIPS[20]のサブセットとして定義した 16 ビット教育用マイクロプロセッサである。教育用を意識し、アセンブリプログラミングのし易さを考慮にいれた命令セットを持つ。単一サイクル、マルチサイクル、パイプライン、スーパースカラの 4 種類のアーキテクチャがある。これはマイクロプロセッサの初期段階である単一サイクル方式から、マルチサイ

クル、パイプライン方式を通して、現在のマイクロプロセッサの主流であるスーパースカラへと段階的に学習することで、プロセッサアーキテクチャの歴史を紐解きながら、その理解を深めようという意図からである。また、学習者は各プロセッサのコアとなる部分をHDL により設計し、ハードウェア設計の学習を行う。命令セットアーキテクチャ、及び設計を担当したマルチサイクル、パイプラインプロセッサの詳細は 3 章で述べる。

2.2.2 命令セットシミュレータ

MONI プロセッサを対象とした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を表示
- 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

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詳細は 4 章で述べる。

2.2.3 最適化コンパイラ

4 種類のプロセッサアーキテクチャの最適化コンパイラを設計する。これは、プロセッサアーキテクチャに最適化されたアセンブリプログラムを生成するコンパイラを設計することでプロセッサアーキテクチャの理解をより深めることと同時に、アーキテクチャを意識した高級言語によるプログラミング学習に役立つ。

2.3 ハードウェアとソフトウェアの協調学習

コラーニングシステムにおける学習方法の詳細について述べる。図 1 にコラーニングシステムの全体像を示す。

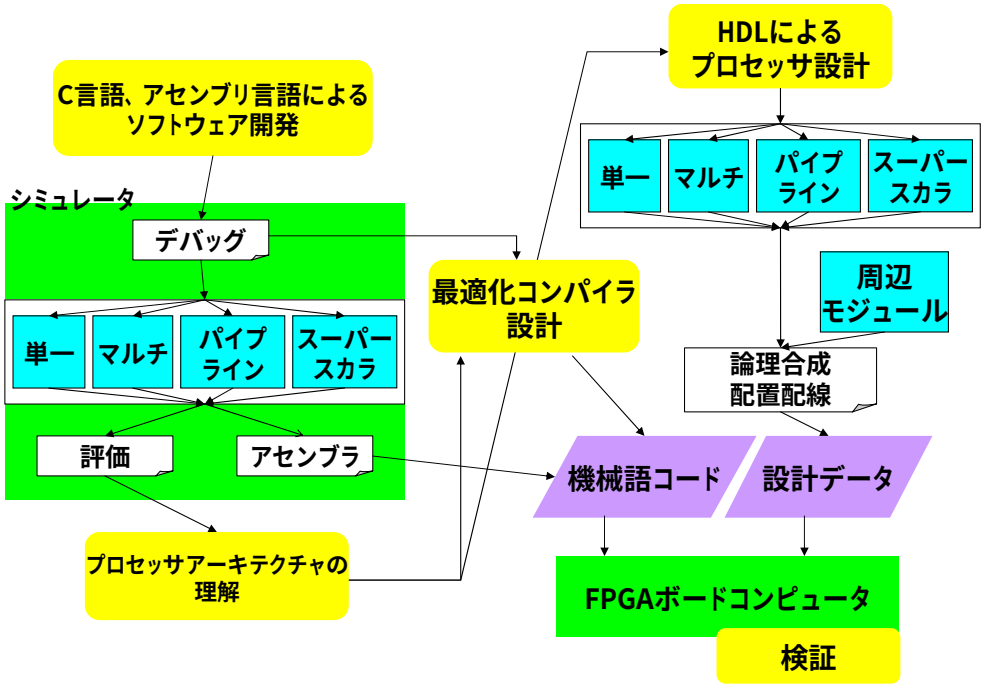


図 1：ハード/ソフト・コラーニングシステム

図の左側がソフトウェア学習、右側がハードウェア学習となる。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でデバッグを行う。また、アーキテクチャを変更することで設計したプログラムの評価を行い、最適化コンパイラ設計を通してソフトウェアの学習を行う。ハードウェア学習では HDL によるプロセッサコア設計を通して LSI 設計フローの学習を行う。また、実際にプロセッサの設計を行い、プロセッサアーキテクチャの理解度を確認し、シミュレータだけでは分からない遅延などのハードウェア設計特有の問題を理解する。

3 VerilogHDL によるプロセッサ設計

3.1 命令セットアーキテクチャ

ハード/ソフト・コラーニングシステムで使用するプロセッサの設計を行った。設計したプロセッサは 16 ビット固定長命令であり、教育用を意識して必要と思われる 43 命令を用意した。それぞれの命令はビット位置から以下のような 4 種類の命令形式に分類される。

● R 形式(ADD, SUB, OR, XOR, SLT, SGT, SLE, SGE, SEQ, SNE, SLL, SRL, SRA, NOT)

5	3	3	3	2
Opecode	Rs	Rt	Rd	Function

● I5 形式(ADD, SUBI, ANDI, ORI, XORI, SLTI, SGTI, SLEI, SGEI, SEQI, SNEI, LD, ST, SLLI, SRLI, SRAI)

5	3	3	5
Opecode	Rs	Rt	Immediate/Address

● I8 形式(LDHI, LDLI, BEQZ, BNEZ, PUSH, POP)

5	3	8
Opecode	Rs	Immediate/Address

● J 形式(JUMP, CALL, RETURN, HALT, NOP)

5	11
Opecode	Target absolute address

命令の上位 5 ビットは命令の動作を示す命令操作コードが配置されている。その命令操作コードにより、R 形式、I5 形式、I8 形式、J 形式の 4 種類に分類される。Rs、Rt はソースレジスタであり、演算の入力データを格納するレジスタを示す。Rd はディスティネーションレジスタであり、結果の格納先のレジスタを示す。I5、I8 の Immediate/Address は即値またはアドレスを示している。J 形式の Target absolute address 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 R 形式命令のより詳しい動作を示すコードである。

3.2 マルチサイクルプロセッサ

マルチサイクルプロセッサは、ひとつの命令を複数のステップに分け、それぞれのステップを 1 クロックで実行する方式である。そのため、1 クロックサイクルを短くすることが可能で、処理を高速化できる。また、1 命令につき同じ機能ユニットを 2 回以上使用できる

ため、ハードウェアコストの削減にもなる。図 2 に設計したマルチサイクルプロセッサのデータパスを示す。

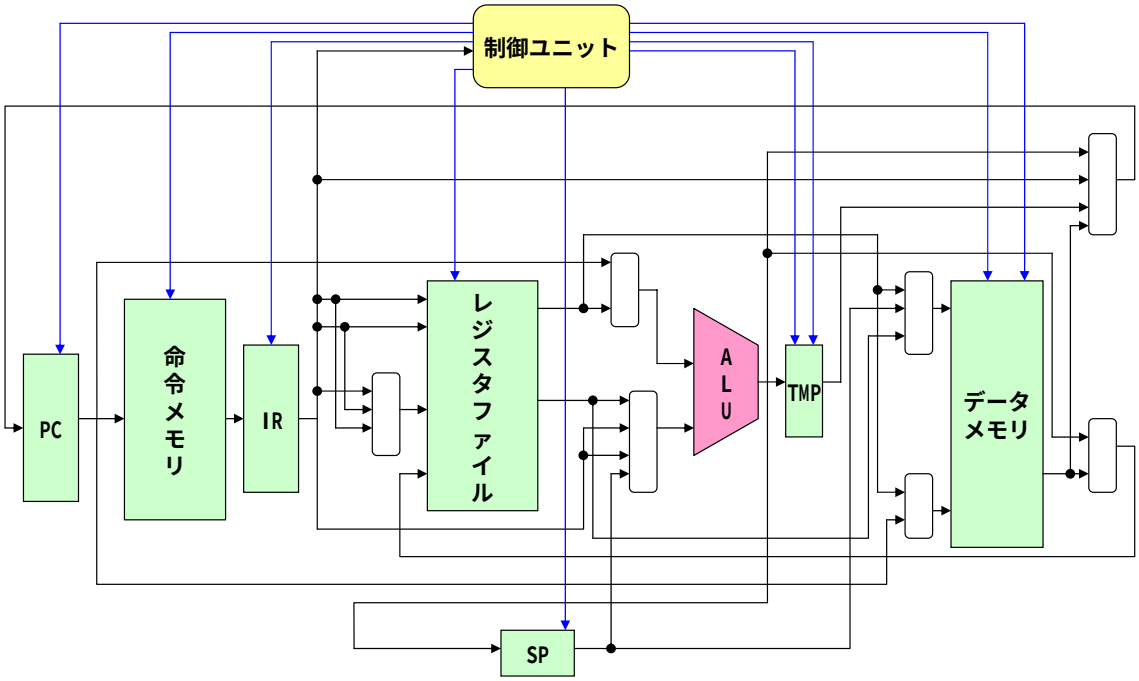


図 2：マルチサイクルプロセッサのデータパス

マルチサイクルプロセッサは以下のユニットで構成される。

- PC：プログラムカウンタ
- 命令メモリ、データメモリ
- IR：命令レジスタ
- レジスタファイル
- ALU：算術論理演算、分岐先アドレス計算、スタックポインタの加減算、ゼロ判定
- TMP：ALU で計算された分岐先アドレスを保存
- SP：スタックポインタ計算
- 制御ユニット

1 命令を複数のクロックで実行するため、実行中の命令を保存しておく命令レジスタ（IR）や分岐先のアドレスを格納しておく一時レジスタ（TMP）が必要となる。

また、マルチサイクルプロセッサの実行ステップは以下の 5 ステップであり、各ステップを 1 クロックサイクルで行う。

1. 命令フェッチ
2. 命令デコードとレジスタのフェッチ、及び分岐先アドレスの計算
3. 演算の実行、メモリ読み出し、pc 書き換え、スタックポインタ操作
4. レジスタ書き込み、及びメモリ書き込み

5. CALL 命令における PC 書き換え

表 1 にマルチサイクルの実行サイクルを示す。

表 1：マルチサイクルプロセッサの実行サイクル

命令／ステップ	1	2	3	4	5
R	IR ← IM[pc] pc++	$rs ←IR[11:9] $rt ←IR[8:6] TMP ←pc + imm	alu_out← Reg[$rs] 演算子 Reg[$rt]	Reg[$rd] ← alu_out	
I5			alu_out ← Reg[$rs] 演算子 imm	Reg[$rt] ←alu_out	
I8			Reg[$rs]← imm		
LD			dm_data ←DM[RF[$rs]]	Reg[$rt] ←dm_data	
ST			DM[Reg[$rt]] ←Reg[$rs]		
条件分岐			pc←TMP		
JUMP			pc←target_add		
POP			sp++, Reg[$rs] ←DM[$sp]		
PUSH			sp--	DM[sp] ←Reg[$rs]	
CALL			sp--	DM[sp]←pc	pc← target_add
RETURN			pc ← DM[sp], sp++		

注)IM：命令メモリ、DM：データメモリ、Reg：レジスタファイル、imm：即値

全ての命令においてステップ 1 の命令フェッチと、ステップ 2 の命令デコードとレジスタのフェッチは共通している。命令のフェッチ(ステップ 1)ではメモリから命令を読み出し、命令レジスタに格納する。これは上述したように、マルチサイクル方式では実行中の命令が終わるまで、その命令を保持する必要があるからである。さらにプログラムカウンタを次の命令のアドレスを指すように繰り上げる。命令のデコードとレジスタのフェッチ(ステップ 2)では、読み出すレジスタのレジスタ番号を振り分ける。また、分岐先のアドレスを計算する。この段階では現在実行中の命令がどの命令かはまだわかっていない。しかし、たとえ分岐命令でなくても分岐先アドレスを計算しておく。分岐先アドレスを前もって計算しておいても実害はないからである。ステップ 3 から命令によって処理が異なる。R 形式

と I5 形式ではステップ 3 で ALU において所定の演算を行い、ステップ 4 でレジスタに書き込む。I8 形式ではステップ 3 でレジスタに即値を書き込む。LD 命令はステップ 3 でメモリからデータを読み出し、ステップ 4 でレジスタに格納する。ST 命令はステップ 3 でレジスタから読み出した値をデータメモリに格納する。条件分岐命令ではステップ 3 で条件が成立した場合のみプログラムカウンタを書き換える。JUMP 命令はステップ 3 で無条件にプログラムカウンタを書き換える。POP 命令はステップ 3 でスタックポインタをインクリメントし、データメモリから読み出した値をレジスタに書き込む。PUSH 命令はステップ 3 でスタックポインタをデクリメントし、レジスタから読み出した値をデータメモリに書き込む。CALL 命令はステップ 3 でスタックポインタをデクリメントし、ステップ 4 でプログラムカウンタをデータメモリに格納し、ステップ 5 でプログラムカウンタをジャンプ先のターゲットアドレスに書き換える。RETURN 命令はステップ 3 でデータメモリから読み出した値をプログラムカウンタに書き込み、スタックポインタをインクリメントする。

3.3 パイプラインプロセッサ

パイプラインプロセッサはひとつの命令を複数ステップ（ステージ）に分け、連続した命令の各ステージを少しずつずらして同時並行的に実行する方式である。本研究で設計したのは単一サイクルプロセッサにパイプラインレジスタを挿入し、パイプライン化した 5 段パイプラインプロセッサである。以下に、各ステージの処理を示す。

ステージ 1（IF ステージ）：命令フェッチ

ステージ 2（ID ステージ）：命令でコード、条件分岐判定

ステージ 3（EXE ステージ）：ALU による演算

ステージ 4（MEM ステージ）：メモリアクセス

ステージ 5（WB ステージ）：ライトバック

パイプライン化することで命令のスループットが増大し、命令の全体のクロックサイクル数が大幅に減少するが、パイプラインステージを増やせば増やすほど高速化が期待できるというわけではない。実際には次のクロックサイクルで次の命令が実行できないという現象（パイプラインハザード）が起こるためである。パイプラインハザードには大きく分けて構造ハザード、データハザード、制御ハザードの 3 つがある。以下にそれぞれのハザードの説明とその回避方法について述べる。

- 構造ハザード

  一緒に実行される命令の組み合わせにハードウェアが対処できない場合に起こる。データメモリと命令メモリを分割し、十分な資源を用意することで回避する。

- データハザード

  前に実行している命令の演算結果を後の命令で使用する場合に発生する。演算結果のデータを先送り（フォワーディング）することで回避する。しかし、先行するロード命令

の直後に、ロードされた値を使用する命令があった場合には、フォワーディングだけでは回避できない。その場合は、パイプラインを停止（ストール）させる必要がある。

- 制御ハザード

分岐ハザードは制御ハザードとも呼ばれる。ある命令の実行に対する判断を、まだ実行中の他の命令の結果に基づいて下さなければならない場合に生じる。分岐命令に続く命令を正しく実行しようとすれば、分岐判定が下されるまでパイプラインをストールさせ続けなければならない。しかし、それではパイプラインの性能が極端に落ちてしまう。そのために、分岐命令の場合でも分岐しないと仮定して後続の命令をフェッチし続け、分岐が成立した場合にのみパイプラインをストールさせる方法を取る。分岐判定は本来の EXE ステージから ID ステージに前倒しし、分岐する場合に無駄になる命令を 1 命令に削減した。また、先行する命令の演算結果が分岐判定に必要な場合はその演算結果のフォワーディングを行う。

図 3 に設計したパイプラインプロセッサのデータパスを示す。

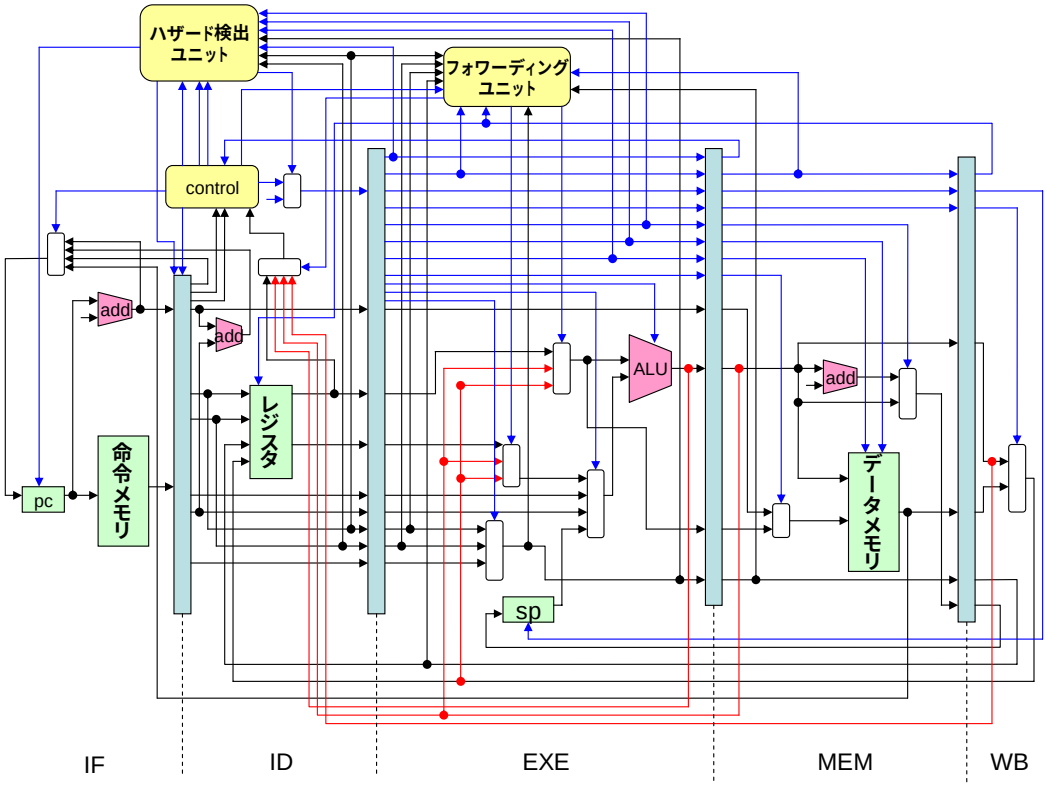


図 3：パイプラインプロセッサのデータパス

パイプラインを構成する主なユニットについて述べる。

- ハザード検出ユニット：ハザードを検出しパイプラインをストールさせる
- フォワーディングユニット：フォワーディングが必要かどうかの判定をする

3.4 プロセッサの性能評価

マルチサイクル、及びパイプラインプロセッサの設計は Xilinx 社の Foundation ISE を用いて行った。使用言語は VerilogHDL であり、論理合成ツールは XST Verilog を使用した。対象 FPGA は 20 万システムゲートの Spartan2 である。テストパターンには N までの和、最大値、最大公約数、バブルソート、シェルソート、フィボナッチ数列を与え、動作確認を行った。表 2 と表 3 に設計した 2 種類のプロセッサを、動作周波数や配置配線後の回路規模などにより性能比較を行った結果を示す。

