修士論文

ハード/ソフト協調学習での
プロセッサアーキテクチャ可変な最適化コンパイラの設計

氏名：藤原淳平
学籍番号：6124030198-8
指導教員：山崎勝弘教授
提出日：2005 年 2 月 15 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、プロセッサアーキテクチャ可変な最適化コンパイラの設計とその利用方法の検討について述べる。

プロセッサアーキテクチャ可変な最適化コンパイラの設計は、教育上必要十分な C 言語のサブセット(CMONI)を定義し、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）を対象としたコンパイラである。対象であるプロセッサは単一サイクルプロセッサ、マルチサイクルプロセッサ、パイプラインプロセッサ及びスーパースカラプロセッサである。字句解析、コンパイラコンパイラ(yacc)を使用し作成した構文解析、意味解析、教育用としてどの程度の最適化が必要かなどを考え単一サイクルプロセッサ、マルチサイクルプロセッサの最適化、コード生成というコンパイラ一連の設計を行い正常に動作を確認した。また、パイプラインプロセッサ。スーパースカラプロセッサは各プロセッサ特有の最適化手法の提案をした。性能評価として複数のテストプログラムをＣＭＯＮＩ言語でプログラミングしコンパイラを通したＭＯＮＩコードと、直接ＭＯＮＩコードでプログラミングしたものをシミュレータで実行し比較を行った。その結果、直接ＭＯＮＩコードでプログラミングしたテストプログラムの方がよい結果になった。これは、直接ＭＯＮＩコードでプログラミングしたため、対象とするテストプログラムの規模があまり大きくなかったことと、教育用として最適化部を考えたため最適化は局所的なものしか行っていないというのが原因として挙げられる。さらに教育用と最適化の性能のトレードオフを見極めよりよい最適化が今後の課題である。

この結果を使用し、このプロセッサアーキテクチャ可変な最適化コンパイラの教育上での利用方法の検討を行った。目標として、プロセッサアーキテクチャを意識した高級言語プログラミングの習得を掲げ、いかにこの目標にたどり着けるようにするかそのプロセスを提案した。

目次

図目次

表目次

１．はじめに

近年の急速な半導体集積技術の進歩によって、1 チップ上にマイクロプロセッサ、メモリ、ASIC、アナログ回路など搭載することができる。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[16]〜[18]。さらに、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っている。

ハード／ソフト・コデザイン[19]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である。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要である。[20]

システム LSI に搭載するプロセッサは、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、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。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る設計が重要視され求められている。

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、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。

本研究では、ハード/ソフト・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における最適化コンパイラの設計を行う。ハード/ソフト・コラーニングシステムは HDL によるプロセッサ設計と FPGA ボード上での実機検証、ソフトウェアシミュレータによる動作理解、最適化コンパイラの設計によるソフトウェア側からの理解の大きく３つ柱を中心とした教育システムであり、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、最適化コンパイラの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）を対象としており、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。

ソフトウェア開発時において、コンピュータの構造やアセンブラの知識を十分に持つソフトウェア開発者と、ハードウェア側の知識が全く無いソフトウェア開発者が記述したプログラムは、同一の動作を行ったとしても実行速度やメモリの使用量など大きな差が出る。そこで最適化コンパイラを設計することによって、プロセッサとコンパイラが協力することが高速な処理を実現するために必要であることを認識させ、ソフトウェア開発においても常にハードウェア側のことを意識し、より効果的、効率的なプログラミングを習得することができる。

本研究では、学習システムの構成要素であるプロセッサアーキテクチャ可変な最適化コンパイラの設計を行う。目的は、ソースプログラムがどのように機械コードへ変換され、またプロセッサによって生成された機械コードがどのように違うか、なぜ違うのかなどの理解の支援と、その生成された機械コードをシミュレータで動作させることでよりわかりやすく理解できる援助を目指し、ハードウェアを意識した高級言語プログラミングができるようにする。そして、ハードウェアとソフトウェアの両方が分かる人材の育成を目指す教育用コンパイラである。

関連研究を以下に示す。教育用コンパイラでは慶応義塾大学で開発された MiniL コンパイラ[1]があり、使用できるデータは整数、文字、実数であり、演算用にスタックを備えた仮想マシン(VSM―E)を対象とし、独自の言語である MiniL を定義し、字句解析、構文解析と順を追いながらコンパイラの完成を目指す。ハード/ソフト協調学習システムでは、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「港」[21]がある。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。また、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[22]。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。

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ア

ーキテクチャ可変な最適化コンパイラ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、対象とするプロセッサアーキテクチャについて述べ、3 章では最適化コンパイラの設計手順、対象とする C サブセットについて述べる。4 章では最適化コンパイラのフロントエンド部の設計、5 章で最適化コンパイラのバックエンド部の設計と動作検証について述べる。6 章では教育システムでの利用方法の検討を述べ、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　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
- ソフトウェアシミュレータによる可観測性と最適化コンパイラの設計を融合
  - 高級言語から実際にどのようにプロセッサが動作するか直感的に学べる
  - シミュレータからどのような最適化が必要か理解を促せる
- 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

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。

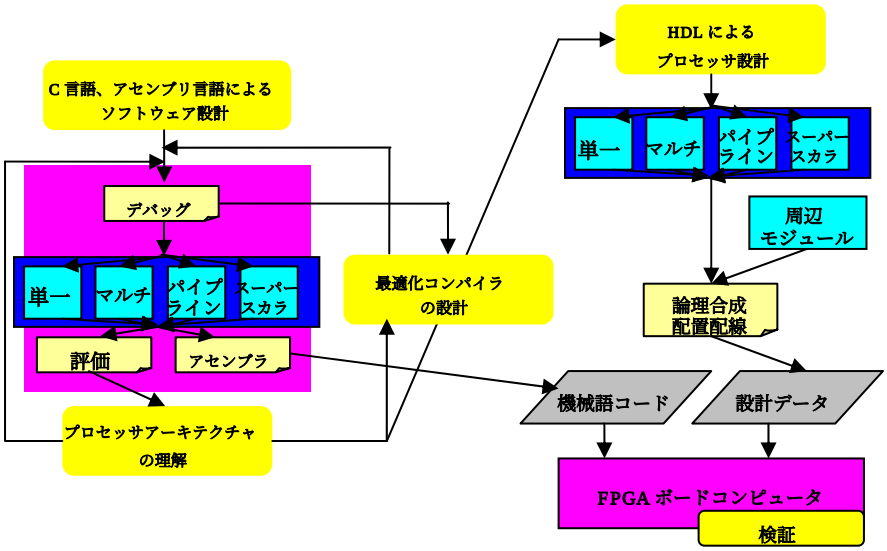

図 1：ハード/ソフト・コラーニングシステム

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。

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。

2.2 最適化コンパイラの位置づけ

ソフトウェア開発において、コンピュータの構造やアセンブラの知識を持つ経験豊富なソフトウェア設計者が記述したプログラムと、何も知らない新人の設計者が記述したプログラムは、同一の動作を行っても、実行速度やプログラム量、操作性、実行時に使用するメモリに大きな差が出る。最適化コンパイラ作成の目的は、教育用として本コン

パイラの利用者に、プロセッサとコンパイラが協力することが高速な処理を実現するために必要であることと、高級言語と機械語の関係、対象とするプロセッサアーキテクチャの違いでどのような最適化が必要かなどの理解を促し、コンパイラの理解やプロセッサアーキテクチャの理解をへてソフト側からでもハード側のことを意識できるようにする。そして、プロセッサアーキテクチャを意識した高級言語プログラミングを目指すものである。

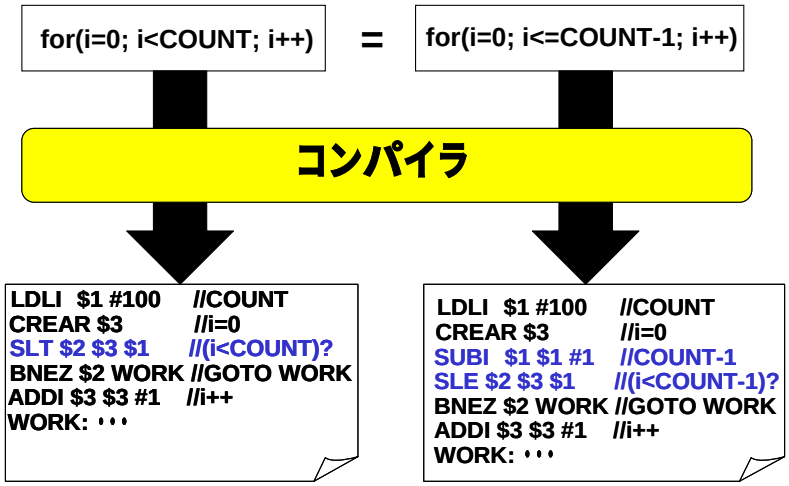

図２：アーキテクチャを意識したプログラミング

例えば図２を例にとると同じ意味の高級言語プログラムを書いたとしても、実際に機械語にコンパイルすると 1 命令余分に増えていることが分かる。この 1 命令の積み重ねが性能低下を招くそのため、ハードウェア側を意識してプログラミングできることが重要なのである。以下に最適化コンパイラの作成目的を図示する。

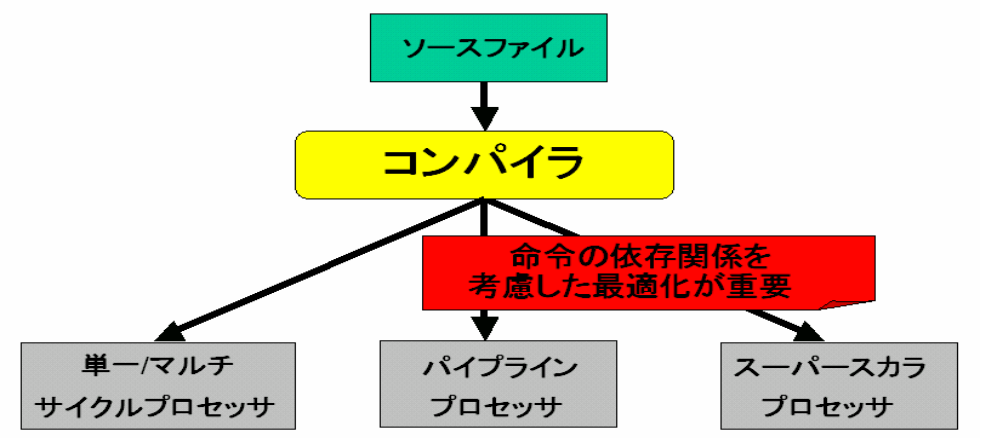

図３：最適化コンパイラの作成目的

2.3 MONI プロセッサアーキテクチャ

2.3.1 命令セットアーキテクチャ

教育用としてふさわしい、コンパクト且つ必要十分な命令数を目指す。16 ビットの命令語長でレジスタ間演算には 3 オペランド形式を取り 43 命令ある。コラーニングシステムの目標はアーキテクチャを理解することにあるので、割り込み等の命令は用意していない。以下に全命令を示す[9]。

● 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(ADDI, 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。

2.3.2 MONI 単一サイクルアーキテクチャ

単一サイクルプロセッサは、1 つの命令を 1 つのクロック･サイクルで実行完了しようとするものである。これは、どのデータパス資源も 1 クロック･サイクル中に 2 回以上の使用は出来ない事を意味する。つまり、2 回以上必要となる要素は必要個数分だけ別個に設けなければならない。図４ に単一サイクル方式実行のデータパスを示す[5]。