表 2：マルチサイクルとパイプラインの動作周波数

	マルチサイクル	パイプライン
動作周波数	21.991	13.481

（単位：MHz）

表 3：マルチサイクルとパイプラインの回路規模

	マルチサイクル	パイプライン
記憶素子（FF）	8000（4%）	14000（7%）
ロジック（LUT）	42000（21%）	48000（24%）
回路規模	50000（25%）	62000（31%）

（単位：システムゲート）

動作周波数はマルチサイクルの方が速いという結果になった。これは、パイプラインプロセッサにはパイプラインレジスタに大きなレジスタを使用しており、その遅延が原因だと考えられる。回路規模はマルチサイクルを構成する記憶素子（フリップフロップ・FF）とロジック（ルックアップテーブル・LUT）が、FPGA のどれだけの割合を占めるかで比較した。マルチサイクルは 20 万システムゲート FPGA のうち FF と LUT 合わせて 25%を使用している。パイプラインは 31%を占めた。パイプラインはデータパスが複雑であり、大きなレジスタを使用しているため、マルチサイクルよりも回路規模が大きくなった。

今後の課題は、パイプラインの動作周波数を上げることと回路規模の縮小である。今回の設計ではパイプラインレジスタが動作周波数や回路規模のボトルネックとなっていることが分かったので、パイプラインのデータパスを再考し、使用するレジスタを小さくすることで、動作周波数の向上を図る。

4　アーキテクチャ可変なプロセッサシミュレータの設計

4.1 要求仕様

設計するシミュレータは 16 ビット MONI 命令セットを対象とした命令セットシミュレータである。単一サイクル、マルチサイクル、パイプライン、スーパースカラの 4 種類のアーキテクチャを任意に選択することができ、各アーキテクチャにおいて複数の実行モードを用意する。それぞれの実行モードにおいてメモリ、プログラムカウンタ、レジスタの変化の様子を逐一表示し､使用しているユニットが目で見て確認できるような GUI を持つ。また、プログラム実行後には、メモリアクセス数、分岐回数、ストール回数などを表示し、プログラムの評価に用いる。学習者はプロセッサアーキテクチャの理解や、アーキテクチャに最適なプログラミングの学習を行う。

4.2 シミュレータの構成画面と使用法

設計したシミュレータの構成画面を図 4 に示す。

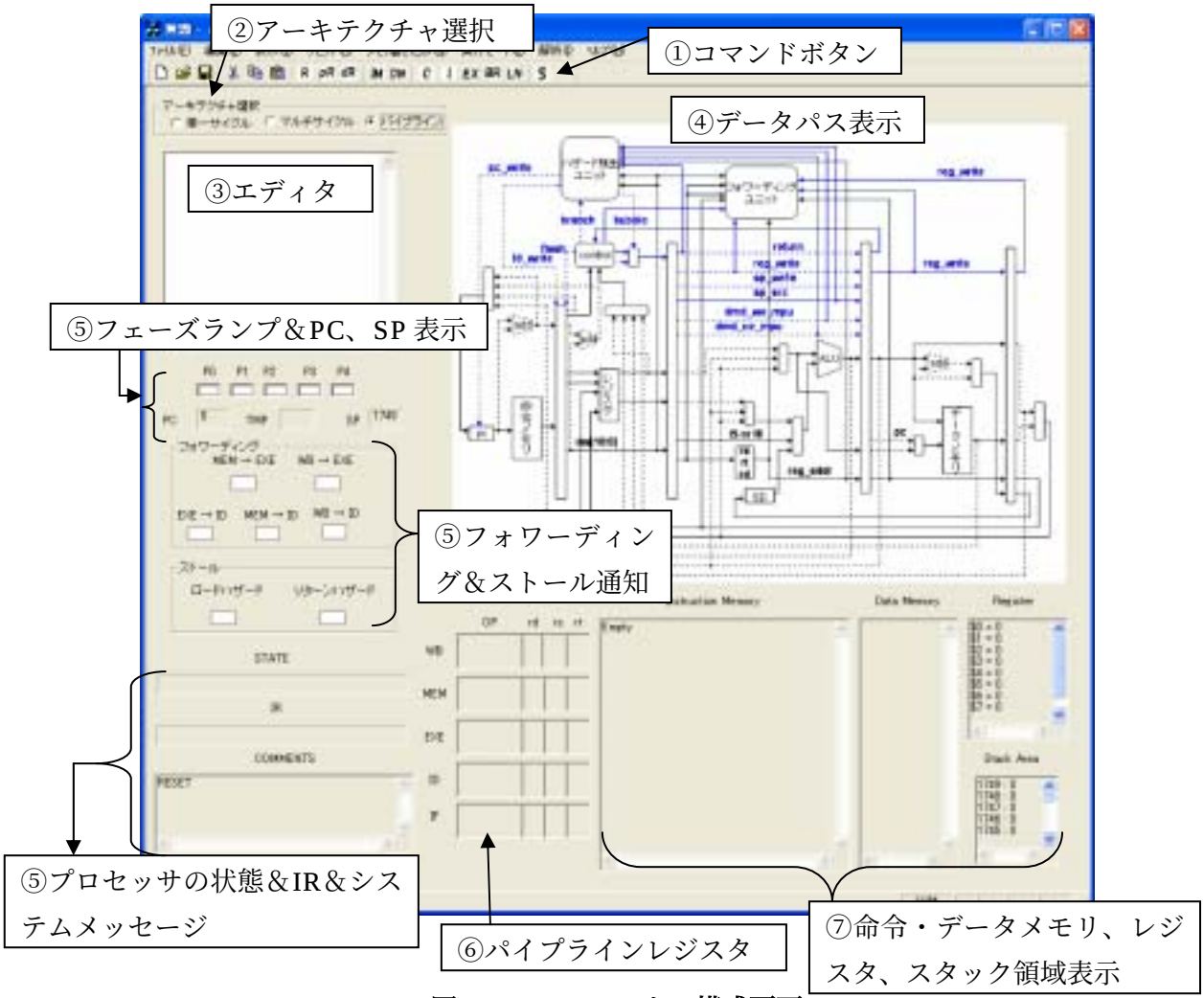


図 4：シミュレータの構成画面

シミュレータはコマンドボタン、アーキテクチャ選択、エディタ、データパス表示、プロセッサの状態表示、パイプラインレジスタ、メモリ・レジスタの表示の部分に分かれる。以下にそれぞれの詳細について述べる。

① コマンドボタン

シミュレータを動作させるためのコマンドボタン。リセット、メモリ書き込み、各種実行モード、プログラム解析などのコマンドがある。

② アーキテクチャ選択

現在、単一サイクル、マルチサイクル、パイプラインの 3 種類のアーキテクチャが選択可能である。

③ エディタ

プログラムやデータの入力、またデバッグ時のブレークポイント設定に用いる。

④ データパス表示

選択したプロセッサのデータパスが表示される。また、クロック毎、命令毎に使用されているユニットやアクティブになっている制御線が強調表示される。

⑤ 内部状態表示

フェーズランプはマルチサイクルを選択し、クロック実行を行った際に、現在命令のどのフェーズを実行しているのかを示すものである。また、PC（プログラムカウンタ）や SP（スタックポインタ）、IR（命令レジスタ）の内容を表示する。パイプラインを選択している場合には、ストールやフォワーディングが行われていることを通知する機能がある。さらに、全てのアーキテクチャ、実行モードにおいて、現在プロセッサがどのような状態にあるかを示すステート表示や、エラーなどを表示するシステムメッセージ表示部がある。

⑥ パイプラインレジスタ

パイプラインを選択した際に、各パイプラインレジスタに入っている命令やデータの依存関係などを表示する。

⑦ メモリ・レジスタの表示

命令メモリ、データメモリ、レジスタファイル、スタック領域の内容を表示する。

プロセッサを使用する場合は、先ずエディタ上でセンブリプログラムを作成し、メモリに書き込む。アセンブルエラーがなければアーキテクチャ選択後、実行を開始する。実行終了後もデータやアーキテクチャを変更しながらデバッグを繰り返し行える。そして、最後にシミュレーションデータを表示し、プログラムやアーキテクチャの評価を行う。図 5 にシミュレータの使用法の一例を示す。

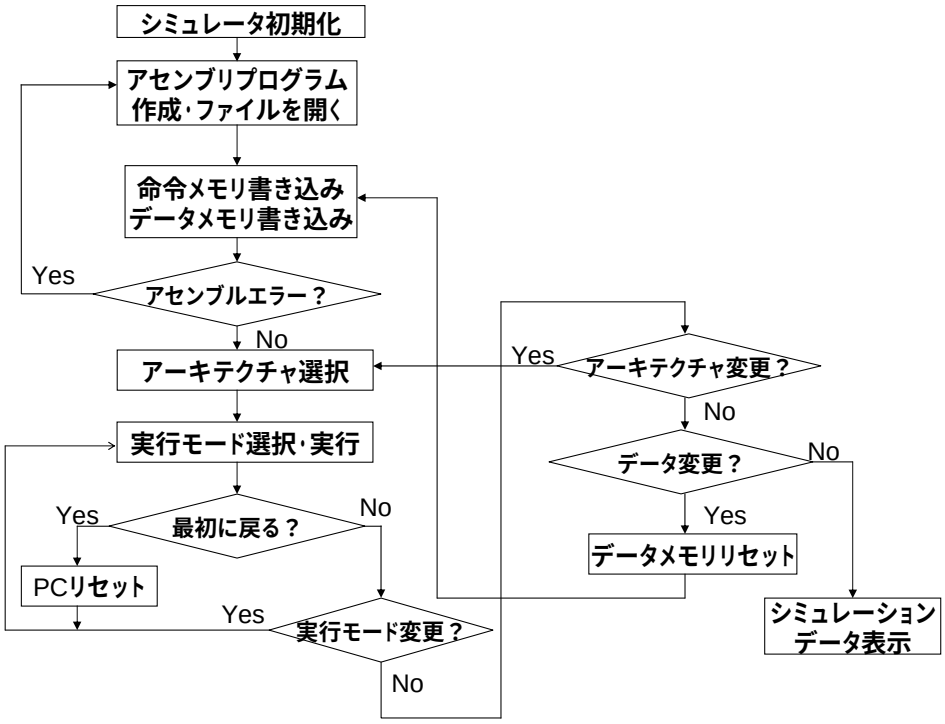


図 5：シミュレータの使用例

4.3 シミュレータの機能

シミュレータには 1 命令実行、1 クロック実行、プログラム実行、ブレーク実行、設定行数実行、及びストール発生まで実行の 6 種類の実行モードがある。以下では、選択したアーキテクチャにおいて、それぞれの実行モードがどのような動作をするのかを述べる。また、プログラムを評価するための情報を表示するプログラム解析について述べる。

4.3.1 単一サイクル実行

単一サイクルでは 1 クロック実行と 1 命令実行は同じ動作をする。1 クロック実行または 1 命令実行コマンドを実行すると 1 命令ずつ実行される。単一サイクル実行の際に注目すべき場所は以下の通りである。

- データパス表示

1 命令ごとにプロセッサの中で使用されているユニットが強調して表示される。また、そのときのアクティブになっている制御線も強調表示される。

- プロセッサの状態

各命令において、現在プロセッサが行っている命令と、その命令ではどのような動作をするのかが表示される。表 4 にいくつかの例を示す。

表 4：単一サイクル選択時のプロセッサの状態例

アドレス	命令メモリの内容				STATE 表示
13	SLT	$5	$1	$4	13：SLT if（$1< $4）then $5 ← 1
14	BEQZ	$5	WORK2		14：if($5 = 0) then pc ← 8
17	LDLI	$3	#1		17：$3 ← 1

- PC：現在のプログラムカウンタが表示される
- SP：現在のスタックポインタが表示される
- COMMENTS：現在までに実行した命令の数が表示される

図 6 に単一サイクルを選択した際のシミュレータの実行画面を示す。データパス表示部では、使用されているユニットとアクティブになっている制御線が強調表示されている様子や、レジスタに書き込まれた様子がわかる。

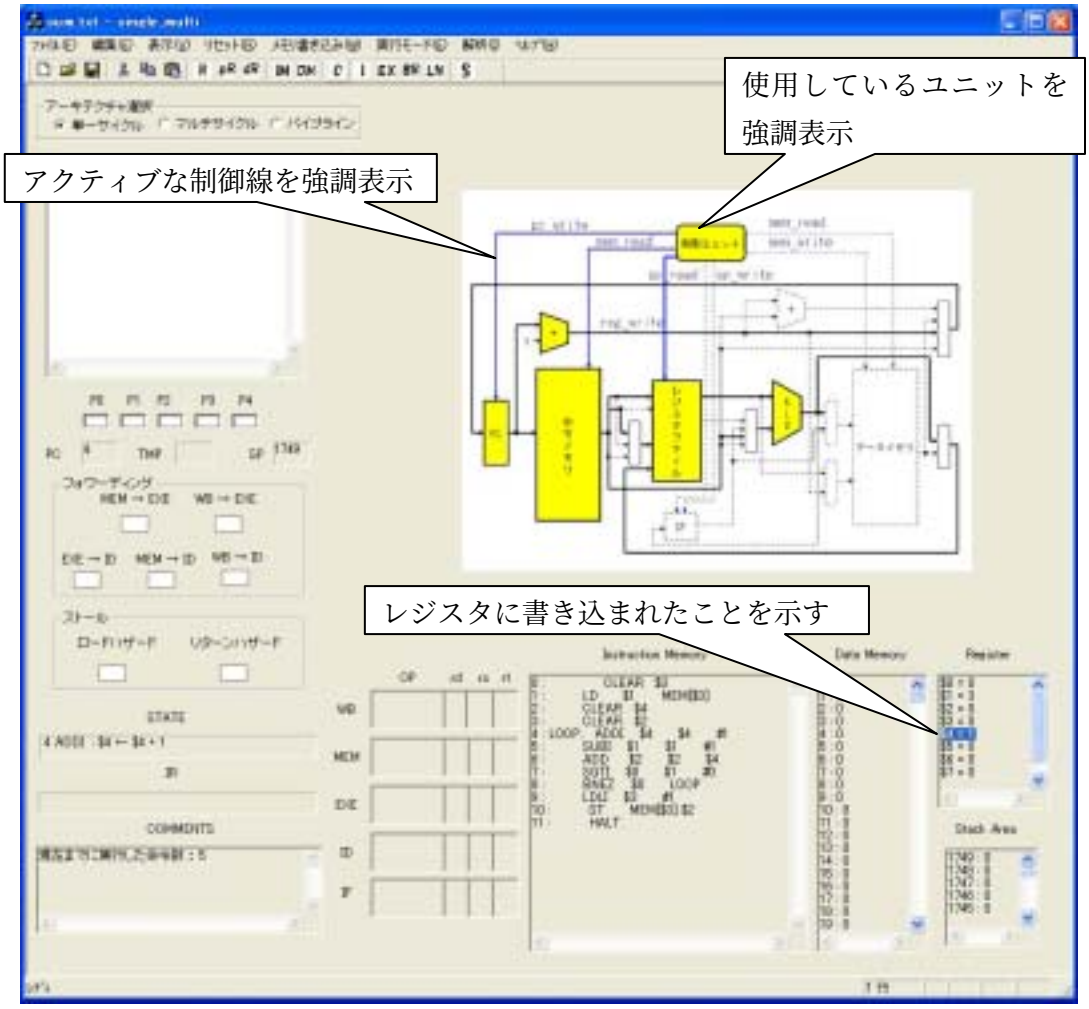


図 6：単一サイクルの実行画面

4.3.2 マルチサイクル実行

マルチサイクルでは 1 クロック実行と 1 命令実行では動作が異なる。マルチサイクルでの 1 命令実行の振る舞いは単一サイクルと同様であるので、ここではマルチサイクルの 1 クロック実行について述べる。マルチサイクルの 1 クロック実行の際に注目すべきものは以下の通りである。

- データパス表示

  各クロックで使用されているプロセッサのユニットとアクティブになっている制御線が強調表示される。
- フェーズランプ：現在、実行中のフェーズを示す。
- プロセッサの状態

  命令の各クロックでプロセッサがどのような動作をしているのかが表示される。表 5 に LD 命令の例を示す。

表 5：マルチサイクルで LD 命令実行時のプロセッサの状態例

アドレス	命令メモリの内容	Step	STATE 表示
10	LD $5 MEM[$4]（$4=2、DM[2]=4）	0	IF：IR ←INSTRUCTION_MEM[10]、PC++
		1	ID：TMP ← PC+IMMEDIATE
		2	LD-3：DATA_MEM[2] =4
		3	LD-4：REG[5] ← 4

- PC：現在のプログラムカウンタを示す。命令フェッチステップでインクリメントされている。
- IR：命令レジスタの内容が表示される。
- TMP：TMP レジスタの内容が表示される
- SP：スタックポインタの内容が表示される
- COMMENTS：現在までに実行した命令の数が表示される

マルチサイクルを選択した場合の実行画面を図 7 に示す。フェーズランプにより、現在命令のどのフェーズを実行しているかがわかる。また、使用しているユニットやレジスタに書き込まれたことが強調して表示されるのは単一サイクルと同様である。

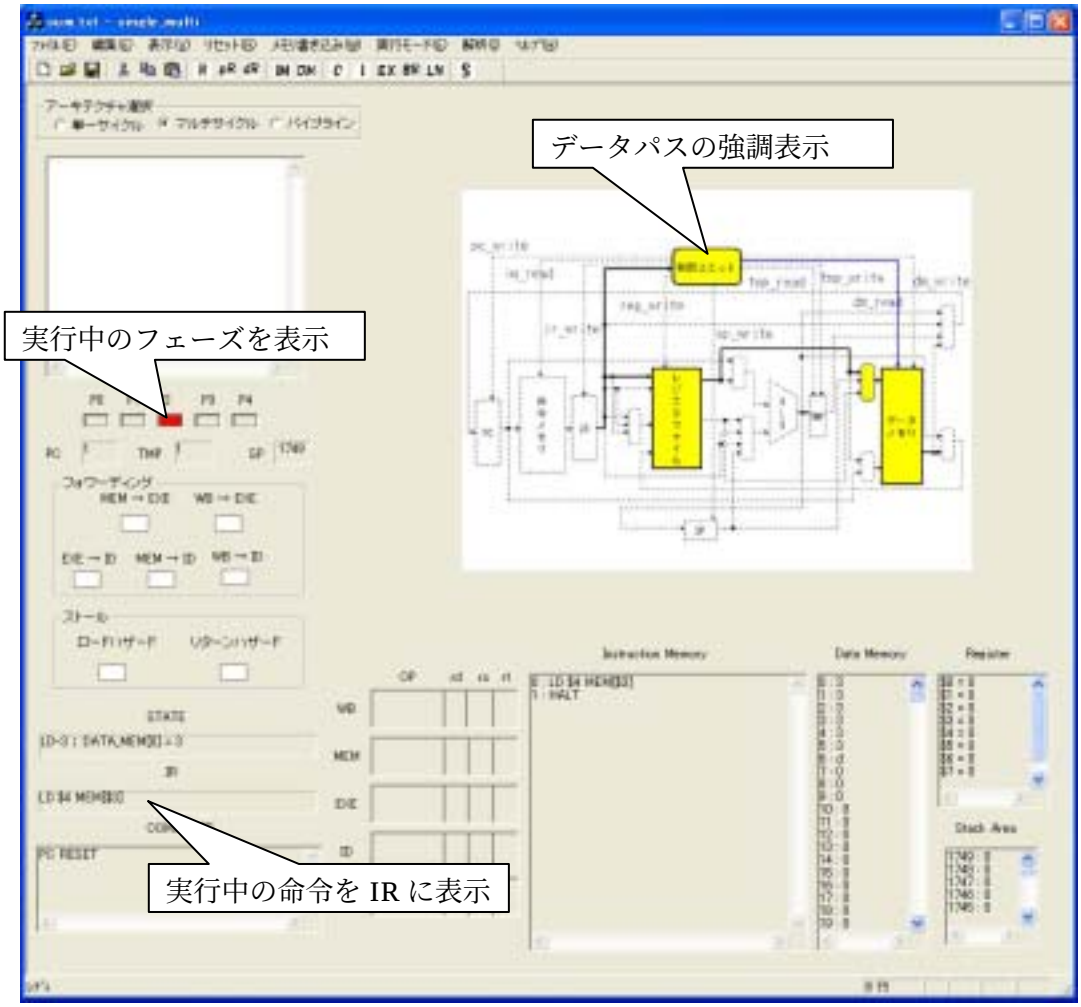


図 7：マルチサイクルの実行画面

4.3.3 パイプライン実行

パイプラインではストール発生まで実行という実行モードが選択可能になる。これはストールが発生し、パイプラインにバブルが挿入されるまでプログラムを実行するというモードである。パイプライン実行の際に注目すべき場所は以下の通りである。

- データパス表示

パイプラインに命令が投入される様子を、各パイプラインステージで使用されているユニットを強調表示することにより示す。フォワーディングが行われた際は、フォワーディング用のデータパスを赤く表示し、ストールが発生した場合は各パイプラインステージにバブルを表示する。

- フォワーディングランプ

パイプラインでフォワーディングが起こったことを通知する。MEM→EXE、WB→EXE、EXE→ID、MEM→ID、WB→ID の 5 種類のフォワーディング別にランプが点

灯する。

- ストールランプ

ロードハザードによるストールと、リターン命令によるハザードの 2 種類を別々に通知する。

- パイプラインレジスタ

データパスと同期して命令が投入される様子を示す。パイプラインに投入された命令のうち、レジスタに依存関係がある場合は、そのレジスタの色を変えて通知する。フォワーディングで対処できる依存関係は赤、ロード命令によるストールが発生する場合は青で示す。また、バブルが挿入された場合は★マークでバブルを表す。

- PC：パイプラインに投入された命令のプログラムカウンタを表示
- SP：スタックポインタの内容が表示される

パイプラインを選択した場合に、フォワーディングが行われている際の実行画面を図 8 に示す。フォワーディングランプやデータパスでフォワーディングの様子を示している。

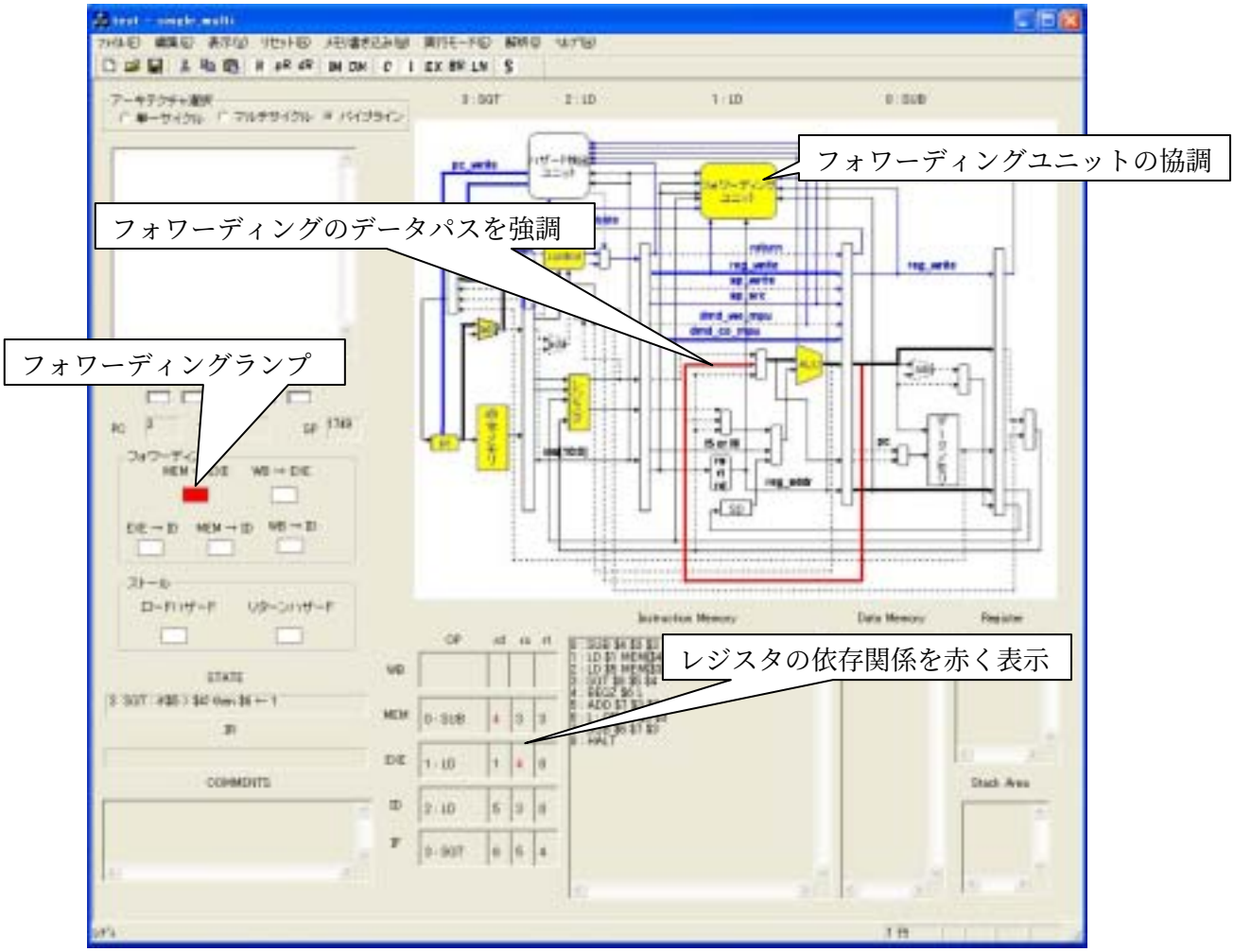


図 8：パイプラインの実行画面（フォワーディング）

次に、パイプラインにバブルが挿入されている実行画面を図 9 に示す。パイプラインレジスタやデータパスにバブルを挿入し、ストールランプを点灯させ、ストールの発生を示す。

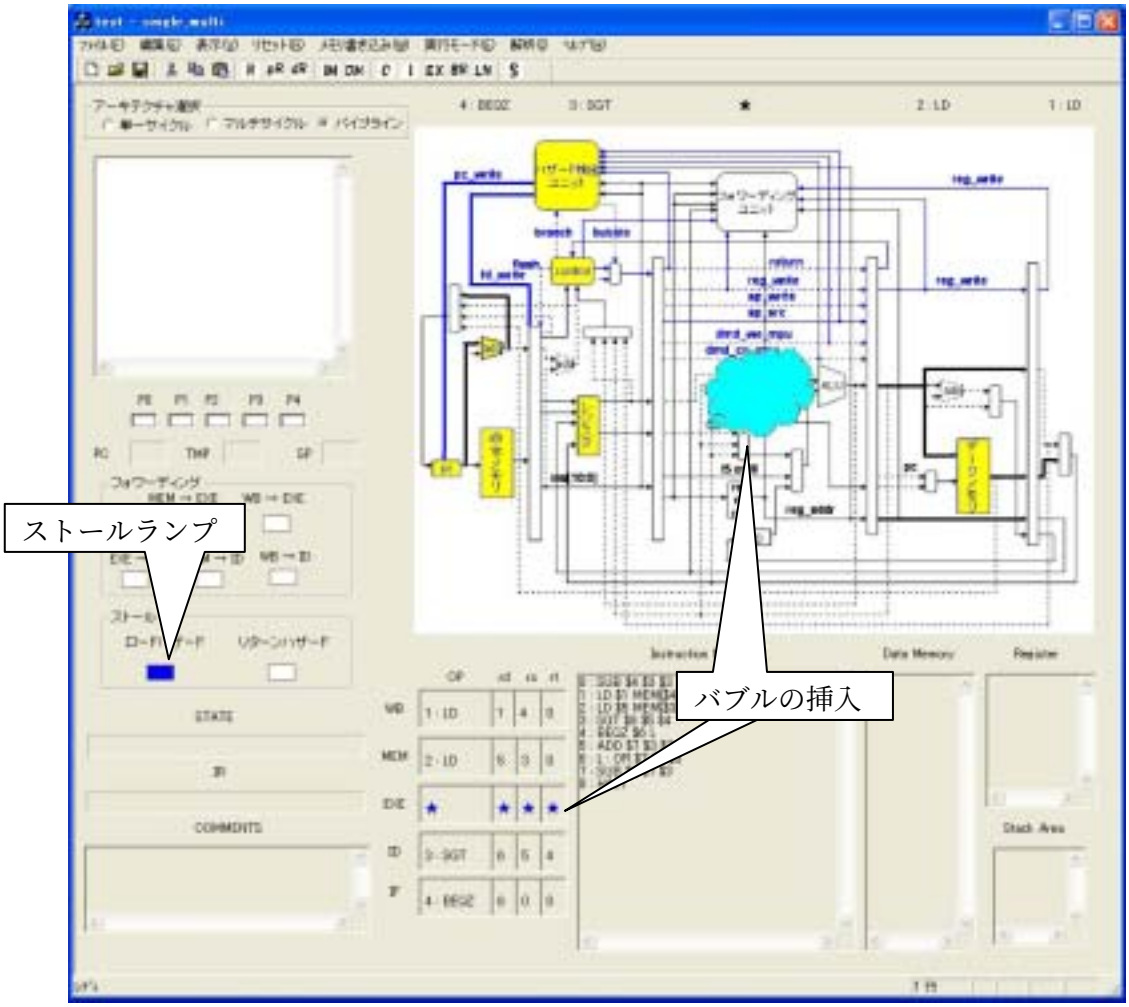


図 9：パイプラインの実行画面（ストール）

4.3.4 プログラム実行

プログラム実行は全てのアーキテクチャに共通であり、プログラムの最後まで実行するモードである。プログラムカウンタ、スタックポインタ、命令レジスタ、レジスタファイル、及びデータメモリの内容はプログラムが終了した時点で状態が表示される。

4.3.5 デバッグ

アセンブリプログラムのデバッグを行うための機能として以下の実行モードを用意した。

- ブレーク実行

  エディタで指定したブレークポイントまでプログラムを実行するモードである。
- 設定行数実行

エディタで指定した行数分だけアセンブリプログラムを実行するモードである。

- ストール発生まで実行

パイプラインを選択しているときのみ有効な実行モードで、パイプラインにストールが発生するまでプログラムを実行する。

4.3.6 プログラム解析

プログラムの実行が完了すると図 10 のようにシミュレーションデータのダイアログを表示させ、プログラムの解析を行うことができる。ダイアログに表示された情報はファイル出力が可能である。

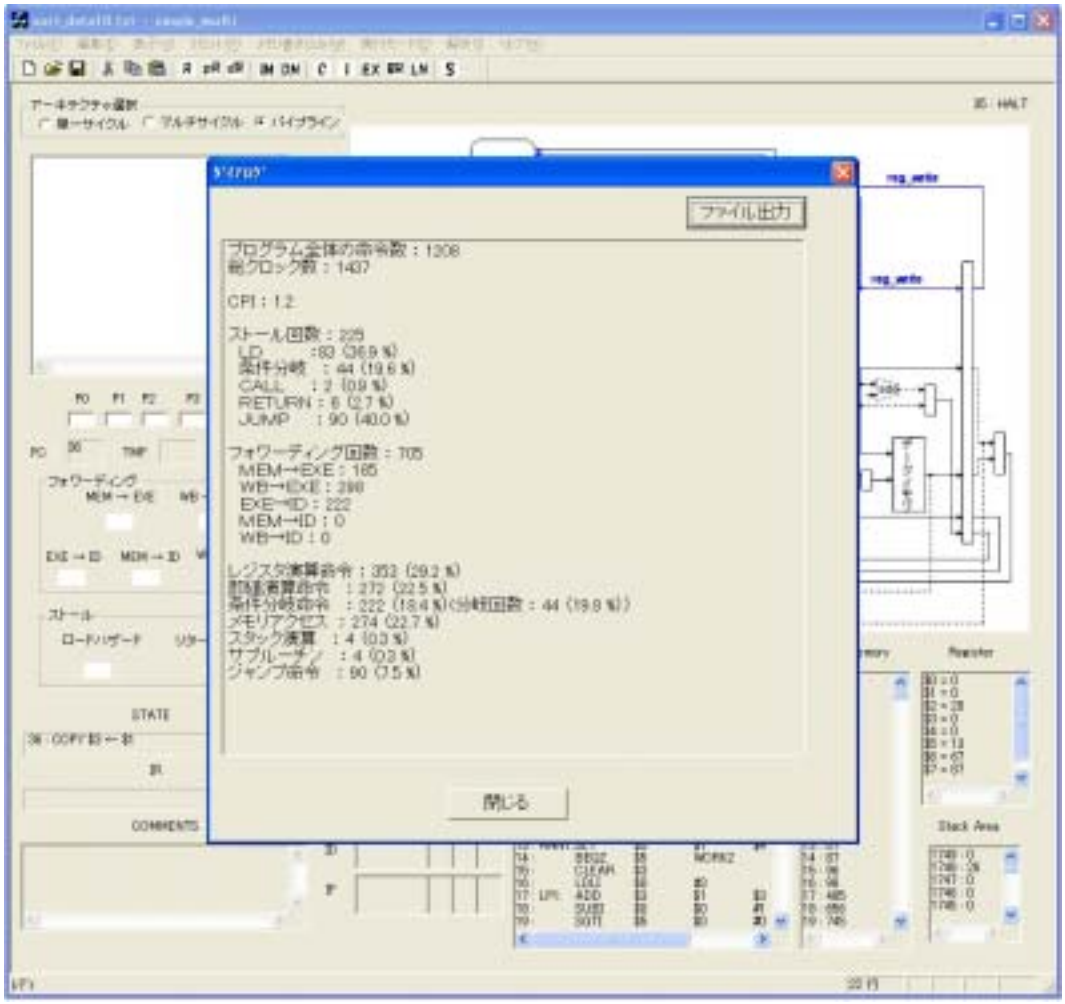


図 10：シミュレーションデータダイアログ

ダイアログに表示される内容は以下のとおりであり、単一サイクル、マルチサイクル、パイプラインで共通である。

- プログラム全体の命令数
- 総クロック数

- CPI
- ストール回数（ストールの発生した命令とその割合）
- フォワーディング回数（フォワーディングの種類とその割合）
- 命令の種類ごとの実行回数とその割合

シェルソートを要素数 20、パイプラインで行った場合の例を以下に示す。

プログラム全体の命令数 ： 1208
総クロック数 ： 1437

CPI ： 1.2

ストール回数 ： 225
ＬＤ　　：83 （36.9 %）
条件分岐　： 44 （19.6 %）
ＣＡＬＬ　： 2 （0.9 %）
ＲＥＴＵＲＮ：6 （2.7 %）
ＪＵＭＰ　： 90 （40.0 %）

フォワーディング回数 ： 705
ＭＥＭ→ＥＸＥ ： 185
ＷＢ→ＥＸＥ ： 298
ＥＸＥ→ＩＤ ： 222
ＭＥＭ→ＩＤ ： 0
ＷＢ→ＩＤ ： 0

レジスタ演算命令 ： 353 （29.2 %）
即値演算命令　： 272 （22.5 %）
条件分岐命令　： 222 （18.4 %）
(分岐回数 ： 44 （19.8 %））
メモリアクセス　： 274 （22.7 %）
スタック演算　： 4 （0.3 %）
サブルーチン　： 4 （0.3 %）
ジャンプ命令　： 90 （7.5 %）

命令の種類ごとに分類しそれら命令数と全命令数に対して占める割合を示した。命令の種別は以下のとおりである。

レジスタ演算：MONI 命令セット R フォーマット
即値演算：MONI 命令セット I5 フォーマット及び LDHI、LDLI
条件分岐命令：BEQZ、BNEZ
メモリアクセス：LD、ST、POP、PUSH、RETURN、CALL
スタック演算：POP、PUSH、RETURN、CALL
サブルーチン；RETURN、CALL
ジャンプ命令：JUMP