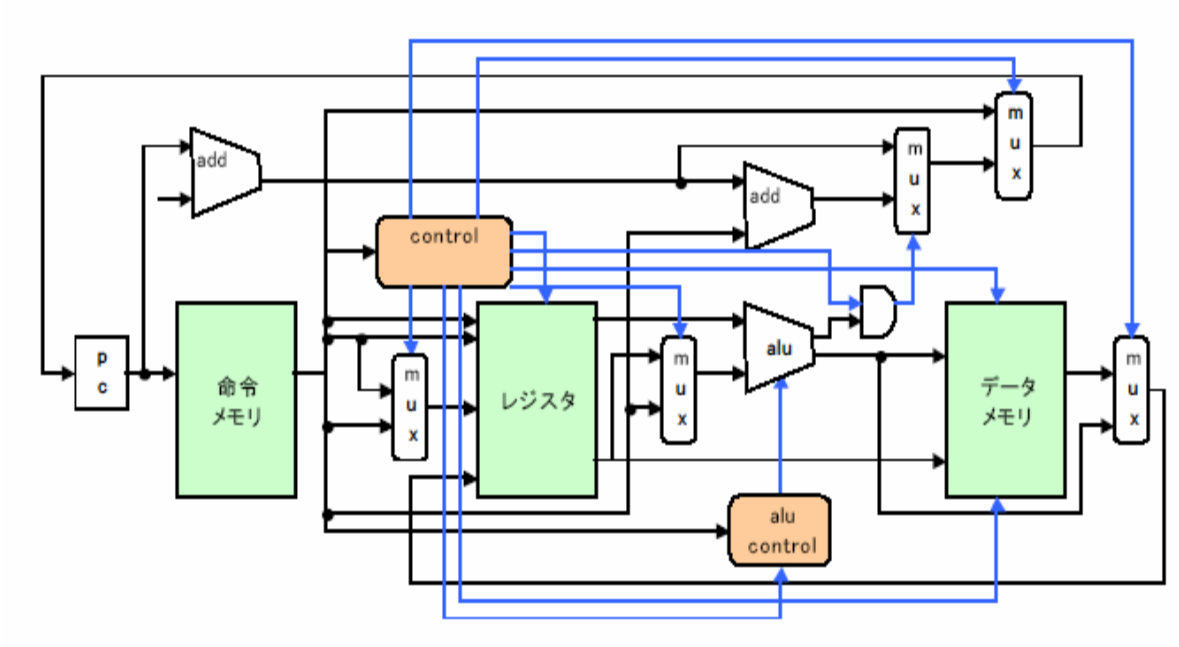

図４：単一サイクルプロセッサのデータパス

単一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
- レジスタファイル
- ALU：算術論理演算、プログラムカウンタ算出
- 制御ユニット

2.3.3　MONI マルチサイクルアーキテクチャ

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 5 にマルチサイクルプロセッサのデータパスを示す[6]。

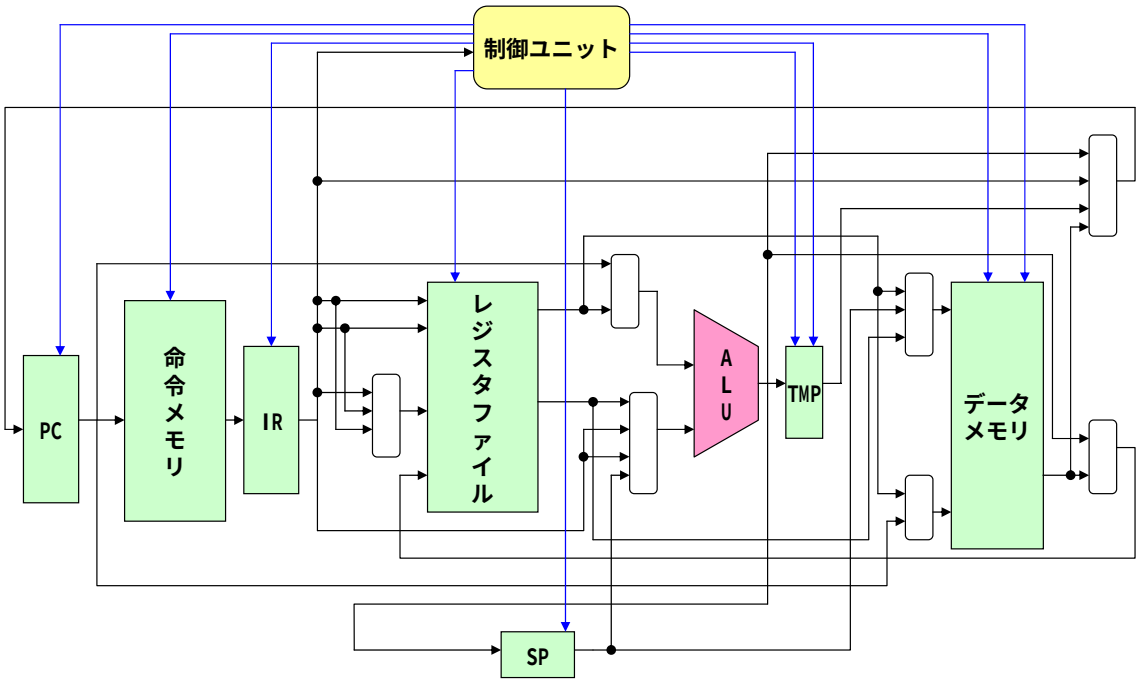

図５：マルチサイクルプロセッサのデータパス

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

2.3.4　MONI パイプラインアーキテクチャ

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。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。図 6 にパイプラインプロセッサのデータパスを示す[6]。

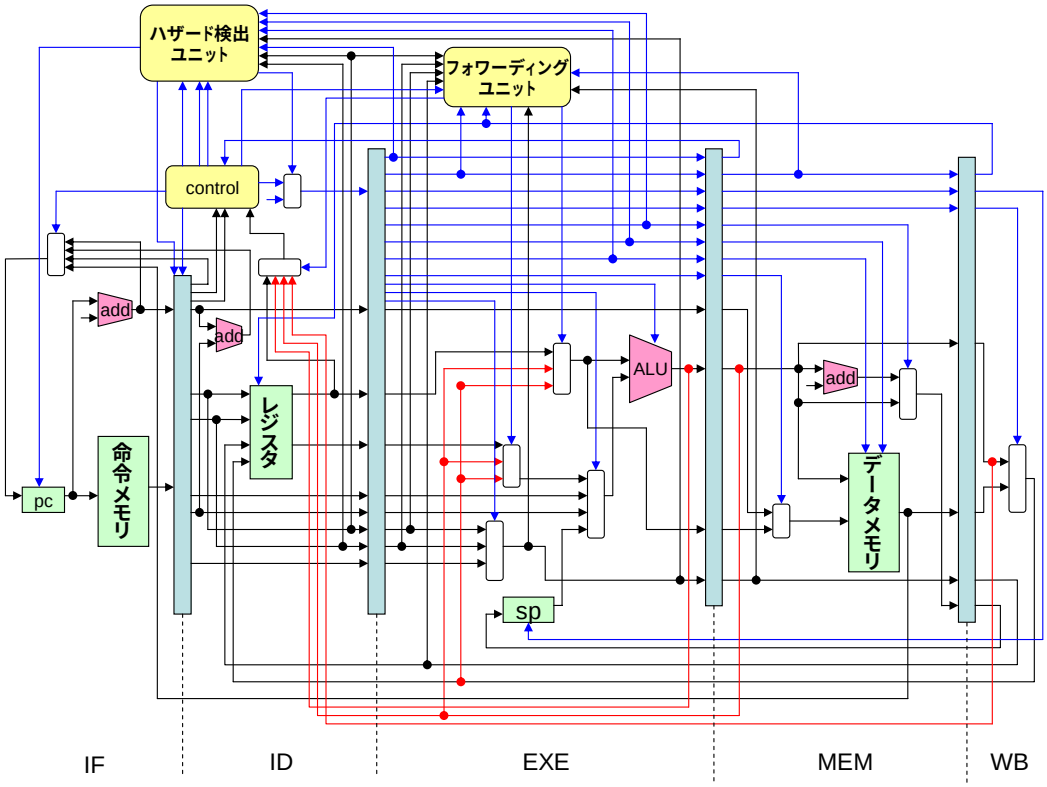

図６：パイプラインプロセッサのデータパス

以下に、各ステージの処理を示す。

IF ステージ：命令フェッチ

ID ステージ：命令デコード、条件分岐判定

EXE ステージ：ALU による演算

MEM ステージ：メモリアクセス

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。

2.3.5　MONI スーパースカラアーキテクチャ

スーパースカラプロセッサとは、同じパイプラインステージ内で複数の命令を並列実行する方式である。すなわち、パイプラインプロセッサと同様に命令フェッチ、命令デコード、実行、メモリアクセス、ライトバックの５ステージからなる。この一連の流れを 2 命令並行に処理するものである。

命令はデコーダでの発行、演算器での処理、実行完了の過程を経て処理される。命令の発行はプログラムに書かれた順序で行うこともでき、矛盾が無ければ、プログラムの順序を無視して行うこともできる。処理が順序通りであることをインオーダ、順序と異なることをアウトオブオーダと呼ぶ。そのためスーパースカラ方式はインオーダ発行インオーダ完了、インオーダ発行アウトオブオーダ完了、アウトオブオーダ発行インオーダ完了、アウトオブオーダ発行アウトオブオーダ完了の 4 種類に分類することができる。今回の対象はインオーダ発行アウトオブオーダ完了のスーパースカラプロセッサである。図 7 にスーパースカラのブロック図を示す[15]。

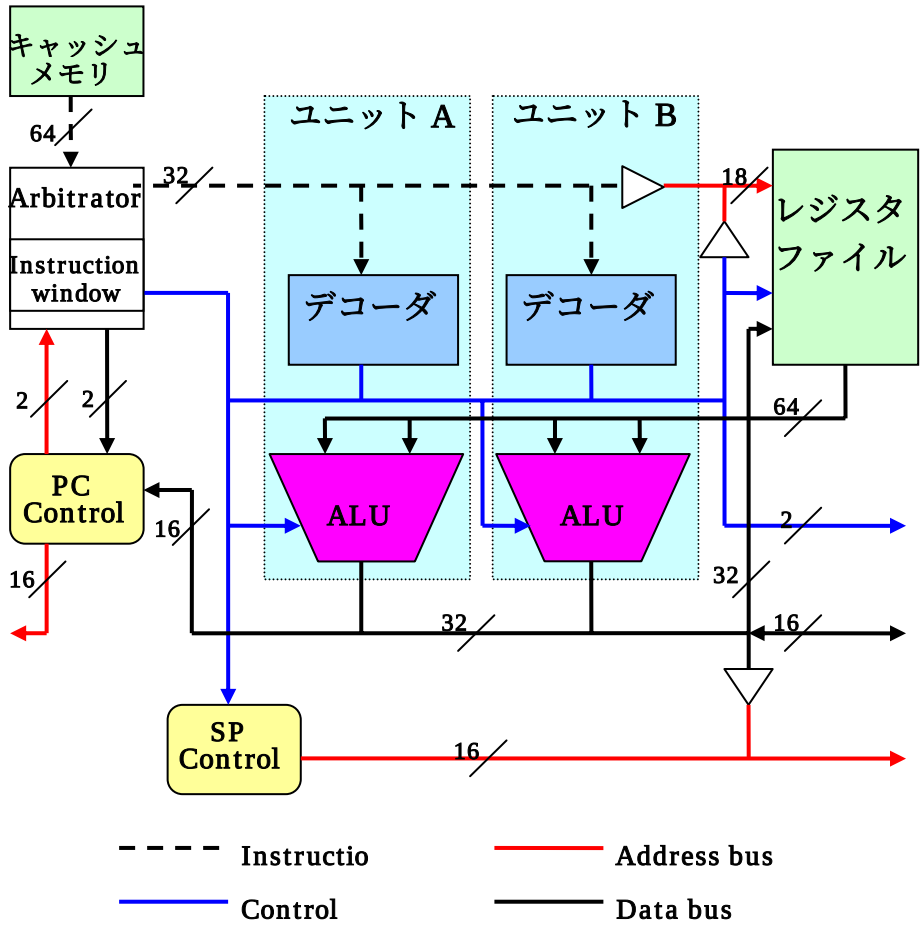

図７：スーパースカラプロセッサのブロック図

スーパースカラプロセッサは常に 2 命令並行に実行できるわけではない。以下に 2 命令実行できない場合を示す。

- **Cache miss**　A のアドレスが xxxx111 で B のアドレスが xxxx000 の場合
- **Control hazard**　A の命令が J 形式の場合
- **Destination conflict**　A と B が同じレジスタに値を書き込む場合と B の命令が Load Immediate の場合(ldhi,ldli)
- **Storage conflict**　A と B が外部メモリに同時にアクセスする場合（ld,pop,st,push)