条件分岐命令は、条件分岐命令がプログラム全体で占める割合と、さらに条件分岐命令の中で分岐した回数とその割合を示した。つまり、上記の例では全 1437 命令中条件分岐命令が 222（18.4%）あって、222 の条件分岐命令中 44 回（19.8%）分岐したということである。

5　C++によるプロセッサシミュレータの試作

5.1 開発環境

開発ツールとして VisualC++6.0 を用いた。SDI（Single Document Interface）形式を採用した単一の窓構成となっている。アプリケーションの見せ方を決める View クラスを中心にプログラミングを行い、ファイルを保存する Document クラス、独自のダイアログを定義する Dialog クラス、SDI のフレームを構成する MainFrame クラスなどとデータをやり取りしながら設計した。以下に、各クラスで使用するファイル名とその機能を示す

- MainFrm.cpp：SDI 形式アプリケーションのフレームを制御する
- single_multiDoc.cpp：ユーザデータを保持する。ファイル編集を含む
- single_multiView.cpp：ウィンドウの表示方法や見せ方を処理する
- DataDialog.cpp：独自に定義したダイアログを制御する
- InfoDialog.cpp：独自に定義したダイアログを制御する
- single_multi.cpp：アプリケーションの中心となるプログラム
- single_multi.rc：ユーザが使用するリソースを列挙
- stdAfx.cpp：プリコンパイル済ヘッダーファイルを構築するためのプログラム

5.2 モジュール構成

モジュールはリセット、メモリ書き込み、実行、状態表示の 4 つに大別できる。

（1） リセット

各種初期化を行う関数群。実行関数や状態表示関数にフラグを渡し、初期化する。構成する関数を以下に示す。

- Reset()：シミュレータの初期化
- Pcreset()：プログラムカウンタの初期化
- Dmreset()：データメモリの初期化

（2） メモリ書き込み

命令メモリやデータメモリに値を書き込む関数群。命令とデータの配列にそれぞれ値を格納した後、実行関数に渡す役割をする。アセンブルエラーの検出なども行う。構成する関数を以下に示す。

- Imwrite()：命令メモリ書き込み
- Dmwrite()：データメモリ書き込み

（3） 実行

実際にプログラムを実行するための関数群である。実行モードにあわせて関数を作成した。それぞれの実行関数は、Imwrite()関数や Dmwrite()関数から配列を受け取り、その値を使

用して命令を実行する。構成する関数を以下に示す。

- Clk()：1 クロック実行
- Inst()：1 命令実行
- Exe()：プログラム実行
- Break()：ブレーク実行
- Line()：設定行数実行
- Pipe()：パイプライン実行
- Stole()：ストール発生まで実行

（4） 状態表示

それぞれの状態にあった情報を表示する関数群である。実行関数から状態を示す値を受け取り、それを元に画像や文字で情報を表示する。構成する関数を以下に示す。

- OnDraw()：シミュレータの状態にあった画像を表示する
- OnMenuDlg()：プログラム終了後のシミュレーションデータを別窓で表示する

以上で述べたモジュール構成を図 11 に示す。

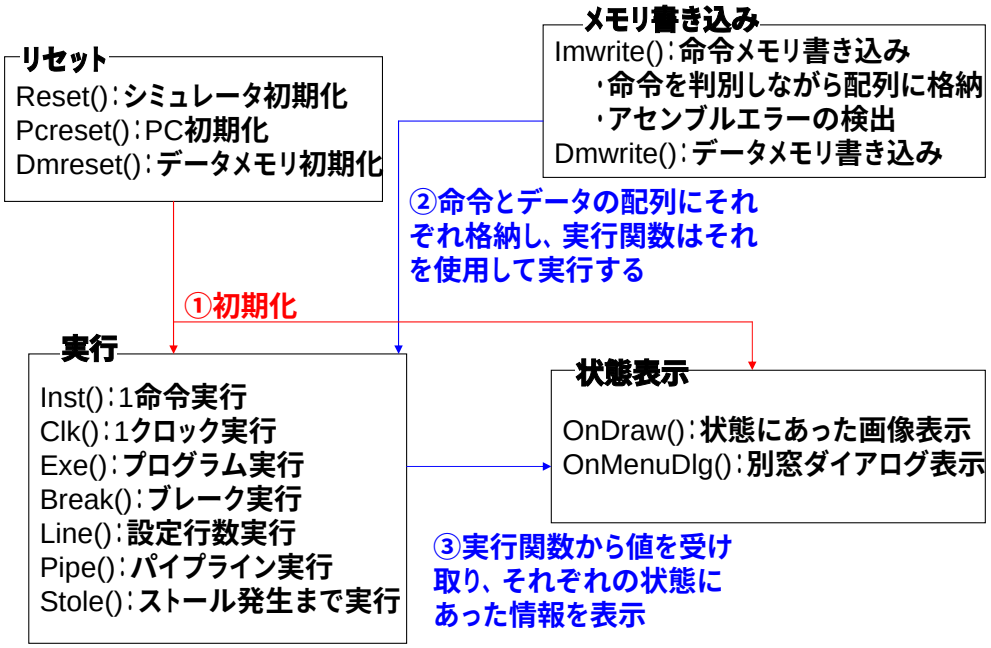


図 11：モジュール構成

先ず、リセットモジュールは実行モジュールと状態表示モジュールの初期化を行う（図 11①）。次に、命令メモリとデータメモリへ値を書き込む Imwrite()や Dmwrite()が実行されると、命令とデータの値がそれぞれ配列に格納され、実行関数はそれらの配列を使用して、実行を行う（図 11②）。さらに、状態表示モジュールは実行関数から現在のシミュレータの

状態を値として受け取り、それに基づいて画像などの表示を行う（図 11③）。

シミュレータを設計する上で重要な役割を果たすのは、7 種類ある実行関数である。それらの関数は 3 種類のアーキテクチャごとに異なる動作をする。実行関数のアーキテクチャごとの動作を表 6 に示す。

表 6：実行関数のアーキテクチャごとの動作

	単一サイクル	マルチサイクル	パイプライン
Inst()	命令実行、PC、SP、RF、DM 表示、命令数のカウント	命令の途中のクロックならばその命令を最後まで実行し、その後 1 命令ずつ実行	無効
Clk()	Inst()を呼び出す	ステートに従い実行	Pipe()を呼び出す
Pipe()	無効		命令を実行しながらパイプラインステージに見立てた配列に格納。配列の依存関係を調べ、フォワーディングやハザードを検出
Stole()	無効		Pipe()をストールが発生するまで実行
Exe()	Inst()を命令の最後まで呼び出す		Pipe()を命令の最後まで呼び出す
Break()	エディタで指定した pc まで Inst()を呼び出す		エディタで指定した pc まで Pipe()を呼び出す
Line()	エディタで指定した行数分、Inst()を呼び出す		エディタで指定した行数分、Pipe()を呼び出す

5.3 各モジュールの作成方法

シミュレータを構成する主な関数の作成方法を述べる。

（1） Imwrite()

命令メモリに書き込む関数である。エディタに表示されたアセンブリプログラムを 1 行ずつ、1 次元配列に格納する。それを 2 次元配列にコピーした後、コメントを除去し、分岐命令などのラベルがあれば、それらをラベル専用の配列に格納する。そして、命令をオペコード、オペランド、即値の配列に格納しながら、エラーチェックを行う。エラーがなければ、シミュレータのメモリ表示領域に命令を表示する。図 12 に Imwrite()関数のフローチャートを示す。

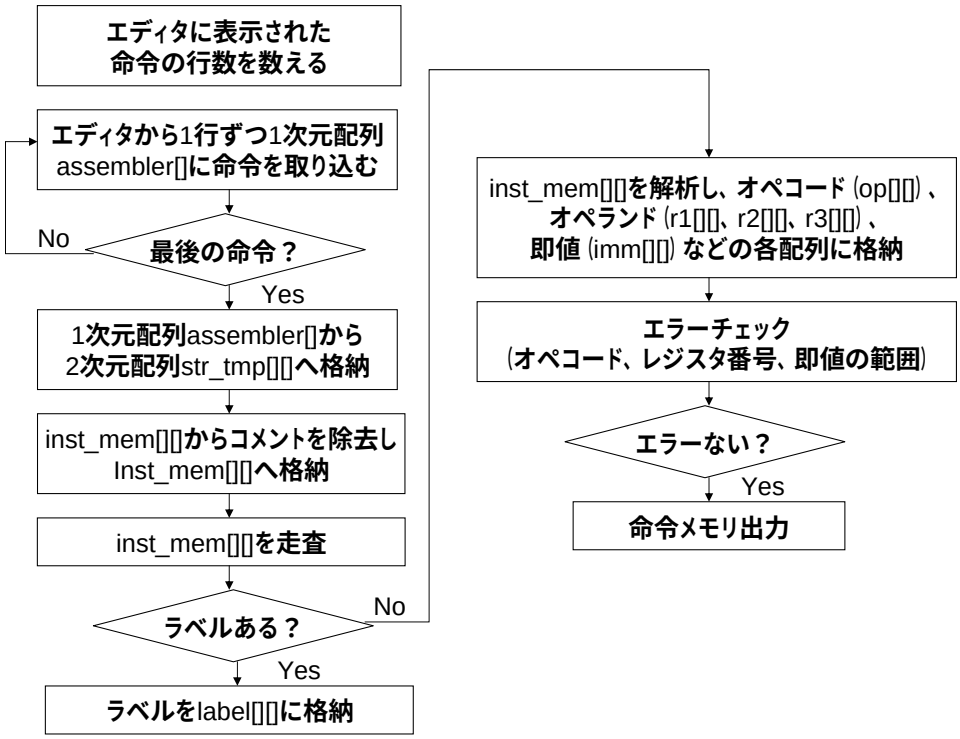


図 12：Imwrite()関数のフローチャート

（2） Inst()

1 命令実行を実現する関数である。先ず、現在選択されているアーキテクチャを取得する。マルチサイクルが選択されている場合、実行中の命令がフェーズの途中であれば、1 クロック実行関数 Clk()を呼び出し、最後のフェーズまで実行する。その後、すでに Imwrite()（命令メモリ書き込み）により値が格納されているオペコード、オペランド、即値の配列を読み出し、命令を実行する。命令を実行しながら、実行命令数や種類別の命令数のカウントやレジスタ、画像などの表示を行う。パイプラインが選択されている場合は Inst()関数は使用されない。図 13 に Inst()関数のフローチャートを示す。

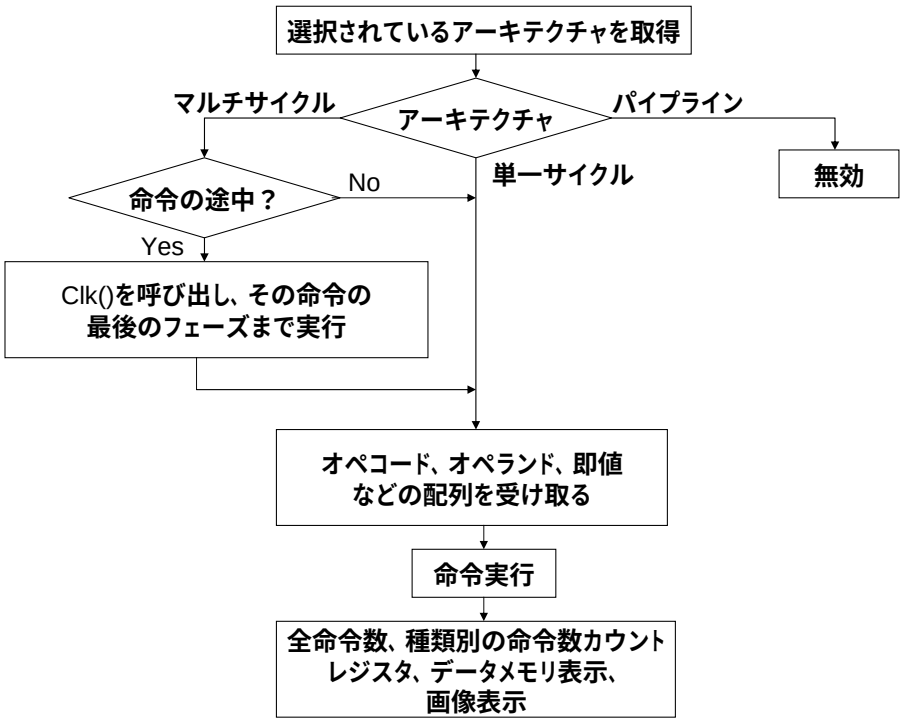


図 13：Inst()関数のフローチャート

（3） Clk()

1 クロック実行を実現する関数である。先ず、現在選択されているアーキテクチャを取得する。単一サイクルが選択されている場合は前述の Inst()関数が呼び出される。また、パイプラインの場合は後述する Pipe()関数が呼び出される。マルチサイクルではオペコード、オペランド、即値の配列を読み出し、「命令フェッチ→命令デコードとレジスタのフェッチ、分岐アドレスの計算→演算の実行、メモリ読み出し、pc、sp 操作→レジスタ書き込み、メモリ書き込み→CALL 命令における pc の書き換え」のステートに従い、命令を実行する。その後、実行命令数や種類別の命令数のカウントやフェーズランプの点灯、及びレジスタ、画像などの表示を行う。

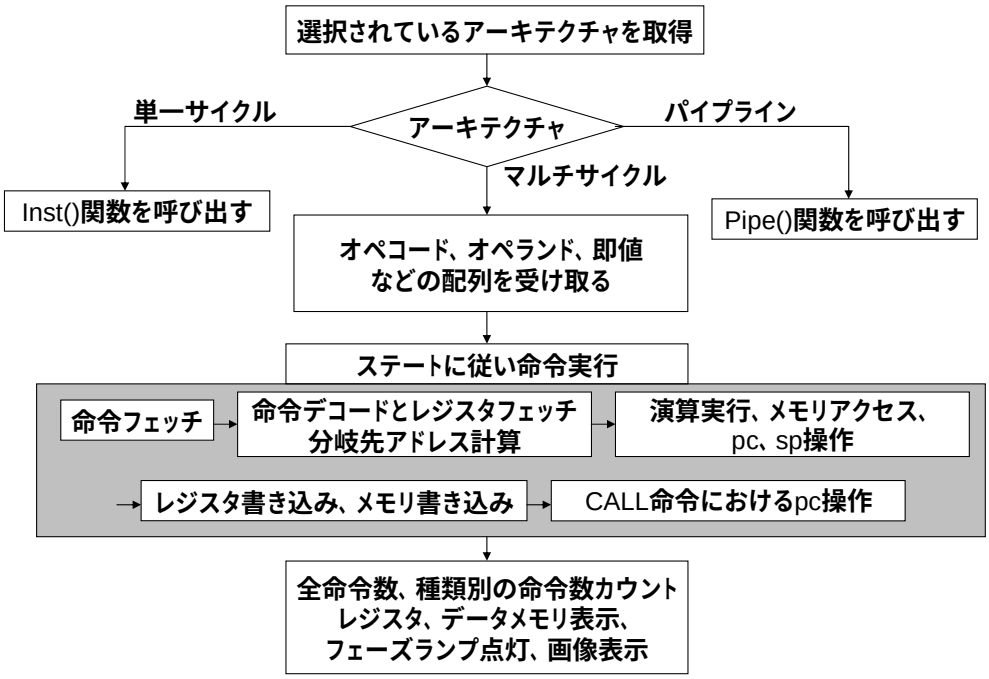


図 14：Clk()関数のフローチャート

（4） Pipe()

パイプライン実行を実現する関数である。Inst()関数を呼び出し 1 命令実行した後、その命令を配列 pipe[][0]に格納する。pipe[][]はパイプラインステージに見立てた 2 次元配列で、pipe[][0]は IF ステージ、pipe[][1]は ID ステージ、pipe[][2]は EXE ステージ、pipe[][3]は MEM ステージ、pipe[][4]は WB ステージに対応している。pipe[][]に格納された命令の依存関係を調べ、ストールが発生した場合は pipe[][]に“★”を格納し、バブルが挿入されたことを示す。また、ストール発生フラグを立て、画像表示やストールランプ点灯の際に使用する。同様に、フォワーディングが発生した場合はフォワーディングフラグを立てる。その後、ストールフラグやフォワーディングフラグの状態に基づき、画像表示などを行う。そして、次の命令が pipe[][0]に格納できるように、pipe[][1]から pipe[][4]の値を 1 つずつずらす。それを HALT 命令まで繰り返す。pipe()関数のフローチャートを図 15 に示す。

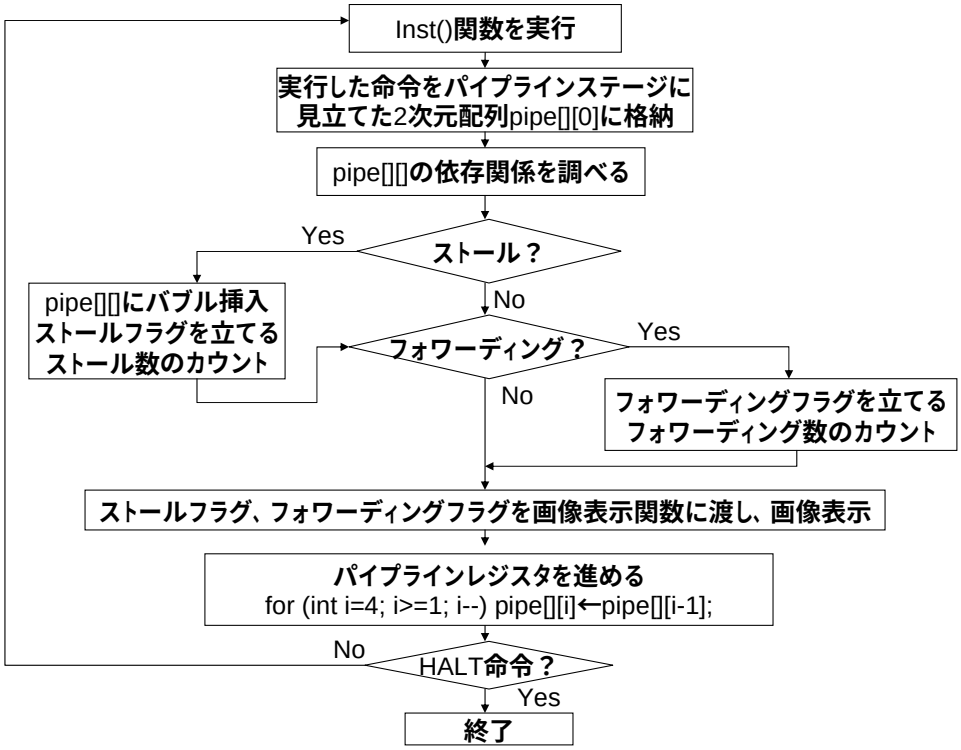


図 15：pipe()のフローチャート

6　シミュレータのテストと評価

6.1 テストと考察

シミュレータのテストプログラムとして「N までの和(N=100)」、「最大値（要素数：50）」、「最大公約数（124、36）」、「バブルソート（要素数：50）」、「シェルソート（要素数：50）」、「フィボナッチ数列（8 番目）」を実行した。４章で述べたプログラム解析から得られた結果を表 7 に示す。

表 7：テストプログラムの実行結果

		単一サイクル	マルチサイクル	パイプライン
N までの和（N=100）	総実行命令数	507	507	507
	総クロック数	507	1925	610
	CPI	1.0	3.8	1.2
	ストール回数	-	-	99（16.2%）
	フォワーディング回数	-	-	302
最大値（要素数：50）	総実行命令数	369	369	369
	総クロック数	369	1318	573
	CPI	1.0	3.6	1.6
	ストール回数	-	-	200（34.9%）
	フォワーディング回数	-	-	206
最大公約数（124,36）	総実行命令数	58	58	58
	総クロック数	58	213	74
	CPI	1.0	3.7	1.3
	ストール回数	-	-	12（16.2%）
	フォワーディング回数	-	-	36
バブルソート（要素数：50）	総実行命令数	17038	17038	17038
	総クロック数	17038	63081	25289
	CPI	1.0	3.7	1.5
	ストール回数	-	-	8247（32.6%）
	フォワーディング回数	-	-	6919
シェルソート（要素数：50）	総実行命令数	4216	4216	4216
	総クロック数	4216	15390	5002
	CPI	1.0	3.7	1.2
	ストール回数	-	-	782（15.6%）
	フォワーディング回数	-	-	2467

フィボナッチ（8）	総実行命令数	553	553	553
	総クロック数	553	2031	692
	CPI	1.0	3.7	1.3
	ストール回数	-	-	135（19.5%）
	フォワーディング回数	-	-	174

マルチサイクルでは 1 命令を 3～5 クロックで実行するので、CPI は 3.6～3.7 という値になった。一方、パイプラインではストール回数により CPI が左右される。最大値を求めるプログラムではストールが全クロックの 35%を占め、CPI も 1.6 と他のプログラムに比べて大きくなった。ストールを減らすためにはその原因を知らなければならない。4 章で述べたプログラム解析ではストールを発生させている命令が表示されるのでそれを元に、プログラムの変更を行うことが可能である。表 8 に最大値のプログラムを実行した際に得られる情報の詳細を示す。

表 8：最大値のストール発生回数

	合計	LD	Branch	CALL	RETURN	JUMP
ストール回数	200	51	98	0	0	51
(%)		(25.5)	(49.0)	(0.0)	(0.0)	(25.5)

これらの情報から最大値のプログラムは、分岐命令がストールの約半数を占めていることがわかる。よって、プログラム中の分岐命令を減らすことでストールの回数を抑えることが出来る。また、LD 命令や JUMP 命令によるストールも減らせるのかどうかを検討する余地がある。

また、表 9 にバブルソートとシェルソートのストール発生回数の比較を示す。

表 9：バブルソートとシェルソートのストール発生回数の比較

	合計	LD	Branch	CALL	RETURN	JUMP
バブルソート	8247	1225	652	1274	3822	1274
(%)		(14.9)	(7.9)	(15.4)	(46.3)	(15.4)
シェルソート	782	297	151	.3	9	322
(%)		(38.0)	(19.3)	(0.4)	(1.2)	(41.2)

バブルソートでは RETURN 命令によるストールが約半分を占めている。これは、プログラムをモジュールごとに分けすぎていることが原因である。そのため、サブルーチンの数を減らし、1 つのモジュールを大きくするといったプログラムの変更が望ましい。また、シェルソートでは JUMP 命令によるストールの割合が多いため、JUMP 命令を減らす方向でプログラムの改善を行うと良いと思われる。

このように、設計したシミュレータはパフォーマンスデバッガとしても利用できる。学習者はアーキテクチャ学習や、アセンブリプログラム設計・デバッグを行った後、開発し

たプログラムをアーキテクチャごとに最適化することが可能である。単一サイクルやマルチサイクルではプログラムの静的な行数を減らすことが目的となるし、パイプラインでは上述のようにストールを減らすようなプログラミングが必要である。

6.2 シミュレータの評価

試作したシミュレータを同じ研究室の方に使って頂き、感想を伺った。それらを以下にまとめる。

良かった点

- アセンブリプログラミングにおいて、プログラム解析機能が大変役に立った
- パイプラインにおけるハザードの仕組みがよく分かった
- プロセッサ内での現在の状態が見られるので、どのようにプロセッサが動作しているのかが理解しやすい
- アセンブリプログラムで簡単に命令が書け、自分の書いたプログラムがどのように動作しているのかがわかる

改善すべき点

- エディタが小さい
- ボタン（ツールバーの所）の使い方がわかりにくい
- 単一のウィンドウなので図を大きくしたくても出来ない。
- レジスタやデータメモリのウィンドウ上で、マウスを右クリックするなどして、初期化できれば良い
- １クロック進むはできるが、１クロック戻るができない。

良かった点を見ると、プログラム解析やデータパスやレジスタの表示が役立ったということで、設計した際の意図が十分に伝わっている印象を受ける。改善すべき点ではこれからのシミュレータ設計にとって大変参考になった。先ず、画面配置である。エディタやツールバーを工夫することで、使いやすいシミュレータになる。また、複数のウィンドウ（MDI：Multi Document Interface）形式を採ることで、データパス表示を単独のウィンドウで表示し、図の拡大や縮小が出来るようにすればよりわかりやすくなる。さらに、シミュレータの操作性を上げるために、レジスタやデータメモリの表示部分を右クリックすることで初期化できるようにすればいいのではという意見はもっともであり、直ぐにでも実現したい機能である。また、1 クロック戻るというモードは、プログラムのデバッグには確かに必要な機能であり、今後の課題となる。

7 おわりに

本論文では、ハード/ソフト・コラーニングシステムの構成要素であるマルチサイクル・パイプラインプロセッサの設計と、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を行った。プロセッサは VerilogHDL を用いて設計を行った。FPGA を対象としたゲートレベルシミュレーションを完了し、複数のテストパターンで動作確認を行った。また、論理合成後の動作周波数や、配置配線後の回路規模などを比較した。その結果、動作周波数はマルチサイクルの方が速いという結果となった。これは、パイプラインプロセッサではパイプラインレジスタに大きなレジスタを使用しているため、その遅延が原因であることが分かった。回路規模は、データパスが複雑なパイプラインの方が大きくなった。今後の課題は、パイプラインプロセッサの動作周波数の向上と回路規模の縮小である。

また、アーキテクチャ可変なプロセッサシミュレータの設計ではシミュレータの要求仕様とシミュレータの機能を述べた。単一サイクル・マルチサイクル・パイプラインの 3 種類のアーキテクチャが選択可能であり、データパス・レジスタ・メモリなどを可視化し、プログラム実行の様子が目で見て分かるように設計を行った。また、デバッグやプログラム解析機能を設け、アセンブリプログラム開発がスムーズに行くよう設計を行った。シミュレータの試作では、シミュレータを構成するモジュールや関数の設計方法について述べた。テストと評価では、シミュレータで動作させたアセンブリプログラムの実行結果を示し、シミュレータがアーキテクチャ学習やデバッガとしてだけでなく、パフォーマンスデバッガとしても使用できることを考察した。最後に、試作したシミュレータを同じ研究室の方に使って頂き、貴重な意見を頂戴した。シミュレータの画面構成、初期化の方法、及び実行モードの追加などのご指摘はこれからシミュレータを設計する上で、大変参考になった。

今後の課題は第 1 に、スーパースカラプロセッサシミュレータの設計と試作である。パイプラインよりも複雑なスーパースカラアーキテクチャをどのように分かり易く可視化するかが重要である。学習者がスーパースカラを学ぶとき、どのような情報が表示できればスムーズに理解できるかを良く検討したうえで、設計する必要がある。

第 2 の課題は、ハード/ソフト・コラーニングシステム全体の評価である。シミュレータを使用したアセンブリプログラムの実行データと FPGA ボードコンピュータでの実行時間の観点からシステムを評価する必要がある。シミュレーションデータの総クロック数に HDL で設計した各プロセッサの周期をかけることでプログラムの実行時間が計算できるが、現段階では単一サイクルプロセッサが最も実行時間が短い。これは、単一サイクルの動作周波数がマルチサイクルやパイプラインよりも高速だからである。これには、MONI 命令セットの簡易性や、マルチサイクル、及びパイプラインプロセッサのデータパスの問題点などがあるが、それらを含めたシステム全体の評価が必要である。

第 3 の課題は、シミュレータと最適化コンパイラの融合である。アセンブリプログラム

だけでなく C 言語プログラムの入力が可能になれば、組み込みソフトウェア開発ツールとして使用可能になる。高級言語のプログラムがプロセッサでどのように実行されるのかを理解できれば、高級言語によるプログラミングの意識が高まると思われる。

第 4 の課題は、命令セットが可変なプロセッサシミュレータの設計である。本研究で設計・試作したプロセッサシミュレータは命令セットが固定であるが、解こうとする問題によって最適な命令セットを定義し、その命令セットを持つプロセッサシミュレータの自動生成が行えるようにすれば、教育用としてだけでなく、実際の組み込みソフトウェア開発に大きく役立つと考える。

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、本研究に関して貴重なご意見をいただきました Tran So Cong 氏、共同研究者の池田 修久氏、中村浩一郎氏、及びシミュレータの評価に協力して頂き、色々な面で貴重な助言や励ましを下さった研究室の皆様に深く感謝いたします。

参考文献

[1] 内山邦夫:組込み型プロセッサの現状と将来,電子情報通信学会技術研究報告,Vol.99, No.7(FTS99 1-5), pp.23-29 ,1999.04.

[2] GONZALEZ R E: Xtensa: A Configurable and Extensible Processor ,IEEE Micro, Vol.20, No.2, pp.60-64, 65-70 , 2000.03.

[3] 田中茂, 樋渡有:SOC 開発プロジェクトを成功させるには… 激戦区を戦い抜くための LSI 設計術, 第 1 章 システム LSI の設計マネージメント 複雑になるプロジェクトをいかに管理･運用していくか, Design Wave Magazine,Vol.7, No.8, pp.36-41 ,2002.08.01.

[4] 田中茂:SOC 開発プロジェクトを成功させるには… 激戦区を戦い抜くための LSI 設計術, 第 2 章 システム LSI のプラットホーム事例,東芝コンフィギャラブル･プロセッサ「MeP」の概要, Design Wave Magazine,Vol.7, No.8, pp.42-53 ,2002.08.

[5] HOFFMANN A, and KOGEL T, NOHL A, BRAUN G, SCHLIEBUSCH O, WAHLEN O, WIEFERINK A, MEYR H :A Novel Methodology for the Design of Application-Specific Instruction-Set Processors(ASIPs) Using a Machine Description Language,IEEE Trans Comput-Aided Design Integrated Circuits System,Vol.20, No.11, pp.1338-1354 ,2001.11.

[6] 今井正治:高位言語ベースデザイン特論 第 4 章講義資料,立命館大学大学院 STARC 寄附講座,2003.

[7] 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.

[8] 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.

[9] 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.

[10]末吉, 小羽田, 野崎, 田中, 久我:FPGA を利用した教育用マイクロプロセッサ KITE-2 システムソフトウェア教育への対応情報処理学会研究報告,Vol.94, No.50(ARC-106), pp.25-32 ,1994.06.13.

[11]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.

[12]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), pp.47-54 ,2001.10.19.

[13]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.
[14]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.
[15]DLX-View:http://yara.ecn.purdue.edu/~teamaaa/dlxview/Index.html.
[16]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.
[17]Burger, D. and Austin, T.M: The SimpleScalar Tool Set, Version 2.0,University of Wisconsin-Madison Computer Science Department Techical Report,#1342,June,1997.
[18]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), pp.7-12 ,2003.03.07.
[19]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.
[20]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.
[21]林晴比古著:新 VisualC++6.0 入門シニア編, ソフトバンクパブリッシング, 2001.
[22]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.
[23]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.
[24]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.
[25]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.
[26]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.
[27]池田,中村,Tran:設計仕様所 品種名:RC100 を用いたボードコンピュータ,2004.

付録 A シミュレータ実行例

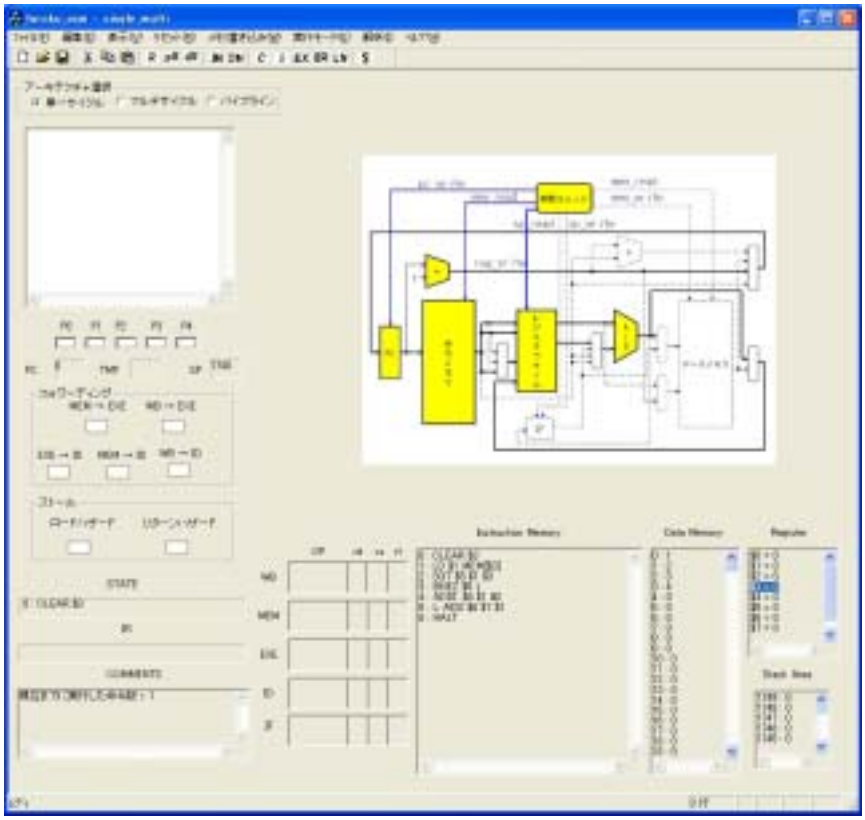
A.1 サンプルプログラム

```
        CLEAR $3
        LD $1 MEM [$3]
        SGT $5 $1 $3
        BNEZ $5 L
        ADDI $6 $1 #3
L:      ADD $6 $1 $1
        HALT
```