３．最適化コンパイラ

3.1　概要と設計手順

設計するアーキテクチャ可変な最適化コンパイラは C 言語のサブセットである CMONI から 16 ビット MONI 命令セットへ変換するコンパイラである。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対象とし、各々のアーキテクチャで最適化コンパイラの設計を行った。コンパイラの構成を述べる。まず、ソースプログラムは、

① **字句解析**
② **構文解析**
③ **意味解析**
④ **最適化**
⑤ **コード生成**

の順に処理されて、最終的にオブジェクトプログラム(MONI 命令セット)が得られる。以下にコンパイラのブロックダイアグラムと各々の処理内容を示す。

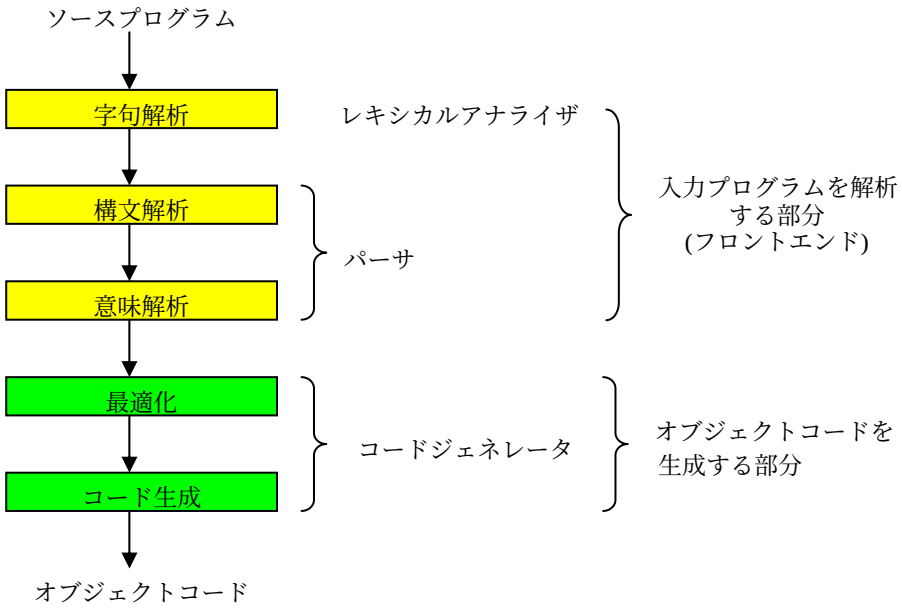

図８：コンパイラのブロックダイアグラム

① **字句解析**　字句解析はコンパイラの入り口である。ソースプログラムは連続した文字の並びとして表現されているが、これをトークンと呼ばれる単位に分離するのが字句解析の役目である。構文解析部は lex や flex を使用して自動生成することができる。

② **構文解析**　構文解析は字句解析からトークンの列を受け取り、言語の文法に従っ

ているかどうかチェックする。文法に正しく従っていれば、その結果を表現する構文木を作り上げる。構文解析部は yacc や bison を使用して自動生成することができる。yacc や bison では、文法規則と、その規則が認識されたときに実行する処理を対にして記述する。

③ **意味解析**　意味解析はプログラムを意味的な側面から解析する。意味解析ルーチンは、構文解析が構造(式、if 文、関数など)を認識したときに起動する。例えば、式が認識されると、式の意味解析用のルーチンが起動する。宣言が認識されると、その宣言の意味解析を行い、宣言されている変数をシンボルテーブルに登録する。意味解析は、出力としてソースプログラムの中間表現を作り出し、コードジェネレータに渡す。

④⑤ **最適化とコード生成**　最適化部とコード生成部には密接な関係がある。意味解析部から渡された中間表現をもとに最適化を行い、最終的にオブジェクトコードを生成する。最適化の目的は”実行速度を上げる””プログラムの大きさを小さくする”の二つがある。最適化には、マシンに依存しない最適化と、マシンに依存した最適化があり、前者はプログラムの意味を変えないでソースプログラムを変形する最適化である。それに対して後者はマシンの特性を活かして行うものでパイプラインが効率よく動くように命令を配置するなどマシンのアーキテクチャに密着した最適化である。最適化部では内部表現に変換したソースプログラムに対して、まずマシンに独立な最適化を行い、次にマシン依存の最適化を行ってオブジェクトコードを生成する。

次に本研究で設計したコンパイラの設計手順について述べる。

① 文法の定義をし、構文解析部を yacc を用いて作成する
② 字句解析部を作成する
③ 宣言と式を表現する木構造を扱うルーチン、シンボルテーブル処理ルーチン、オブジェクトコード生成ルーチンの作成
④ 外部宣言、関数定義、パラメータ宣言、ローカル宣言などの宣言と定義を処理する部分を作成する
⑤ 意味解析部を作成する
⑥ マシンに依存しない最適化部の作成
⑦ マシンに依存する最適化部の作成
⑧ 式のコード生成を作成する
⑨ 文の処理部(文のコード生成)を作成する

以上の順を追ってコンパイラの設計を行った。以下の表に作成ファイル一覧と内容を示す。

表１：作成ファイル一覧

ファイル名	内容
main.c	コンパイラのメインルーチン
declare.c	宣言と定義の処理（外部宣言、関数定義、パラメータ宣言、ローカル宣言などすべての宣言と定義の処理）
exp.c	式の意味解析ルーチン
gencode.c	オブジェクトコード生成ルーチン
genexp.c	式のコード生成
heap.c	ヒープ領域ハンドラ（宣言と式を表現する木構造を扱うルーチン）
lex.c	字句解析部(本体)
lexstr.c	字句解析部の下請けルーチン(文字列の処理)
misc.c	下請けルーチン群(エラー処理、型あわせ、型チェック、マシンに依存しない最適化)
stmt.c	文のコード生成
table.c	シンボルテーブル処理ルーチン
cmoni.h	トークンのマクロ定義
cmonidef.h	型、マクロの定義
function.h	関数のプロトタイプ
cmoni.y	CMONI 言語の文法定義

3.2 対象とする言語

教育用・教材としてふさわしいようにコンパクト且つ教材として役立つことを主眼に置いた C 言語のサブセット（CMONI）を考えた。それは、ハード/ソフト・コラーニングシステムの一部として考えた場合、ソースプログラムがどのように MONI コードに変換されたのかなど、分かりやすくする必要があると考え、コンパイラからも各アーキテクチャの理解を促したいからである。以下に C サブセットの言語仕様を示す。

表２：C 言語との相違点

	サポートする	サポートしない
型	ポインタ、配列	構造体、共用体
イニシャライザ	×	○
データ型	int(16bit) char(8bit)	short ,long,少数,列挙 signed,unsigned
記憶クラス	auto,extern	register,static, typedef

また、他にいくつかの制約を設けている。以下に示す。

- 関数の定義方法はプロトタイプ風に行う
- 複合文の先頭ではローカル変数を宣言できない。

である。１つめの項目は、関数はプロトタイプ風で定義するだけで構文がプロトタイプになっているに過ぎず、関数呼び出し時に、パラメータの型チェックはしない。

２つめの項目は、ローカル変数をネストとして宣言することができないという意味である。例えば C 言語では

```
main()
{
 int a,b;       ----------(a)
 if(a==10){
    int a, x;   ----------(b)
 }
}
```

という具合に、複合文の内側でもローカル変数を宣言（上記の例では(a)、(b)のどちらでも）することができるが、CMONI 言語では、複合文の内側では変数を宣言することができないので、上記の例において CMONI 言語がローカル変数を宣言できる場所は(a)のみとなる。以下に予約語一覧を示す。

break	case	char
continue	default	do
else	extern	for
goto	if	int
return	sizeof	switch
while		

図９：予約語一覧

４．フロントエンド部の設計

4.1　字句解析

字句解析は、文字の列としての原始プログラムを字句の列に区切り、字句を内部表現に変換する処理をする。本研究で設計した字句解析部は、字句解析本体とその下請けルーチンである文字列を扱う部分で構成されている。

4.1.1　字句解析本体の構成

字句解析本体の処理手順は

① 空白文字(スペース、タブ、改行)を読み飛ばす

② 現れた文字に応じた処理をする

③ 構文解析に送る

のようになり以下に字句解析本体を構成する主要な関数とその内容を示す。

・関数 yylex　　字句解析部の本体というべき部分であり、まず空白文字(スペース､タブ､改行)を読み飛ばし､次に現れた文字に応じた処理を行う。英字か下線ならば関数 lexId､数字ならば関数 lexNumber､文字列ならば関数 lexString を呼び出す。また演算子の場合は演算子のタイプをリターンするとともに、必要に応じてトークンの値をセットする。

・関数 nexteq　　入力から１文字先読みして、次にくる文字が'='であるかどうか調べる。もし違うなら関数 backch を呼び出して先読みした文字をもとにもどします。これは+=などの代入演算子をチェックするのに使用する。