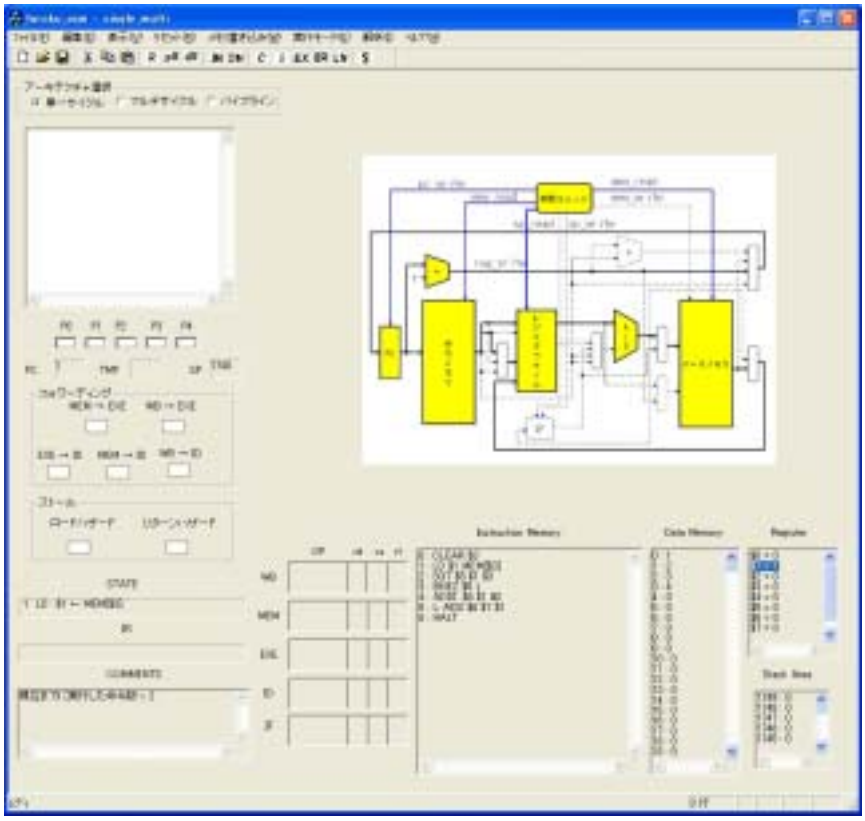
A.2 単一サイクル

1 クロック実行（1 命令実行）を行った際の実行画面を示す。

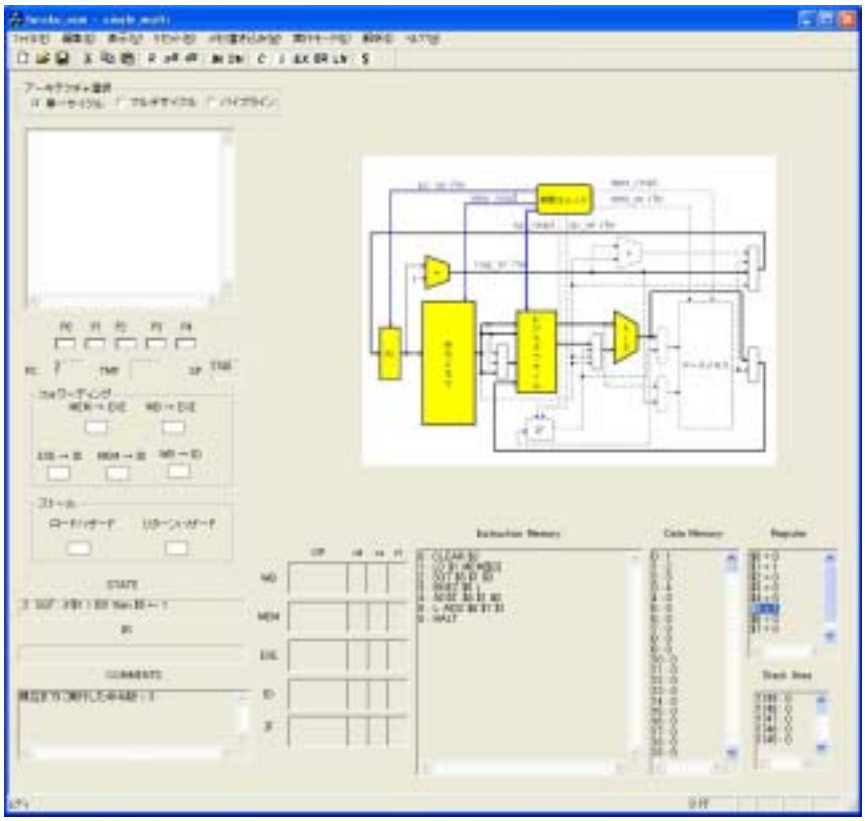
1 クロック目：CLEAR $3



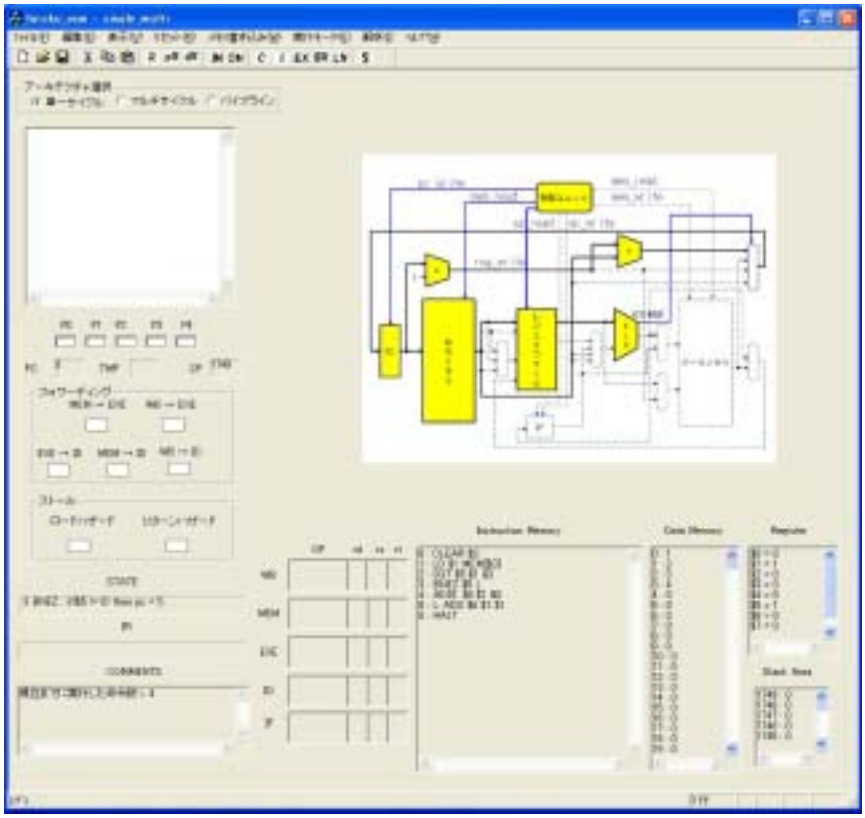
2 クロック目：LD $1 MEM [$3]



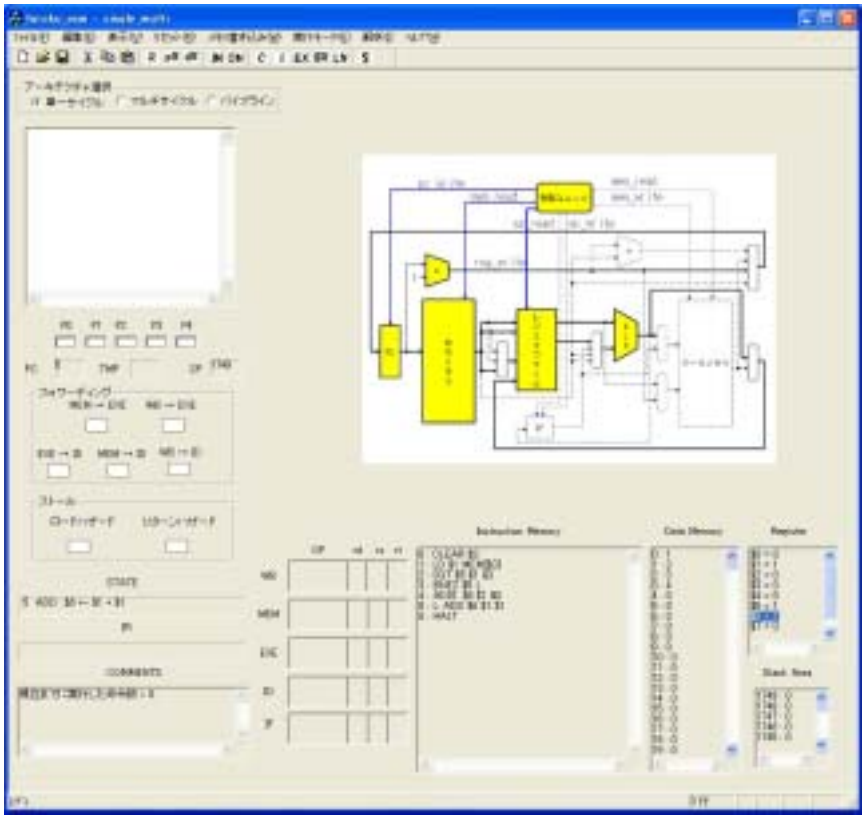
3 クロック目：SGT $5 $1 $3



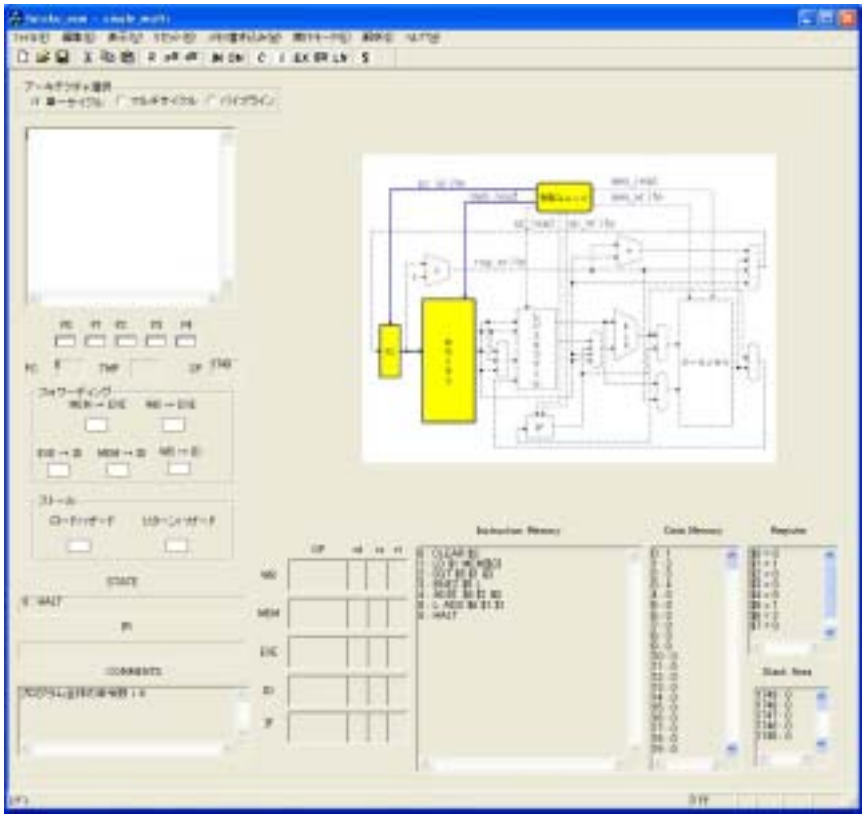
4 クロック目：BNEZ $5 L



5 クロック目：ADD $6 $1 $1



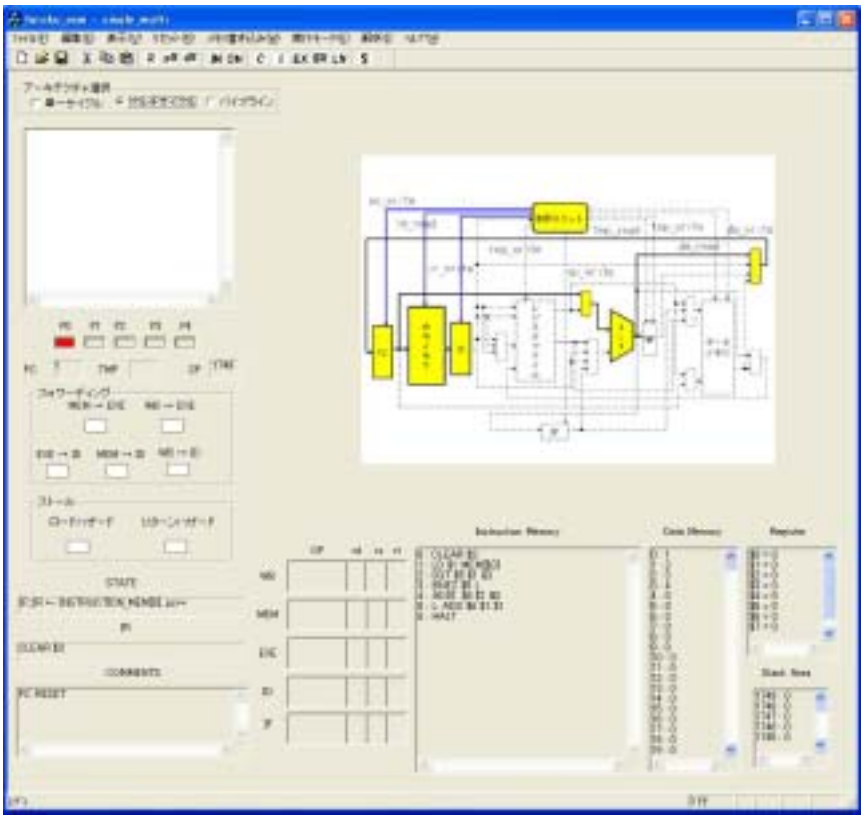
6 クロック目：HALT



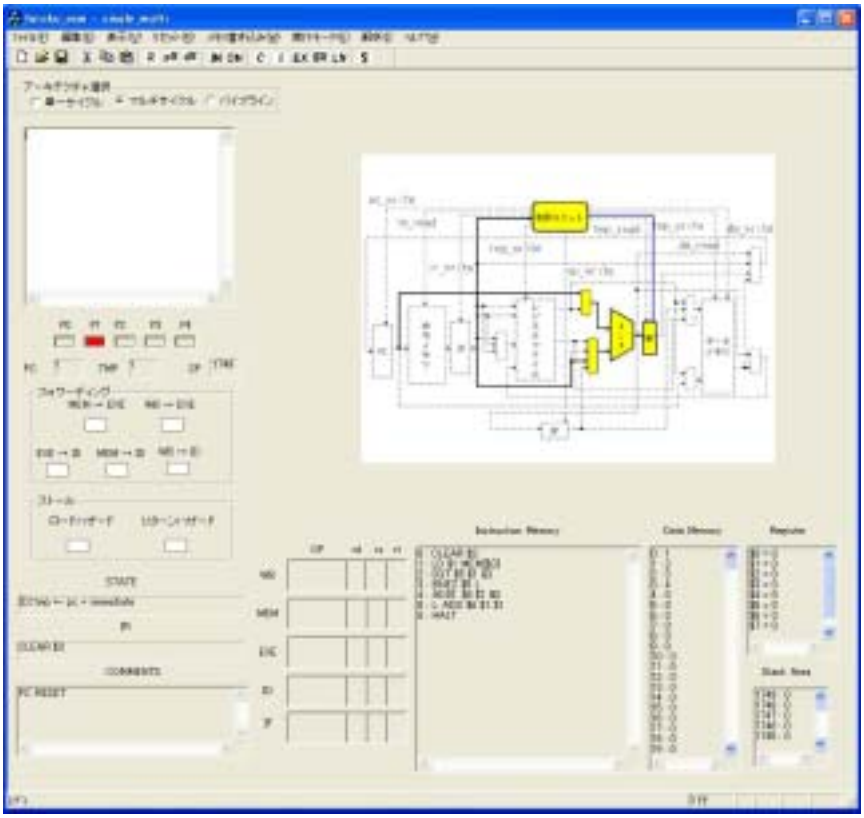
A.3 マルチサイクル

1 クロック実行を行った際の実行画面を示す。IF ステップ（命令フェッチ）と ID ステップ（命令デコード）は全ての命令において共通である。IF ステップと ID ステップ以降は、各命令が異なる動作をする 3 ステップ目からの実行画面となっている。