・関数 lexId　　識別子を配列に読み込み、それがキーワードになっているかどうかを調べる。

・関数 lexNumber　数値定数を読み込み、その値をセットし、トークンを返します。８進数、１０進数、１６進数を処理することができる。

・関数 backch　　関数 nexteq によって先読みした文字を元に戻す。

・関数 skipcmt　　コメント部分を飛ばす

ここで、関数 yylex の処理のひとつである演算子の場合の演算子とリターン値、値のセットの関係の一覧表を示す。

表３：演算子とリターン値、値のセットの関係

演算子	yylex のリターン値	値
+	ａｄｄｏｐ	‘+’
－	ａｄｄｏｐ	‘—’
>>	ｓｈｉｆｔｏｐ	‘>’
<<	ｓｈｉｆｔｏｐ	‘<’
%	ｍｕｌｏｐ	‘%’
/	ｍｕｌｏｐ	‘/’
<=	ｃｏｍｐｏｐ	‘(‘
<	ｃｏｍｐｏｐ	‘<’
>=	ｃｏｍｐｏｐ	‘)’
>	ｃｏｍｐｏｐ	‘>’
!=	ｅｑｏｐ	‘!’
==	ｅｑｏｐ	‘=’
=	ａｓｓｉｇｎｏｐ	‘=’
%=	ａｓｓｉｇｎｏｐ	‘%’
&=	ａｓｓｉｇｎｏｐ	‘&’
*=	ａｓｓｉｇｎｏｐ	‘*’
+=	ａｓｓｉｇｎｏｐ	‘+’
－=	ａｓｓｉｇｎｏｐ	‘—’
/=	ａｓｓｉｇｎｏｐ	‘/’
<<=	ａｓｓｉｇｎｏｐ	‘<’
>>=	ａｓｓｉｇｎｏｐ	‘>’
^=	ａｓｓｉｇｎｏｐ	‘^’
|=	ａｓｓｉｇｎｏｐ	‘|’
++	ｉｎｃｄｅｃｏｐ	‘+’
－－	ｉｎｃｄｅｃｏｐ	‘—’
!	ｕｎｏｐ	‘!’
~	ｕｎｏｐ	‘~’
||	ｌｏｇｏｒ	‘|’
&&	ｌｏｇａｎｄ	‘&’

4.1.2　文字列の処理部

基本的には文字列も定数の一種に過ぎないので、文字定数や数値定数と同じように処理すればいいのだが、文字列は可変長であり、パーサが持っているスタックには可変長のデータを積むことができない。そこで、可変長のデータを直接扱わずポインタを介し

て実現する。

構造体 STROBJ が、文字列の実体で３つのメンバを持っている。メンバ strId は、文字列の識別番号であり、メンバ string は文字列本体へのポインタを、メンバ length は文字列の終わりを示すヌル文字を含んだ文字列長を持っている。文字列を表現するためのデータ構造は図１のようになっている。文字列の実体は STROBJ 型の配列 strobj に置き、この配列をポインタ objptr で管理する。また、文字列の具体的な値は char 型の配列 strpool に格納する。また、１つの関数内で利用できる文字列の個数を５０個、総文字数は１５００バイトに制限を設けた。

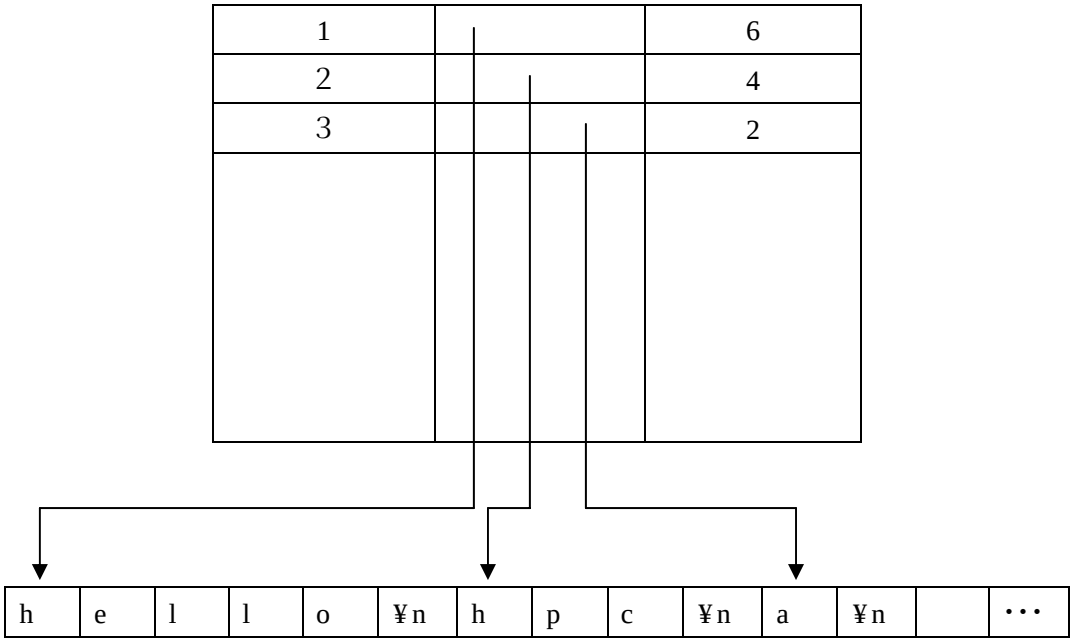

図１０：文字列を表現するデータ構造

以下に作成した関数とその役割を示す。

・関数 lexString　　まず初めに関数 readString を呼び出します。関数 readString は下請けとして関数 chval,tonumber,hassinn を呼び出す。また、関数」readString は文字を配列に格納するのに、関数 appendch を使用する。その後、関数 allocstrobj を呼び出し、strId、string、length の各メンバにデータをセットした後に文字列の識別番号をセットする。

・関数 readString　文字列を配列に読み込んでその先頭へのポインタを返す。

・関数 chval　　１文字分を読み込んでその値を返す。

・関数 tonumber　　ひとつの文字を数字とみなして、その数字としての値を返す。

・関数 hassinn　　文字が８進数になっているかチェックする。

・関数 appendch　　配列に格納する。もし配列が一杯であればエラーを返しコンパイルを中断させる。

・関数 allocstrobj　　配列内に文字列の実体を割り当ててデータをセットする。

4.2　構文解析

構文解析部は字句解析部からトークンの列を受け取り、言語の文法に従っているかチェックし、従っていればその結果を表現する構文木を作り上げていく。構文解析部を自動生成するツールとして yacc,bison がある。ＣＭＯＮＩコンパイラでは yacc を使用し構文解析部を生成する。yacc では文法規則と、その規則が認識されたときに実行する処理を対にして記述するものである。

- 構文解析の構成

構文解析部の主となるプログラム部分は

① セマンスティックスタック定義
② トークン定義
③ 優先順位の定義
④ 文法の定義

の４つの部分に分けられる。

① セマンスティックスタック定義

yacc ではアクション間で情報のやり取りを行うのに、セマンスティックスタックを利用する。そのため、あらかじめスタックの定義とデータ型の定義が必要である。以下に各メンバの役割を示す。

表４：セマンスティックスタックの内容

メンバ	内容
CONST _const;	定数の値
TREE *tree;	宣言の木へのポインタ
TYPE *type;	型テーブルへのポインタ
char *symbol	識別子
EXPR *expr	式の構文木へのポインタ
char op	演算子のトークンの値
int label	制御構造のラベル

② トークンの定義

トークンには識別子、定数(数字、文字定数、文字列)、予約語、区切り記号、演算子に分類され、それぞれ対応するトークンに変換される。本研究では yacc のトークンの宣言をする%token という機能を使用し、トークンの宣言を行った。以下にそのプログラムを示す。

```
%token <symbol> IDENTIFIER
%token <_const> CONSTANT
%token          IF ELSE WHILE FOR DO BREAK CONTINUE SWITCH CASE
%token          DEFAULT RETURN GOTO SIZEOF INT EXTERN
%token          ';' '(' ')' '[' ']' '{' '}' ','
%token          '?' ':' '|' '^' '&' '*' '.' '='
%token <op>     addop shiftop mulop comop eqop assignop
%token <op>     incdecop unop
%token          logor logand
```

図１１：トークン宣言部のプログラム

IDENTIFIER は識別子、CONSTANT は定数を宣言。その下のＩＦ～ＥＸＴＥＲＮまで予約語、';'～'='までは区切り記号の宣言、最後の 3 行は演算子の宣言し、表 3 に対応した値をセットする。

③ 優先順位の定義

優先順位の定義は yacc の%left,%right,%nonassoc という機能を使用して宣言する。%left は左結合を示し、%right は右結合を示し、代入演算子、条件演算子、単項演算子に使用する。また、結合することができないものは%nonassoc という無結合を示す機能を使用し、宣言する。ＣＭＯＮＩ言語はＣ言語と同様に 15 段階の優先順位が存在し、優先順位の低いものから高いものへと順に宣言していく。以下にそのプログラムを示す。

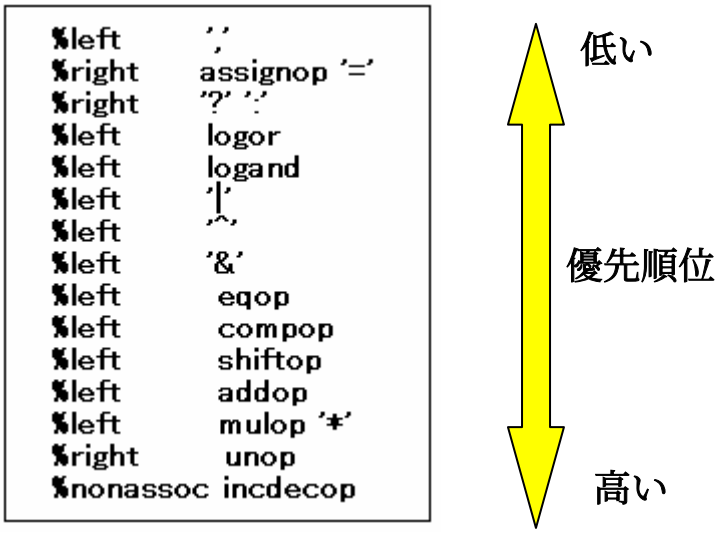

図１２：優先順位宣言部のプログラム

④　文法の定義

式、文、宣言、プログラムの４つのレベルから構成されている。式の定義には演算子を使った式の文法定義とアクションを記述。文の定義には IF 文や WHILE 文などの文の文法定義とアクションを記述、宣言の定義にはポインタや配列[]などの宣言を定義しアクションを記述、プログラムの定義には関数や変数の定義とアクションを記述した。

（１）式の定義

式の定義では nc_expr,expr,primary,expr_list の４つのノンターミナル(文法規則の左辺)が定義されている。実際に式を表しメインとなるのが expr であり、他の３つのノンターミナルは expr を定義するための補助的なものである。nc_expr ではコンマ以外の演算子を含む式を定義。優先順位はすでに定義されているので、演算子の使用方法と実際のアクションを記述。expr では nc_expr を使用しコンマ式を定義。演算子の使用方法と実際のアクションを記述。primary では変数、定数、配列参照、関数呼び出し、括弧による式のくくりだしを定義。実際のアクションを記述。expr_list では関数呼び出し時のパラメータリストを記述するためのものを定義し、実際のアクションを記述。

（２）文の定義

文の定義ではノンターミナル stmt で定義されている。複合文、式文(式＋セミ　　コロン)、if 文、while 文、do-while 文、for 文,switch 文、case 文、continue 文、return 文、goto 文、ラベルつき文、空文をそれぞれ定義しアクションを

記述していく。if 文には else のあるなしの 2 種類がある。また、エラーリカバリーを行いやすいように if 文、while 文、for 文,switch 文には「if()」の様に頭部を表すノンターミナル(if_head,while_head,for_head,switch_head)を独立に定義している。さらに、for 文ではカッコ内に式が３つ書けるが、式を省略できるようにするノンターミナル(expr_opt)を定義している。