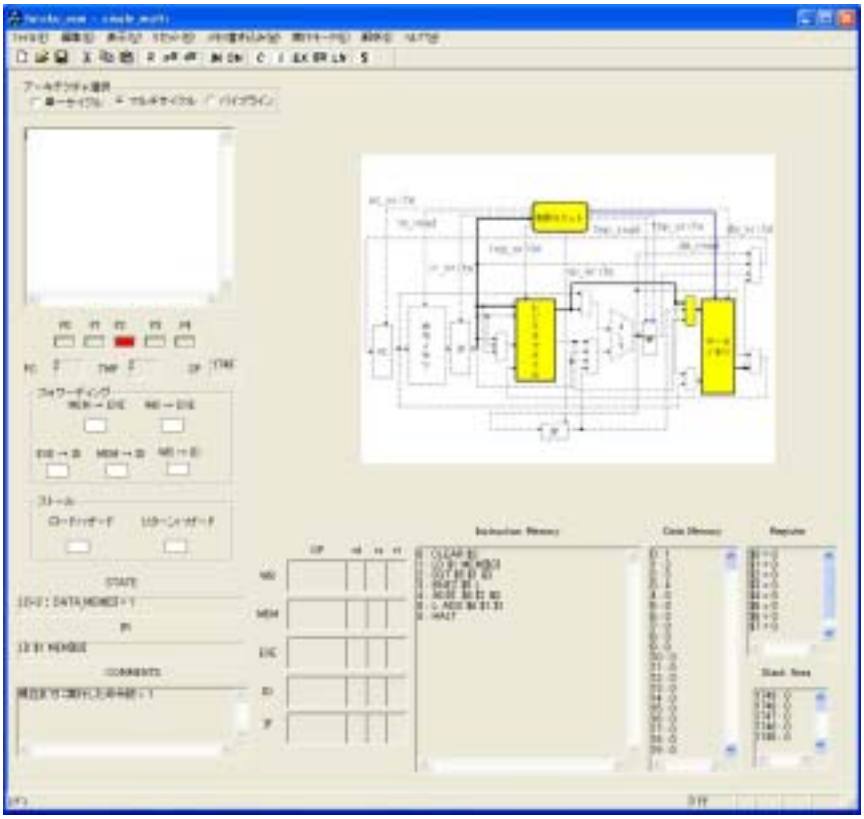
IF ステップ



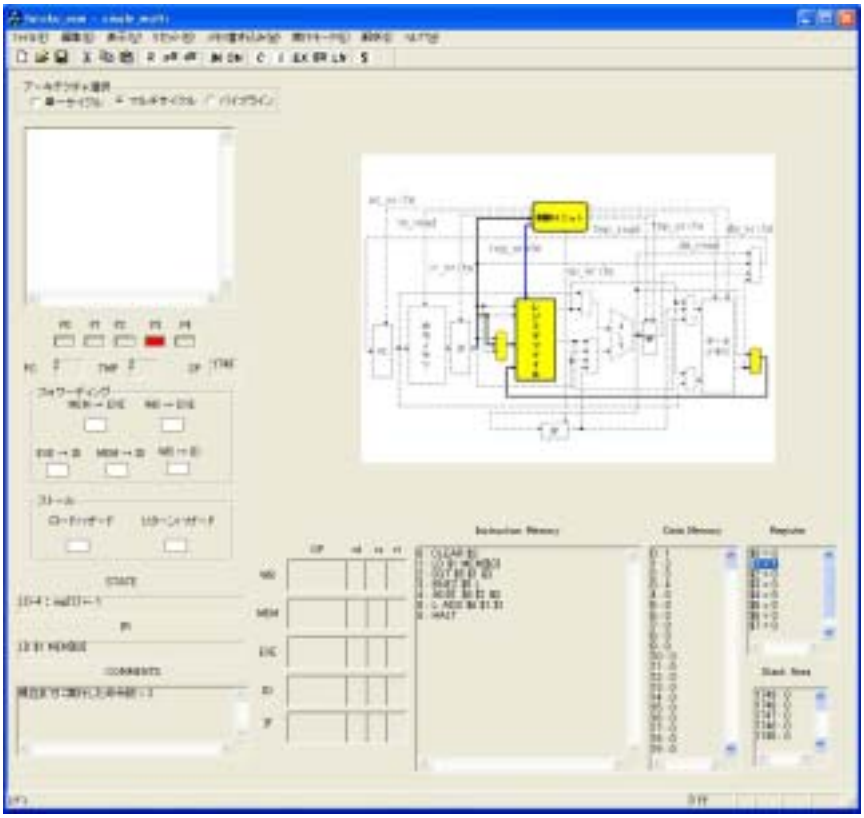
ID ステップ



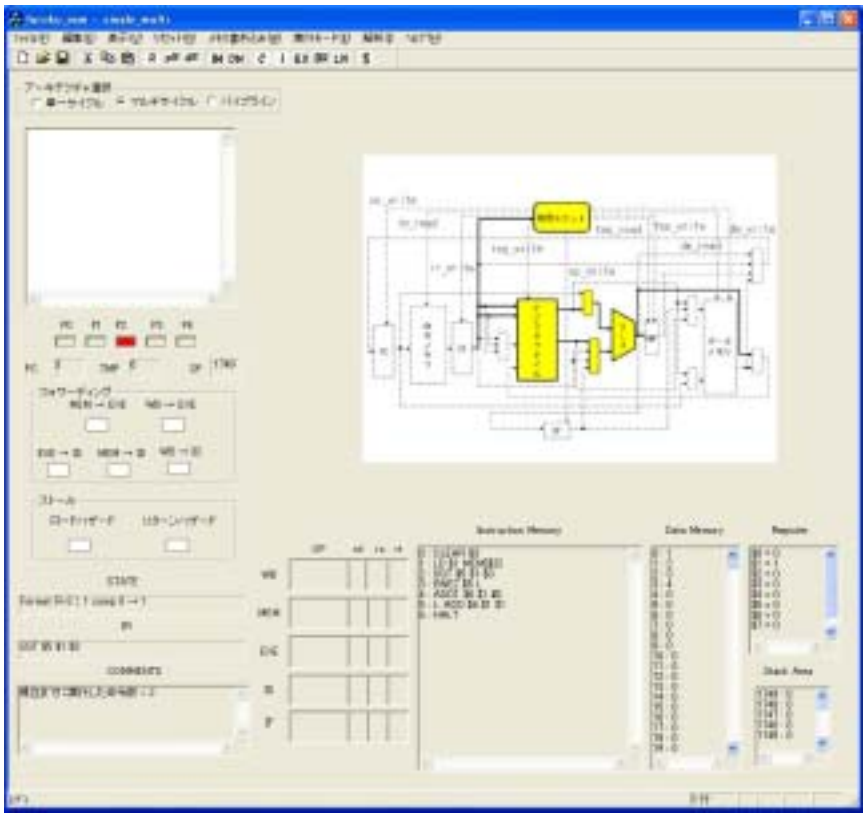
LD $1 MEM[$3]（3 ステップ目）



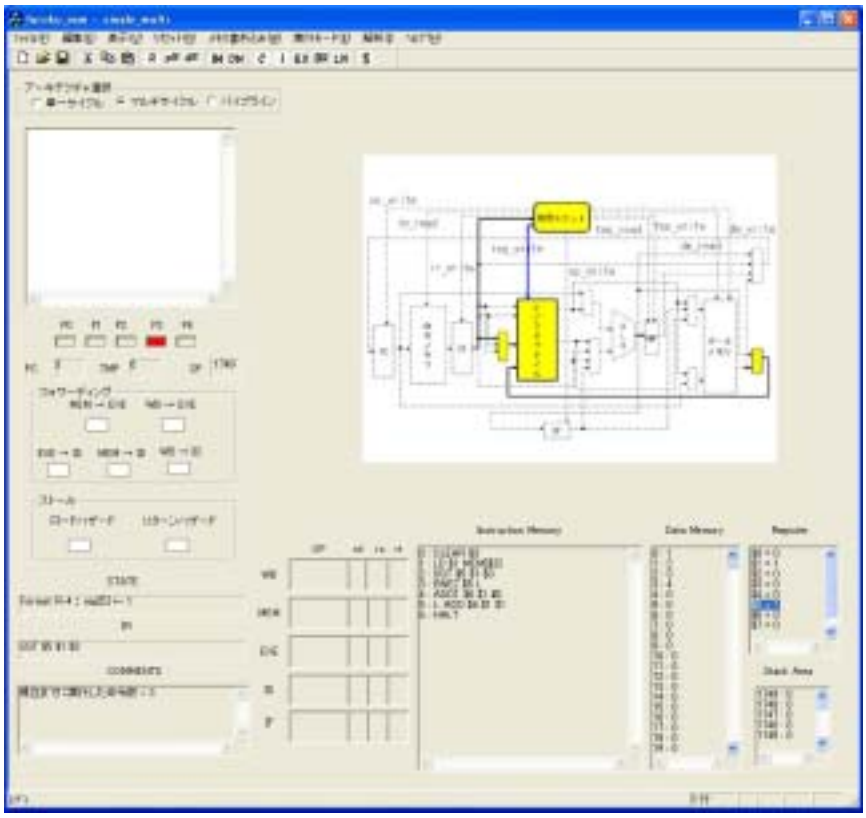
LD $1 MEM[$3]（4 ステップ目）



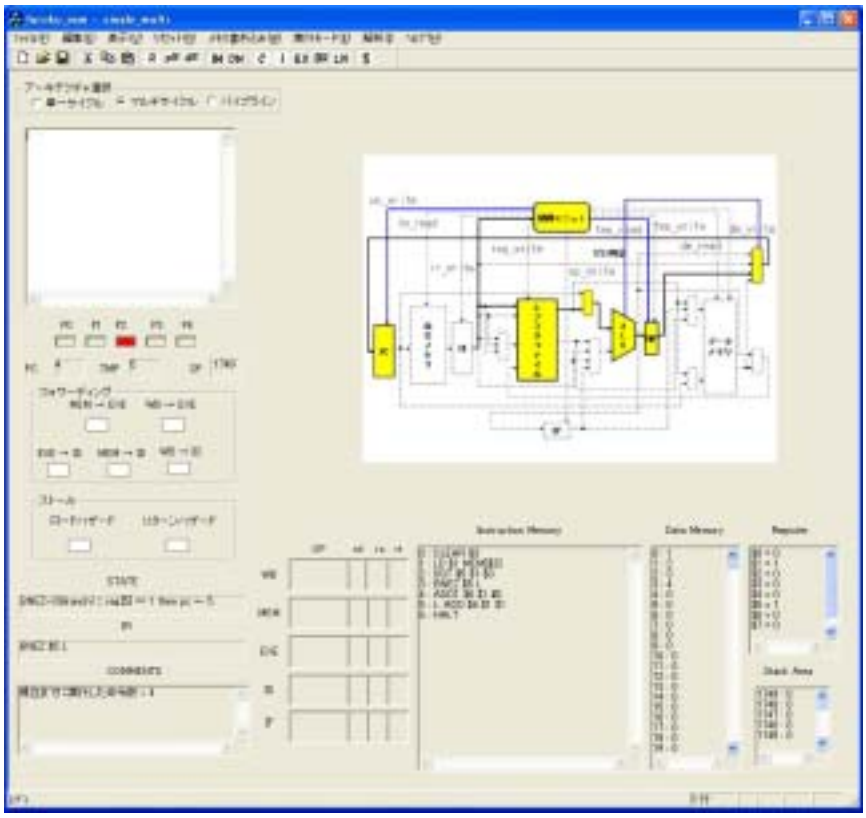
SGT $5 $1 $3（3 ステップ目）



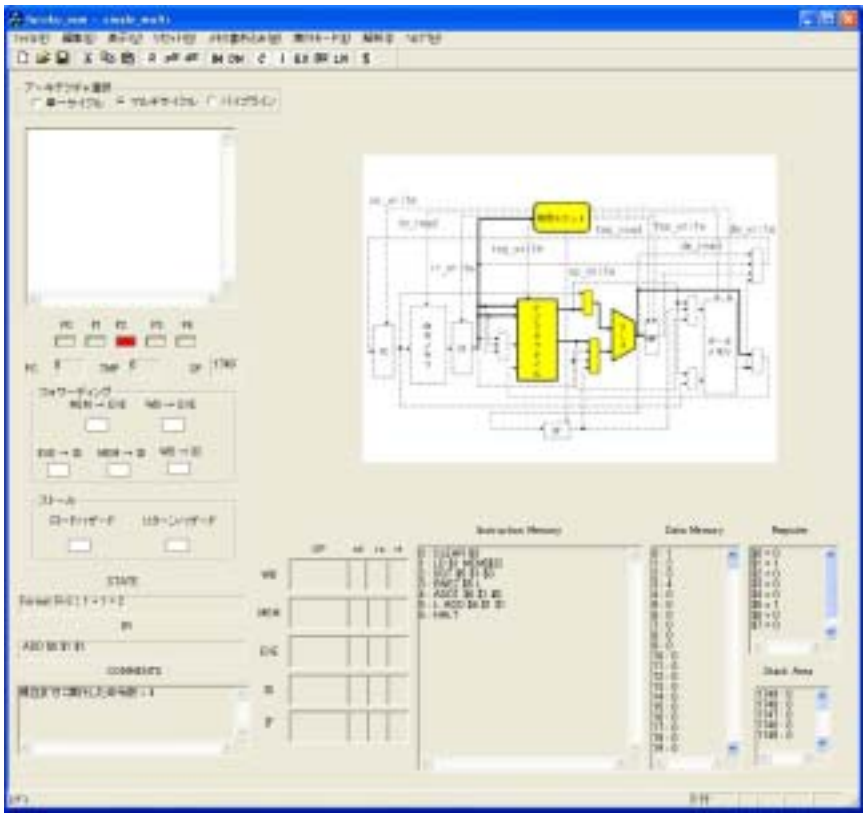
SGT $5 $1 $3（4 ステップ目）



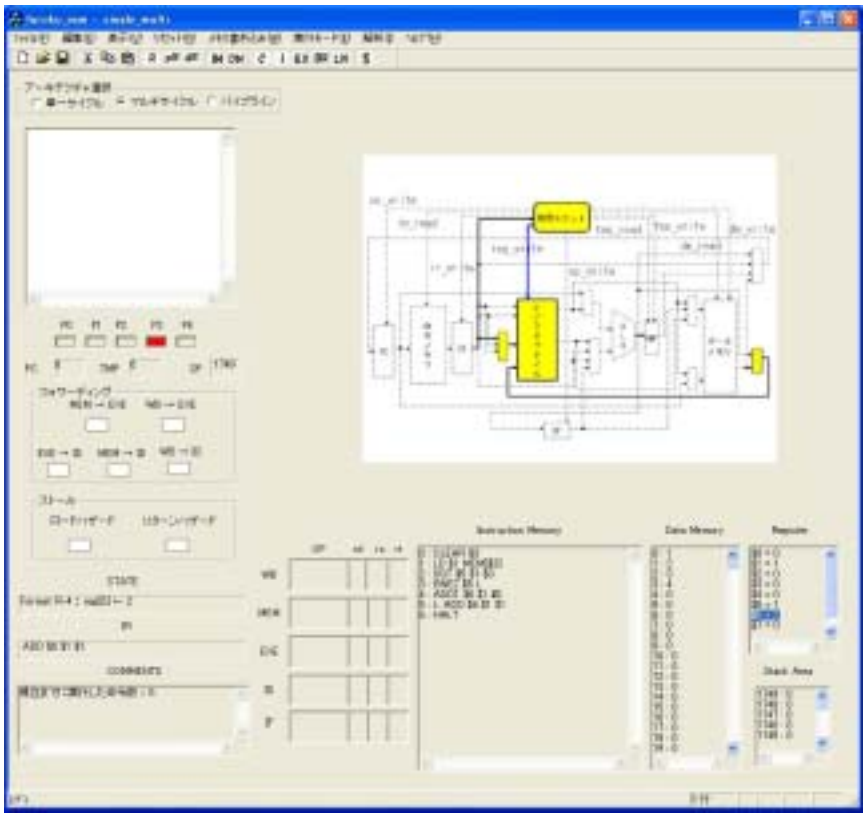
BEQZ $5 L（3 ステップ目）



ADD $6 $1 $1（3 ステップ目）



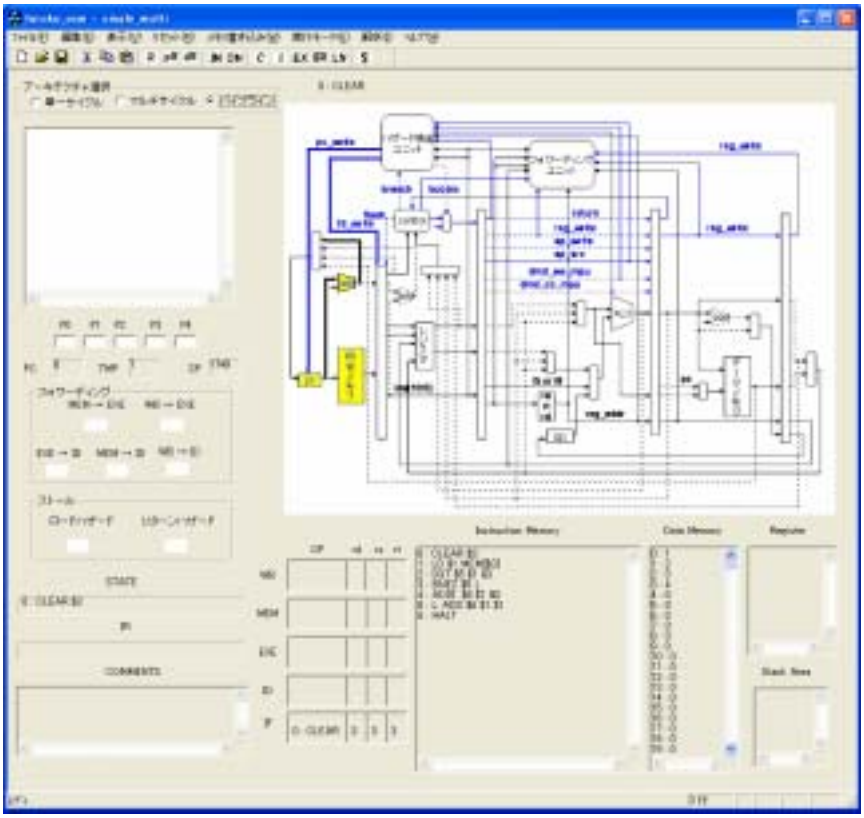
ADD $6 $1 $1（4 ステップ目）



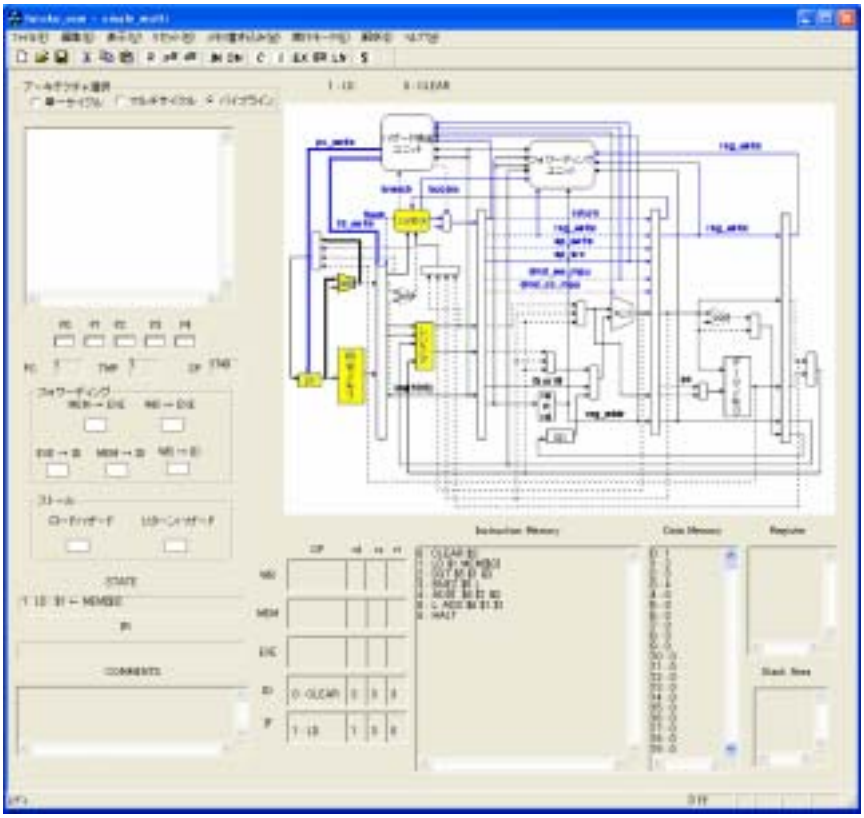