（３）宣言の定義

宣言の定義は、declaration、decelerator、type_name の３つの主要なノンターミナルと type_spec、declarator_list、decelerator2、param_list、pram_decl、abst_decelarator、abst_decelarator2 の補助的なノンターミナルから構成されている。declaration は宣言のことで、型指定子(type_spec)の後に宣言子(decelerator、decelerator2)を０個以上コンマに区切って並べ、最後にセミコロンをおいたものである。また、decelerator2 では関数呼び出しの宣言、パラメータつきの関数宣言をし、パラメータ自体は param_list、pram_decl によって定義している。type_name は型名を抽象宣言子(abst_decelarator、abst_decelarator2)によっての定義をし、キャストや sizeof を定義するのに使用する。

（４）プログラムの定義

プログラムの定義は、ノンターミナル prog をメインとし、extern_def、func_def、func_body の補助的なターミナルから構成されている。prog は関数の定義、変数の定義を宣言している extern_def を１つ以上並べたものである。func_def では、関数が型指定子＋宣言子＋関数本体か宣言子＋関数本体という形であるかチェックする。func_body は関数本体の先頭でローカル変数を定義するものである。

4.3 意味解析

構文解析の次に行われるのが意味解析である。構文解析はプログラム言語の構文規則と原始プログラムとの対応をとる解析であるが、意味解析は意味規則との対応をとる解析である。例えば、変数名を実数型と宣言したらその変数は実数型としてしか使用できないというものである。意味解析ではそのような宣言情報を集めた記号表が重要な役割を持っている。

4.3.1 ヒープ領域ハンドラ

本コンパイラでは宣言や式を直接処理せず、いったん木構造を組み立て、その木構造をもとに意味解析処理を行っている。この木構造が格納される場所がこのヒープ領域である。

ヒープ領域には

①宣言を表す木構造

②式を表す木構造

③識別氏の名前を表す文字列

の３種類のデータが置かれる。ヒープ領域は２０００バイトを用意した。ヒープ領域はポインタで管理され、データ格納用の関数が呼び出されるたびに、アドレスの低い方から高い方へと順に領域を割り当てていく。以下にヒープ領域を操作する関数を示す。

表５：ヒープ領域を操作する関数

目的	関数
宣言を表す木構造を作る	list1 list2 list3
式を表す木構造を作る	makenode1 makenode2 makenode3
文字列を格納する	savechar
ヒープ領域をクリアする	resetheap

宣言の木構造を作る関数は、lsit1 list2 list3 の３つがある。それぞれ、子がないノード、子が１つのノード、子が２つのノードをつくり、作り出したノードへのポインタを返す関数である。

式の木構造を組み立てる関数は、makenode1 makenode2 mekenode3 の３つがあり、それぞれ、単項、２項、３項演算子のノードを作る。これらの関数は、ノードの演算の種類、演算を行う型、演算結果の型、子へのポインタをパラメータとして受け取り、組み立てたノードへのポインタを返す関数である。

識別子の名前をヒープ領域に退避する関数 savechar は、字句処理部で処理された識別子を字句処理部では識別子が読み込まれるたびに上書きされるので、識別子の名前を失わないように格納する関数である。

ヒープ領域をクリアする関数 resetheap は、ヒープ領域の全体を一気にクリアするもので、構文解析部が宣言や文をひとつ処理子し終わったタイミングで呼び出され実行される関数である。

4.3.2　シンボルテーブル処理ルーチン

シンボルテーブルは

①シンボルテーブル本体

②型テーブル

③識別子の名前を入れる領域

の３つの部分から構成されている。また、シンボルテーブルには同じ構成のものをグローバル宣言用、ローカル宣言用の二つを作成した。グローバル宣言用のシンボルテーブ

ルは、最初に初期化すれば後はそのままであるが、ローカル宣言用のシンボルテーブルは、新しい関数の処理を開始するたびにクリアされる。以下にシンボルテーブルを操作／探索する関数をしめす。

表６：シンボルテーブルを操作／探索する関数

目的	関数	
	グローバル用	ローカル用
シンボルテーブルを初期化する	initGloTbl	initLocTbl
文字列を格納する	saveGloId	saveLocId
シンボルテーブルに登録する	regGloId	regLocId
シンボルテーブルをサーチする	searchGlo	searchLoc
型を型テーブルに登録する	regGloType	regLocType
ローカル変数のオフセットの計算をする		computeOffset

シンボルテーブル本体は構造体で実現しており、４つのメンバをもっている。変数／関数の名前を表す文字列へのポインタ、グローバル宣言での文字列を格納する配列、ローカル宣言での文字列を格納する配列、変数／関数のクラスを表すものである。最後の変数／関数のクラスを表すものには、グローバル宣言された変数／関数のうち、そのファイルで定義されているもの、グローバル宣言された変数／関数のうち、別のファイルで定義されているもの、ローカル変数を表すもの、関数のパラメータを表すものの４通りの方法で表す。

型テーブルはシンボルテーブルから呼び出され、シンボルテーブルと対になって、型情報を管理するテーブルである。

4.3.3　宣言と定義の処理部

宣言を認識すると、その宣言の意味解析を行い、宣言されている変数／関数をシンボルテーブルに登録するという操作を行う部分である。以下に宣言と定義の処理する関数を示す。

表７：宣言と定義の処理する関数

目的	関数
グローバル変数の宣言	glodecl
グローバル変数の定義	gloDatadecl
パラメータの宣言	paramdecl
関数の定義	funcDef
関数の開始時コード出力	funchead
関数の終了時コード出力	funcend
ローカル変数の宣言	locdecl
ローカル変数の定義	locDatadecl
抽象宣言子の処理	abstdecl

関数 glodecl paramdecl locdecl abstdecl の４つの関数は、木構造をたどりながら、その宣言が表す型を型テーブルの中に構築していく関数である。関数 locdecl を例に説明する。

① for 文でループしながら木構造をたどっていく

② ノードに応じて必要な処理に分岐する

③ あらかじめローカルシンボルテーブルをサーチして、すでに定義済みであるかチェックする

④ ③でない場合、結果をローカルシンボルテーブルに登録して、ポインタを返す

⑤ ③である場合、抜ける

⑥ 宣言で禁止されている組み合わせを使用していないかチェックする

（例　関数の配列、配列を返す関数、関数を返す関数など）

という処理（意味解析）を行いシンボルテーブルに順に登録していく。表３の残りの関数はすべて構文解析部から起動されている。主に参照を行っている。

4.3.4　.式の意味解析ルーチン

構文解析の中の式のアクションから呼び出され、型情報の保持と型変換のための木構造を作り上げる機関である。

式の木構造は構造体を使用して表現しており、opcode,optype,misc,type の４つのメンバから成り立っている。メンバ opcode は、そのノードがどのような演算を(操作)を表しているかを示す。opcode の意味を以下に示す。

表８：opcode の意味

値	意味
op_ADR	変数のアドレス
op_INDIR	インディレクション
op_CONST	定数
op_STR	文字列
op_ASSIGN	代入（＝）
op_ASSIGN_OP	代入演算子（＋＝、－＝）
op_ASSIGN_OP_R	ポインタと数値の代入演算子(＋＝、－＝)
op_PLUS	単項の＋
op_MINUS	単項のー
op_BNOT	ビット演算の NOT
op_LNOT	論理演算の NOT
op_PREINC	前置の＋＋
op_PREDEC	前置のーー
op_POSTING	後置の＋＋
op_POSTDEC	後置のーー
op_ADD	数値どうしの加算
op_SUB	数値どうしの減算
op_SCALEADD	ポインタと数値の加算
op_SCALESUB	ポインタと数値の減算
op_DESCALESUB	ポインタどうしの減算
op_MOD	乗除
op_DIV	除算
op_MUL	乗算
op_SHR	右シフト
op_SHL	左シフト
op_GT	比較（＞）
op_GE	比較（＞＝）
op_LT	比較（＜）
op_LE	比較（＜＝）
op_EQ	比較（＝＝）
op_NEQ	比較（！＝）
op_BOOL	正規化した比較演算
op_AND	ビット演算の AND
op_XOR	ビット演算の XOR

op_OR	ビット演算の OR
op_LAND	論理演算の AND
op_LOR	論理演算の OR
op_COMMA	コンマ演算
op_COND	条件演算子
op_FUNC	関数呼び出し
op_ItoC	型変換 int→char
op_CtoI	型変換 char→int
op_RtoI	型変換ポインタ→int
op_RtoC	型変換ポインタ→char
op_CtoR	型変換 char→ポインタ
op_ItoR	型変換 int→ポインタ

メンバ optype は、そのノードの演算がどんな型で実行されるかを示すものであり、以下の表 2 にメンバ optype の意味を示す。

表９：メンバ optype の意味

値	意味
C	char 型である
I	int 型である
R	ポインタ型である
B	bool 型である
K	定数形である

※bool 型は､比較演算の結果を効率よく処理するために導入した仮想的な型である。

メンバ misc は、補助的な情報をセットアップするフィールドであり、式の意味解析ルーチンの下請けルーチンの役割を果たし、型あわせ、型チェック、定数式の畳み込みなどを行った結果を示すものである。メンバ type は、そのノードの型で、型テーブルへのポインタをセットする。

式の木構造のノードを作る関数は makeNode1,makeNode2,makeNode3 の３つがあり､それぞれ単項、2 項、3 項のノードを作成する。これらの関数は、ノードの演算の種類、演算を行う型、演算結果の型、子へのポインタをパラメータとして受け取り、組み立てたノードへのポインタを返すものである。

木構造を実際組み立てる式の意味解析ルーチンは

① 演算子の種類に応じたチェック・型あわせをおこなう

② 木構造を組み立てる

③木構造へのポインタを返す

という一連の処理を opcode 毎に行う。

５．バックエンド部の設計

5.1　最適化部の設計

最適化とは、効率のよい目的プログラムにすることである。最適化には、目的プログラムの実行効率をよくする。すなわち、実行速度を上げるものと、目的プログラムの大きさを小さくするものがある。前者はプログラムの一部分、例えばループ部分などプログラム全体の実行時間の大部分を占めるので、その部分を最適化すれば大きな効果が得られる。後者は記憶容量の小さい計算機の場合に重要であるが、広域的な最適化が必要である。本コンパイラでは教育用としてアーキテクチャを意識したものである必要があるため前者の局所的最適化を行った。対象とするプロセッサアーキテクチャは２章で示したＭＯＮＩ単一サイクルプロセッサ・ＭＯＮＩマルチサイクルプロセッサ・ＭＯＮＩパイプラインプロセッサ・ＭＯＮＩスーパースカラプロセッサの４種類である。

5.1.1　単一/マルチサイクルプロセッサの最適化

単一/マルチサイクルプロセッサの最適化は、図３で示したようにプロセッサに突出した最適化は行わず、マシンに依存しない最適化のみを行う。命令の実行回数を減らすことを目標とし、教育を主眼に置き最適化を行った。

命令の実行回数を減らすには

① １度実行した結果を再利用する

② 冗長な命令を取り除く

③ コンパイル時に実行できるものは実行してしまう

ことなどが挙げられる。①を実現するために共通部分式の削除を行い。②を実現するため無用命令の削除を行い。③を実現するために定数の畳み込みを行った。

（１）共通部分式の削除

共通部分式を何回も計算しないようにするのが共通部分式の削除である。例えば

```
A=A+B*C;
D=A+C*B;
C=B*C+A;
```

というプログラムがあるとすると、まずこれらの式を分解し

```
R1＝B*C;
R2=R1＋A;
A＝R2；
R4＝B*C;
R5＝R4＋A;
D=R5;
R7=B*C;
R8＝R7+A;
C=R8；
```

となり、右辺の変数がどこで定義されたかを格納しておく表を作成し、A、B、C に番号を振る。

R1＝B0＊C0;
R2=R1＋A0;
A３＝R2;
R４＝R1;
R5＝R1＋A３;
D６＝R5;
R7＝R1;
R8=R5;
C９＝R5;

となり、結局有効な命令は

R1＝B＊C;
R2=R1＋A;
A＝R2;
R5=R1＋A;
D＝R5;
C=R5;

となる。全体の処理の流れは

① 分岐命令・分岐先命令を含まない基本ブロックを抽出する。
② 式を分解する。
③ 基本ブロック外で定義されている変数には０をつけそれ以外には行番号の値をつける
④ 共通部分式を削除する。

である。

（２）無用命令の削除

無用命令の削除は例えばｘ＝ｘや、ｘ＝ｙ－ｚという式があってもｘを使わない場合など、実行しても意味が無い命令を削除するものである。そのためには変数の持つ値に番号をつけ、同じ値のものには同じ番号を付け削除していく。作成した処理の流れは、

① 分岐命令・分岐先命令を含まない基本ブロックを抽出する。
② 式を分解する
③ 変数の持つ値に番号をつけ、同じ値のものには同じ番号を付ける。
④ 無用命令を削除する。

（３）定数の畳み込み

定数の畳み込みとはコンパイル時に実行できるものは実行してしまおうというものであり、例えば

```
A=3+7;
  …
B=A+10;
```

とあった場合にA=１０、B=２０と置き換えることである。作成した処理の流れは

① 分岐命令・分岐先命令を含まない基本ブロックを抽出する。
② 式を分解する。
③ 後の式で使う値が必ずそれ以前に計算されているか確認する。
④ 以前の計算をしてから後の計算をするまで値が変わっていないか確認する。
⑤ ③④がOKなら値を置き換える。

である。

以下に関数一覧を示す。

表10：マシンに依存しない最適化部の関数

目的	関数
制御フローグラフ作成	flowgraph
データの流れ解析	dataflow
基本ブロック分割	baseblock
共通部分式の削除	comsubelim
無用命令の削除	deadcodeelim
定数の畳み込み	fold

5.1.2 パイプラインプロセッサの最適化

パイプラインプロセッサの最適化は 5.1.1 で示したマシンに依存しない最適化を行い、コード生成を行った後にパイプラインプロセッサ特有の最適化を行う。この最適化手法はピープホール最適化と呼ばれている。なぜこの最適化手法を用いたかは６章で述べる。

本研究で対象としているパイプラインプロセッサは 2.3.3 で述べたプロセッサである。パイプラインプロセッサにはいつでも次のクロックサイクルで次の命令が実行できないパイプラインハザードが起こる。パイプラインハザードには構造ハザード・データハザード・制御ハザードの３つがある。ＭＯＮＩパイプラインプロセッサで起こるパイプラインハザードはデータハザードと制御ハザードの２つである。データハザードは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場合、制御ハザードは分岐が成立した場合に生じる。そのため、次命令にＮＯＰ命令を挿入(ストール)し、プロセッサを

正常に動作させる。以下に例を示す。

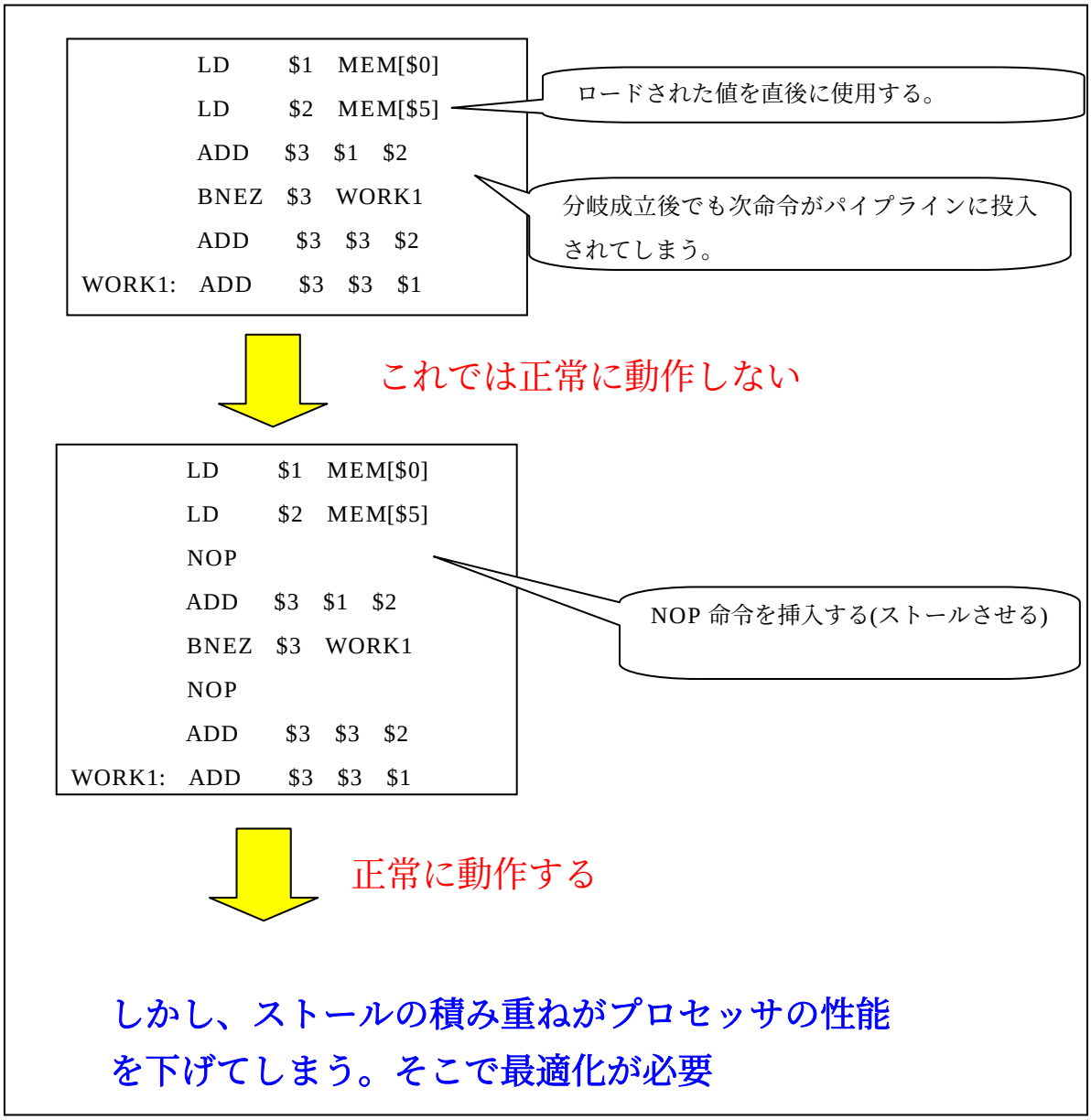

図１３：ハザードの構造と最適化の必要性

図のようにＮＯＰ命令を挿入(ストール)すれば、プロセッサは正常動作するが、ループ内など複数回ストールが起これば、そうとうの性能低下を引き起こされてしまう。そこで、命令スケジューリングを考えストール(ＮＯＰ命令の挿入)回数を減らすことを目標に最適化を行う。

（１） 目標

① レジスタ値の再利用による冗長なロード命令の削除

② データハザード回避のための命令の並べ替え

③ 分岐命令の後には分岐命令に依存しない命令を配置

以下に最適化の具体例を示す。

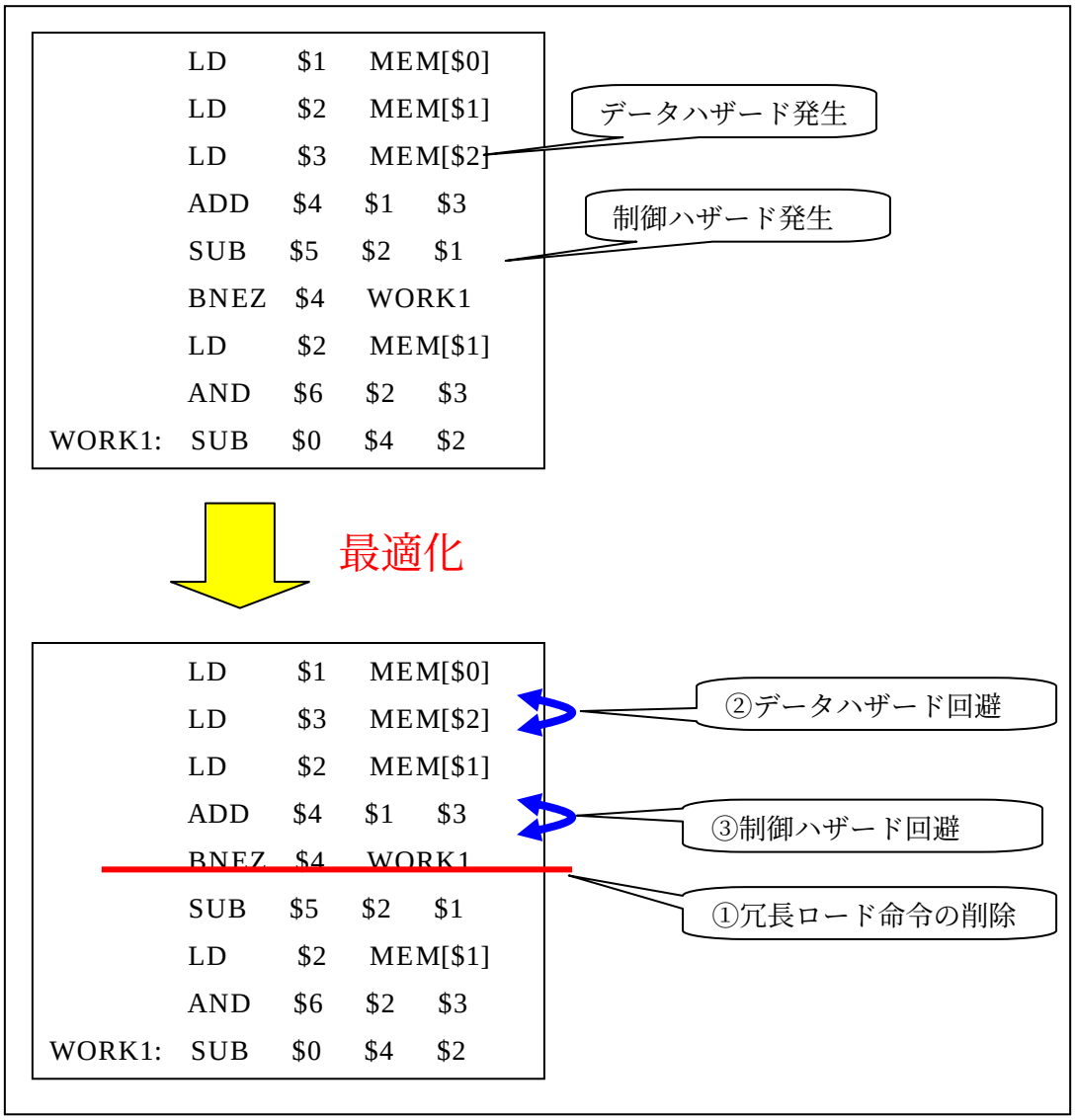

図１４：パイプラインプロセッサ最適化例

（２） 処理手順

① 分岐命令・分岐先命令を含まない基本ブロックを抽出する。
② レジスタ値の書き換えが無いかチェックする。
③ レジスタ値が変化しない間にあるロード命令を削除する。
④ ロード命令の直後に同じレジスタを使用していないかチェックする。
⑤ 命令の入れ替えを行う。不可能ならＮＯＰ命令を挿入する。
⑥ プログラム全体で分岐命令の前の命令と入れ替え可能かチェックする。
⑦ 命令の入れ替えを行う。不可能ならＮＯＰ命令を挿入する。