A.4 パイプライン

1 クロック実行を行った際の実行が面を示す。

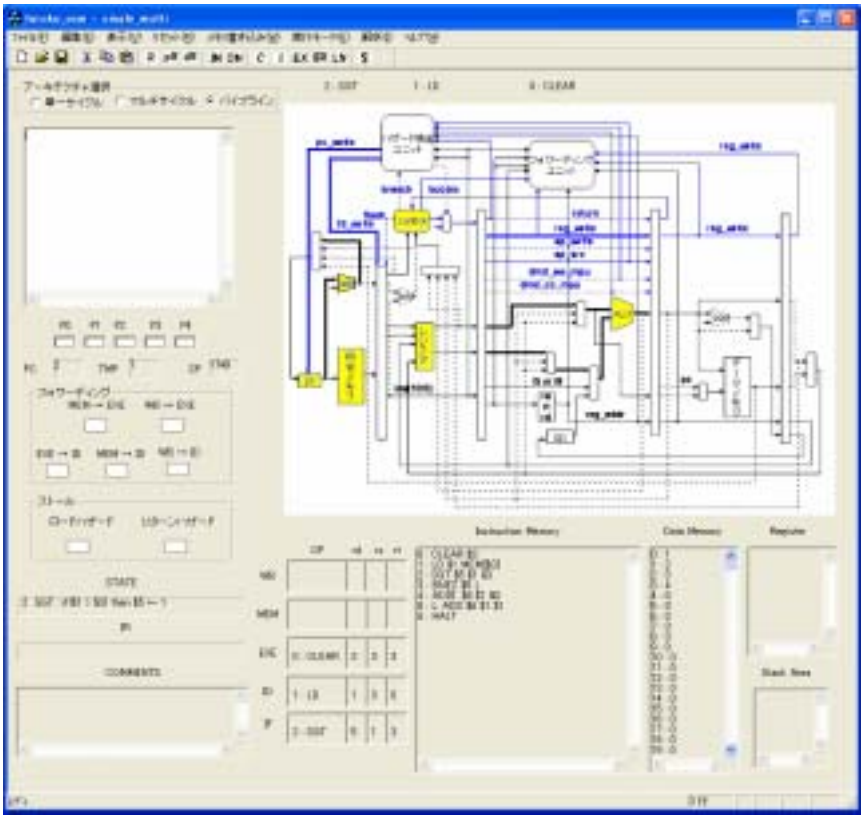
1 クロック目



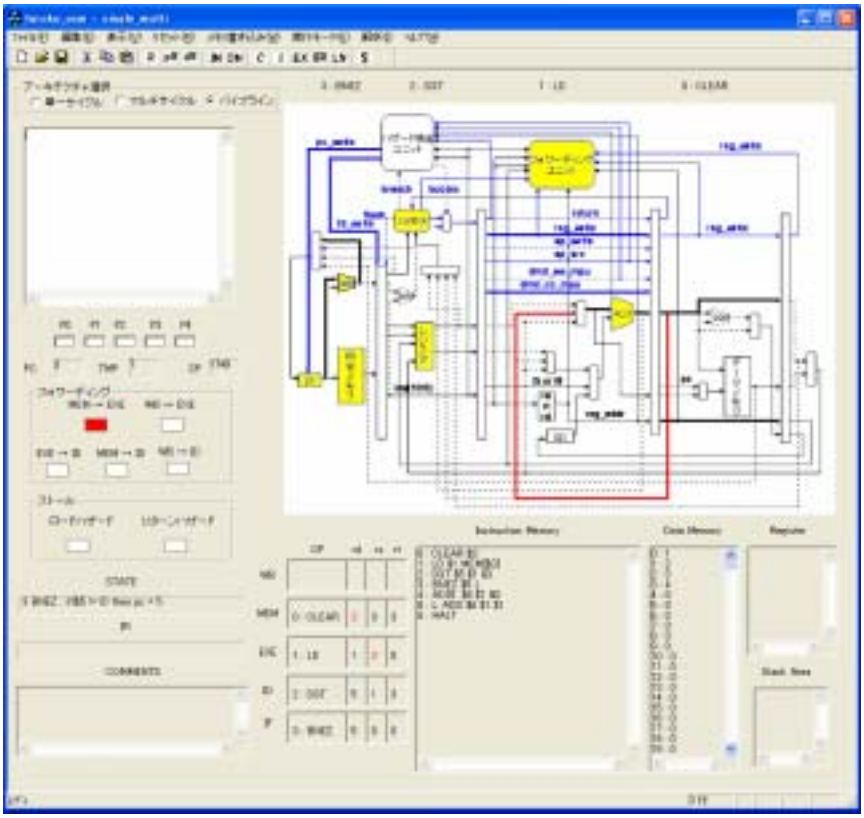
2 クロック目



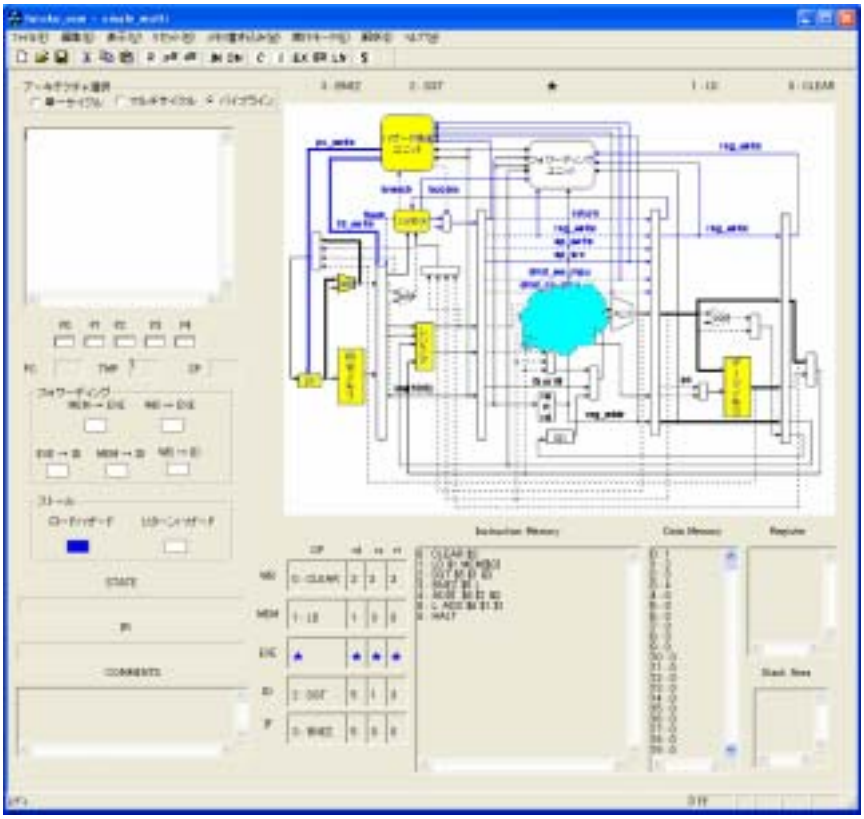
3 クロック目



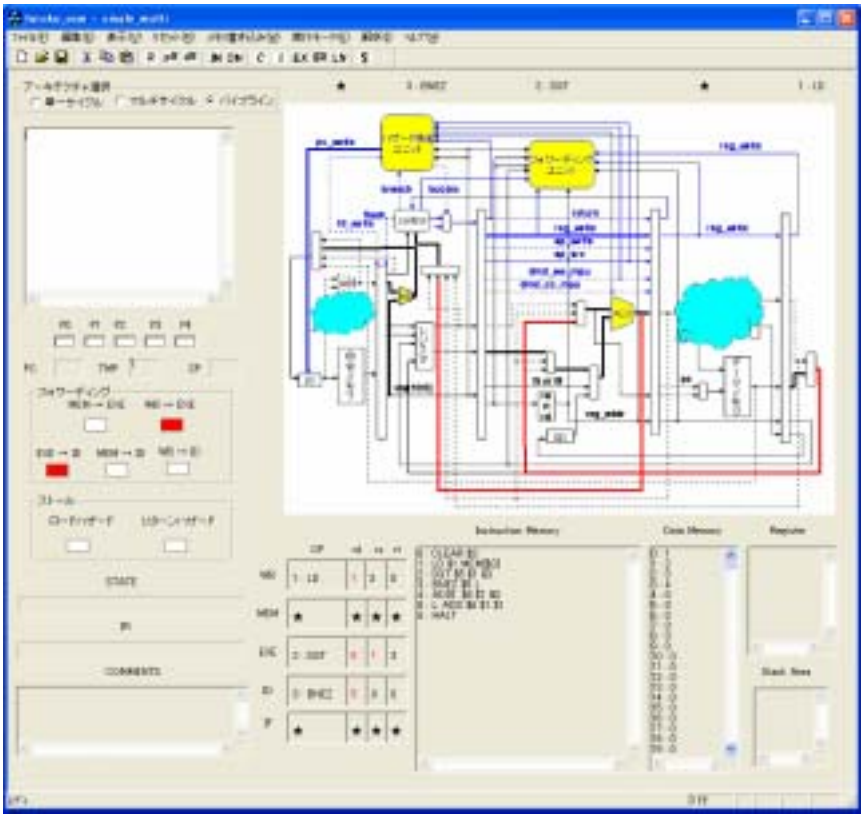
4 クロック目



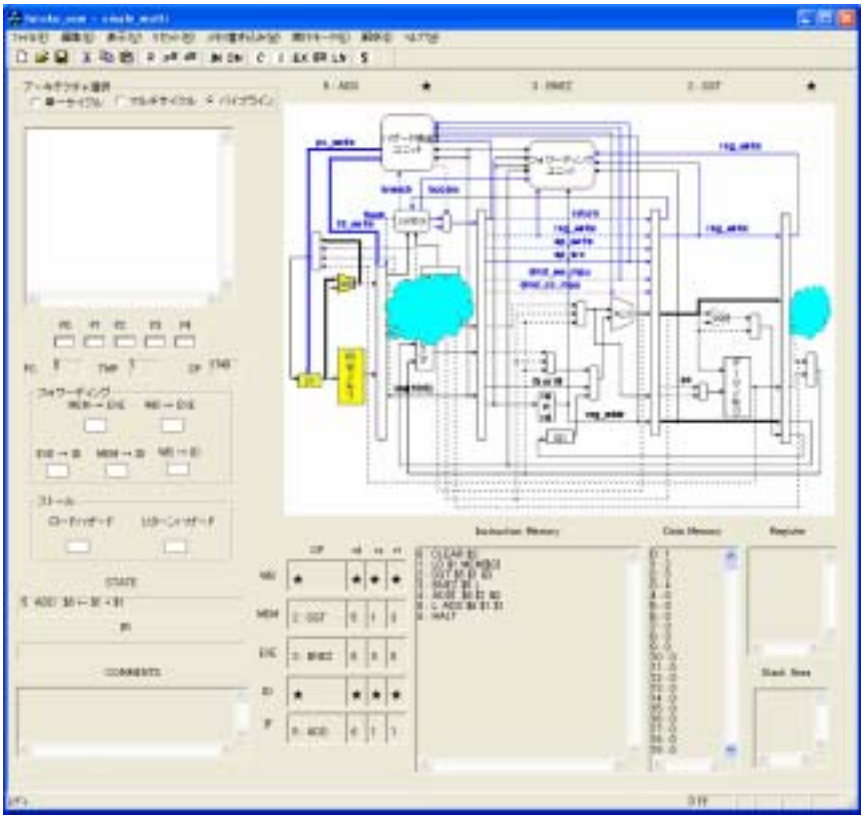
5 クロック目



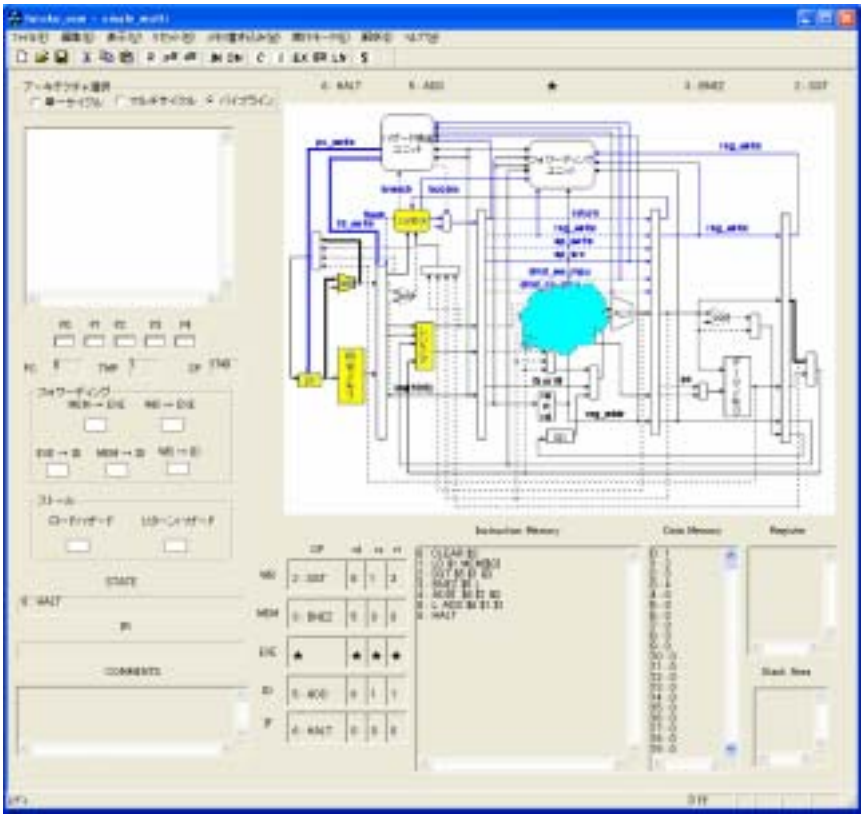
6 クロック目



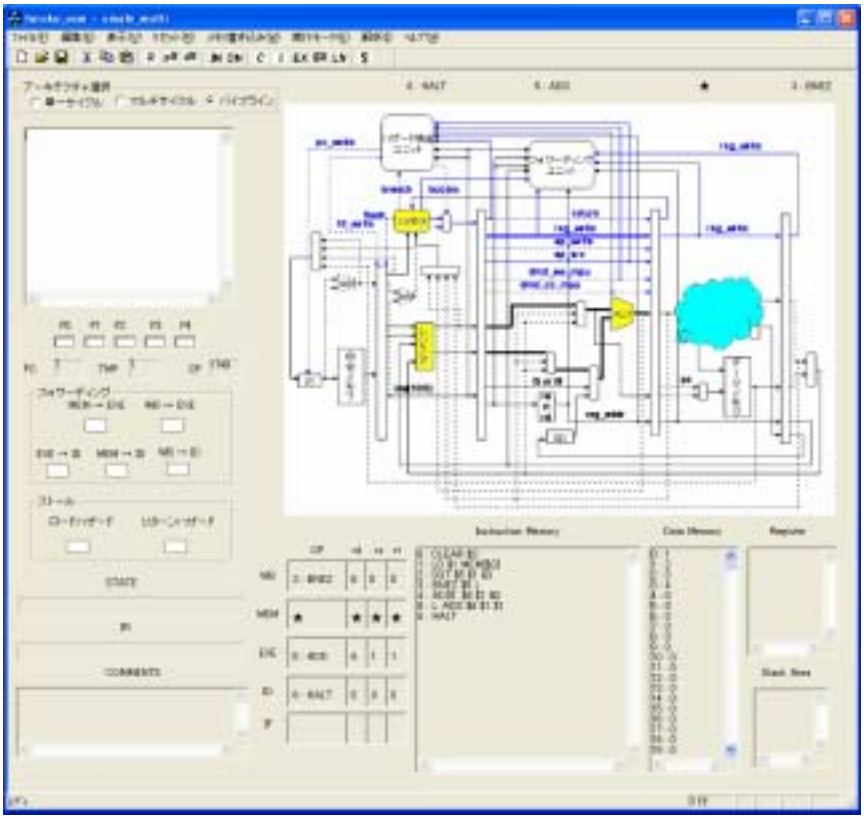
7 クロック目



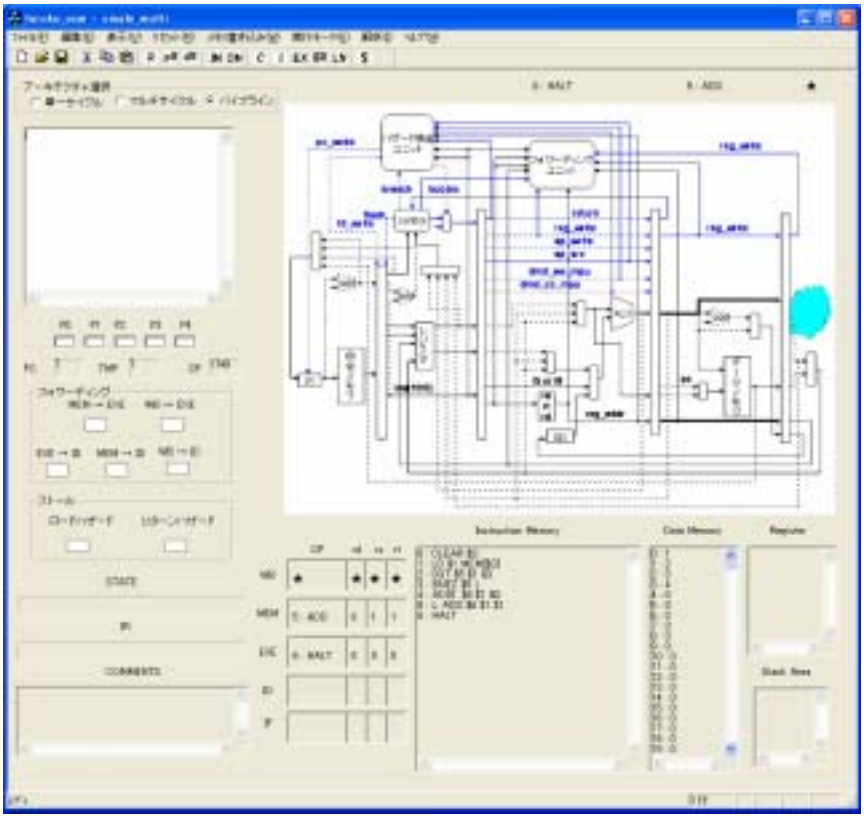
8 クロック目



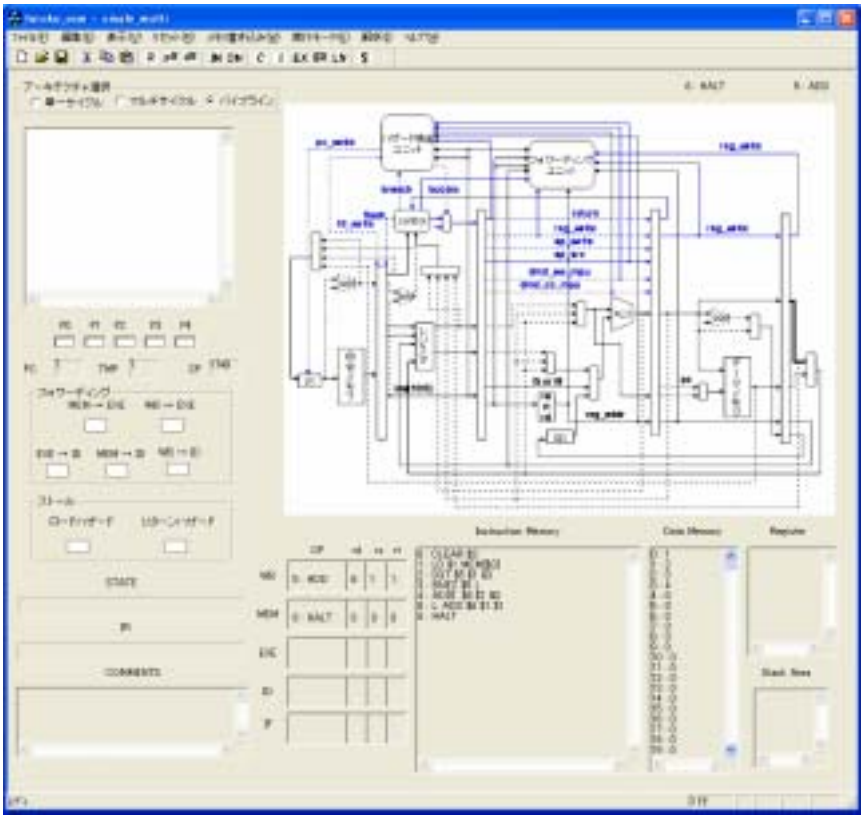
9 クロック目



10 クロック目



11 クロック目



12 クロック目