5.1.3　スーパースカラプロセッサの最適化手法

スーパースカラプロセッサの最適化は 5.1.1 のマシンに依存しない最適化、コード生成、5.1.2 のパイプラインプロセッサの最適化を行った後にスーパースカラプロセッサ特有の最適化を行う。

本研究で対象としているスーパースカラプロセッサは 2.3.4 で述べたプロセッサである。スーパースカラプロセッサは 2 命令を同時平行的に実行することで高速化を図るものであるが、常に 2 命令同時並行的に実行することができない。以下に 2 命令同時並行的に実行できない場合と具体例を示す。

- **Cache miss**　A のアドレスが xxxx111 で B のアドレスが xxxx000 の場合
- **Control hazard**　A の命令が J 形式の場合
- **Destination conflict**　A と B が同じレジスタに値を書き込む場合と B の命令が Load Immediate の場合(ldhi,ldli)
- **Storage conflict**　A と B が外部メモリに同時にアクセスする場合（ld,pop,st,push)

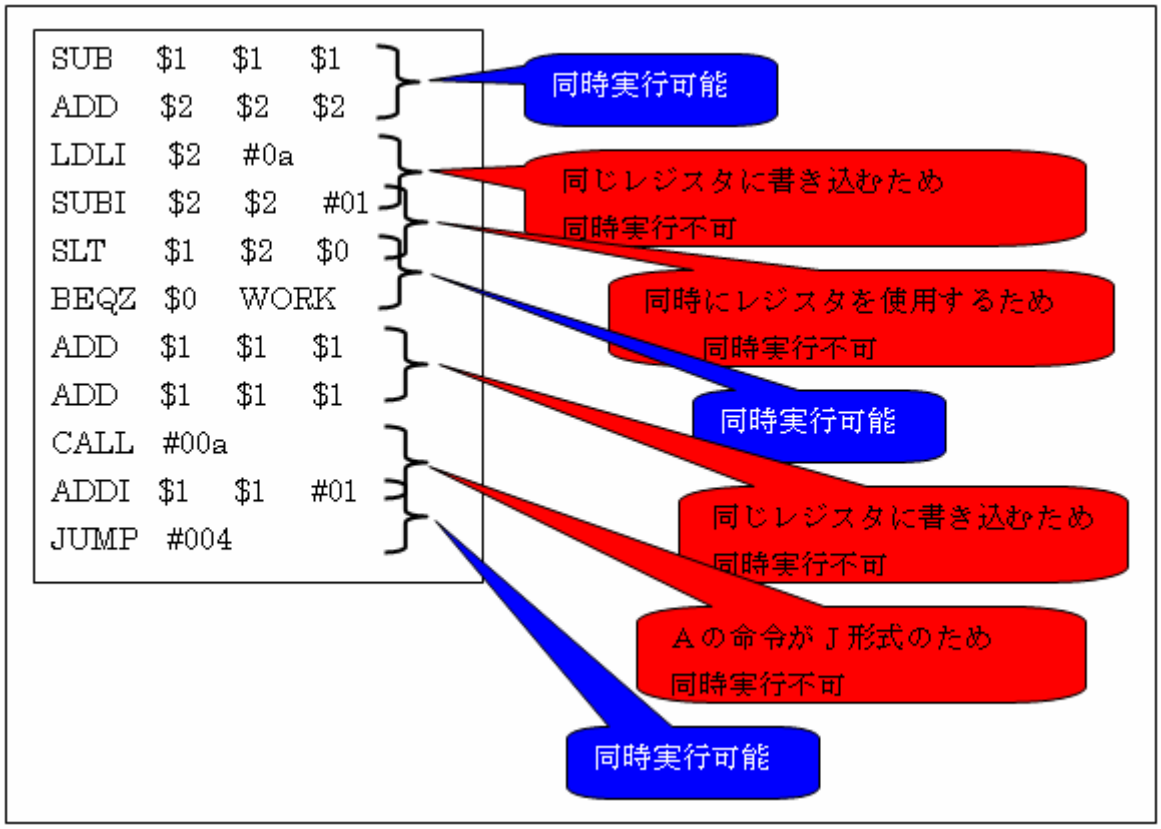

図１５：同時実行できない状態例

図 15 のようにいつでも 2 命令同時実行できるわけではない。対象とするスーパースカラ

プロセッサは連続する２命令の依存関係を調べ、同時実行できるか判断し、１命令実行か２命令実行を行うものである。そこで、命令スケジューリングを考えより多い回数２命令同時実行可能になるような最適化を目指す。

（１） 目標

① 同じレジスタに書き込む命令をできる限り続かないようにする。

② ＬＤ/ＳＴ/ＰＯＰ/ＰＵＳＨ命令をできる限り続かないようにする。

③ ユニットＡにＪ形式ができる限りこないようにする。

④ ユニットＢにＬＤＨＩ/ＬＤＬＩ命令ができる限りこないようにする。

以下に最適化の具体例を示す。

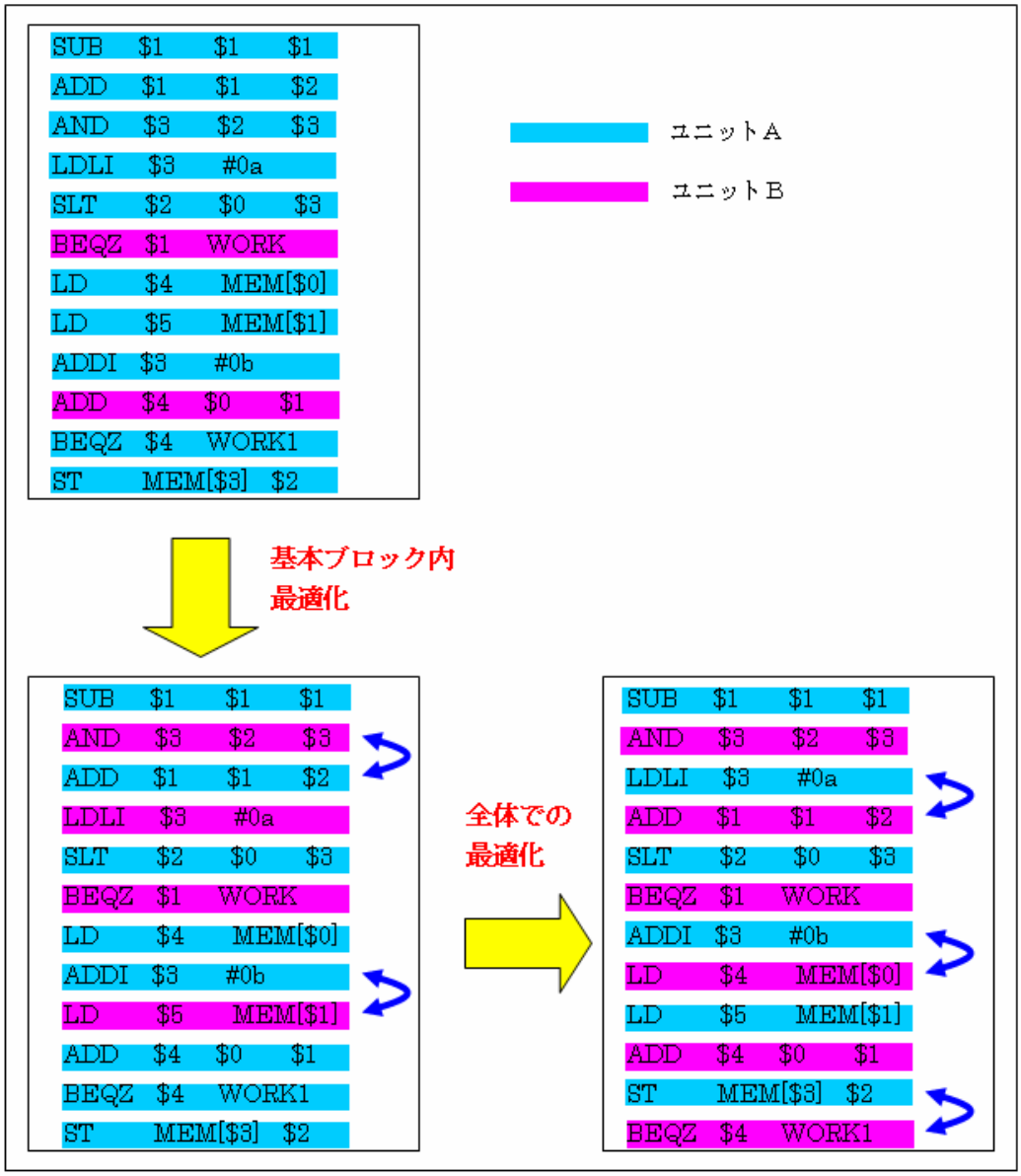

図１６：スーパースカラプロセッサ最適化例

（２） 処理手順

① パイプラインプロセッサの最適化を行う
② 分岐命令・分岐先命令を含まない基本ブロックに分割する。
③ 連続した命令で同じレジスタに書き込みが行われるかチェックする。
④ 命令の入れ替えを行う。不可能ならＮＯＰ命令を挿入する。
⑤ ＬＤ/ＳＴ/ＰＯＰ/ＰＵＳＨ命令が連続していないかチェックする。
⑥ 命令の入れ替えを行う。不可能ならＮＯＰ命令を挿入する。
⑦ 基本ブロックを復元する。
⑧ 分岐命令/ＬＤＨＩ/ＬＤＬＩ命令の行番号をチェックする。
⑨ 分岐命令なら偶数行へ、ＬＤＨＩ/ＬＤＬＩ命令なら奇数行へ入れ替えを行う。
不可能ならＮＯＰ命令を挿入する。

5.2 コード生成部の設計

5.2.1 式のコード生成

式のオブジェクトコードを生成するには、木構造を再帰的にトラバースすることで得ることができる。処理手順は

① 式の木にラベル付けを行う
② そのラベル付けされた木を辿って目的コードを生成する

（１）ラベル付けアルゴリズム

式の木を下から上に辿って （ラベルをＬ( )、Ｎはレジスタ数＝８とする）

① ノードＡが葉であれば、Ｌ(Ａ)とする
② ノードＡがＡ＝Ｂ演算Ｃの場合
(a) Ｌ(Ｂ)＝Ｌ(Ｃ)＜Ｎのとき
Ｌ(Ａ)＝Ｌ(Ｂ演算Ｃ)＝Ｌ(Ｂ)＋１＝Ｌ(Ｃ)＋１とする
(b) Ｌ(Ｂ)≠Ｌ(Ｃ)のとき
Ｌ(Ａ)＝Ｌ(Ｂ演算Ｃ)＝max(Ｌ(Ｂ)、Ｌ(Ｃ))とする
(c) Ｌ(Ｂ)＝Ｌ(Ｃ)＝のとき
Ｌ(Ａ)＝Ｌ(Ｂ演算Ｃ)＝Ｎとする

（２）コード生成アルゴリズム

① ノードＡが葉であれば、ＬＤ命令を生成し、親へ戻る
② ノードＡがＡ＝Ｂ演算Ｃの場合
(a) Ｂ，Ｃのどちらも辿ってないとき
ｱ) Ｌ(Ｂ)≠Ｌ(Ｃ)のとき、ラベルの値の大きいほうを辿る
ｲ) Ｌ(Ｂ)＝Ｌ(Ｃ)のとき、右の子供Ｃを辿る
ただし、Ｌ(Ｂ)＝Ｌ(Ｃ)＝Ｎのときは、その後、Ｃの値を退避するＳ

Ｔ命令を生成する

(b) Ｂ、Ｃのどちらか一方だけがまだ辿っていなければ、それを辿る

(c) Ｂ、Ｃとも辿ってあるとき、Ｂ演算Ｃの演算命令を生成し親に戻る
ただし、Ｌ(Ｂ)＝Ｌ(Ｃ)＝Ｎのときは、退避したＣの値のＬＤ命令を演算命令の前に付ける

以下に式コード生成を行う関数の名前と目的を示す。

表１１：式のコード生成を行う関数

目的	関数
ノードが変数であるかどうかチェック	isVariable
ノードが定数であるかどうかチェック	isConst
ラベルを生成する	genlabel
ラベルへのジャンプ命令を生成する	genjump
式文のコードを生成する	expstmt
代入のコード生成をする	genassign
代入演算子のコード生成をする	genassingnop
ポインタの代入演算子のコード生成をする	genassignopref
＋＋、－－演算子のコード生成をする(トップレベル)	genincdectop
式のコード生成をする(トップレベル)	genexptop
ビット演算子のコード生成をする	genbitop
シフト演算子のコード生成をする	genshigtop
条件ジャンプを生成する	gencondjump
数値比較のコード生成をする	gencompare
ブール演算のコード生成をする	genbool
ポインタと数値の加算/減算のコード生成をする	genscaled
ポインタ同士の減算のコード生成をする	gendescaled
関数呼び出しのコード生成をする	genfuncall
＋＋、－－演算子のコード生成をする	genincdec
式のコード生成をする	genexp

5.2.2 文のコード生成

文のコード生成は、yacc を使用して定義した文法定義の中のセマンティックアクションにおいて、文のコード生成を行う。そして、goto 文で使用するラベル管理、switch 文の case ラベルの管理などの補助するルーチンから構成する。

（１）式文

式文は、後ろにセミコロンを置いたものであり、式のセマンティックアクションでは、表１１の関数 expstmt を呼び出し、式のコード生成を関数 genexptop を呼び出して生成する。

（2）if 文・while 文・do while 文

if 文や while 文などは構造化文であるが、機械語ではサポートされていない。そこで、コンパイラがコード生成時に比較命令と条件分岐命令を使用して展開する必要がある。

・if 文には以下の①、②の２種類がある。

①**if（条件）文**　の場合

```
if（条件が偽）goto label_1
文
labal_1:
```

②**if（条件）文１ else 文２**　の場合

```
if（条件が偽）goto label_1
文１
goto label_2
label_1:
文２
label_2:
```

のように展開する。

・while 文は

wile（条件）文

の形式をしている。これを無条件分岐命令を使用して表現すると

```
loop:
Lc:（continue 文でジャンプするラベル）
   If（条件が偽）goto exit
   文
   goto loop
exit:
Lb:（break 文でジャンプするラベル）
```

と展開する。

・do while 文は

do 文 while（条件）

の形式をしている。これを無条件分岐命令を使用して表現すると

```
loop:
  文
Lc:（continue 文でジャンプするラベル）
  if（条件が偽）goto loop
```

Lb:（break 文でジャンプするラベル）

と展開する。

（３）break 文と continue 文

break 文と continue 文は、ループや switch 文が多重にネストされているときには、その時点で一番内側のループまたは、switch 文に対して作用する。それぞれグローバル変数を使用し、飛び先のラベル番号を常にセットしておく必要がある。以下に実現方法を述べる。

① ループや switch 文に入った時点でそれぞれのラベル番号をスタックに退避させる

② ループや switch 文内での飛び先のラベル番号をセットする

③ ループや switch 文を抜けるとスタックに退避していたラベル番号を復帰させる

④ それぞれの命令に遭遇すればそれぞれの飛び先へのジャンプ命令を生成する

（４）for 文

for 文では、for（初期化；条件；再初期化）のカッコ内の３つの式をいずれも省略できることに注意して作成を行った。以下に実現方法を述べる。

① for 文に必要なラベル３個(label,label+1,label+2)を発生させる

② ラベルをセマンティックスタックに積む

③ break 文、continue 文用のラベルをスタックに退避させる

④ break 文のラベルを label に、continue 文のラベルを label+2 にする

⑤ 初期化式の処理（初期化式がＮＵＬＬでないか確認後、関数 genexptop を呼び出す）

⑥ 再初期化式の処理（再初期化式がＮＵＬＬでないか確後、関数 genejump を呼び出す）

⑦ 条件判定の処理（条件式がＮＵＬＬでないか確認後、bool 型に変換し関数 genbool に渡す）

⑧ 真の場合は下に抜け、偽の場合は label+2 にジャンプする（関数 genejump を呼び出す）

⑨ 式のヒープ領域を開放する

⑩ 本体を認識（ループを繰返すジャンプ命令の生成、ループ出口のラベル発生）

⑪ break 文、continue 文のスタックに退避していたラベル番号を復帰させる

（5）switch 文

switch 文は、ジャンプテーブルをサーチして分岐するコードに展開する。ジャンプテーブルには、case ラベルの値と飛び先アドレスが対になって格納されている。以下に実現方法を述べる。

① case 文が現れるとオブジェクトコード中にラベルを立てる
② その case ラベルの値と出力したラベルの値を対にしてテーブルに保存する
③ ２重定義のチェックを行う
④ テーブルにある case ラベルと飛び先ラベルの組からジャンプテーブルを出力

（6）goto 文とラベル

goto 文のラベルは、専用のシンボルテーブルで管理する。以下に実現方法を述べる。

① 関数 regLabel によってラベルを登録する
② 関数 serchLabel によってラベルをサーチする
③ 関数 endFunc によって未定義のラベルの有無をチェックし、（未定義のラベルがあればエラーメッセージを表示する）処理を終了する

5.3 動作検証と評価

コンパイラのテストプログラムとして、Ｎまでの和(100)、バブルソート(50)、シェルソート(50)、ユーグリット互除法による最大公約数(152,36)、フィボナッチ数列(8)をＣＭＯＮＩ言語で記述後、本コンパイラを使用し、生成されたＭＯＮＩコードをＭＯＮＩシミュレータで使用し実行した。比較対象として直接ＭＯＮＩコードからプログラムを生成して実行したものを挙げる。実行結果を以下に示す。

表１２：実行結果

		単一サイクル		マルチサイクル		パイプライン	
		手動生成コード	コンパイラ生成コード	手動生成コード	コンパイラ生成コード	手動生成コード	コンパイラ生成コード
１からＮまでの和	静的命令数	12	16	12	16	12	16
	総実行命令数	507	639	507	639	507	639
	総クロック数	507	639	1925	3225	610	772
	ストール回数	-	-	-	-	99	125
バブルソート	静的命令数	28	34	28	34	28	34
	総実行命令数	17038	25325	17038	25325	17038	25325
	総クロック数	17038	25325	63081	91761	25289	34036
	ストール回数	-	-	-	-	8247	9531
シェルソート	静的命令数	57	73	57	73	57	73
	総実行命令数	4216	5974	4261	5974	4216	5974
	総クロック数	4216	5974	15390	19206	5002	6888
	ストール回数	-	-	-	-	782	938
ユーグリット互除法による最大公約数	静的命令数	16	21	16	21	16	21
	総実行命令数	58	84	58	84	58	84
	総クロック数	58	84	213	306	74	116
	ストール回数	-	-	-	-	12	27
フィボナッチ数列	静的命令数	76	94	76	94	76	94
	総実行命令数	553	825	553	825	553	825
	総クロック数	553	825	2031	2463	692	986
	ストール回数	-	-	-	-	135	157

ここでのパイプラインプロセッサは、マシンに依存しない最適化を行った結果でＮＯＰ命令を挿入して動作可能にしたものである。どのプログラムでもコンパイラを使用した方の結果のほうが悪かった。これは、ＭＯＮＩコードでのプログラムする関係上、あまり長いプログラムを作成することができなかったためではないかと考える。なぜなら、この程度の長さのプログラムでは人間が最適なコードを導き出せる可能性が高いと考えるからである。その例を以下に示す。

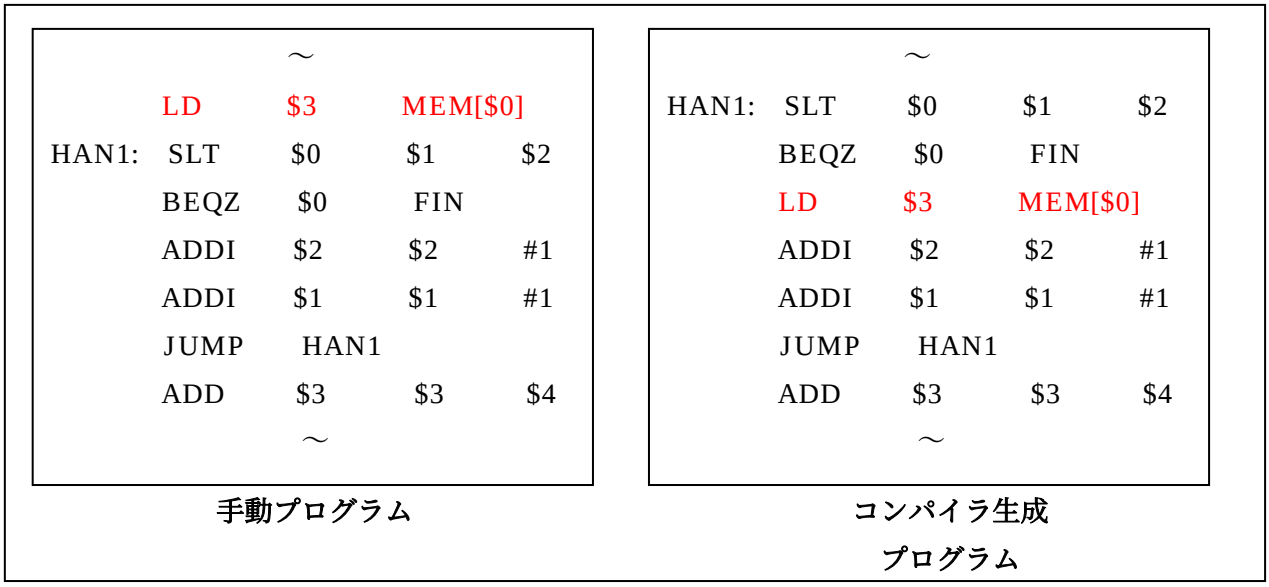

図１７：手動プログラムとコンパイラ生成プログラムとの比較

図１７にあるように手動プログラムでは、ループ内で実行する必要の無い LD 命令がループ外にあるが、コンパイラ生成プログラムでは LD 命令がループ内にある。そのため、実行回数が大幅に増加する原因であると考えられる。

しかし、本コンパイラは、教育用として開発したので、最適化も CMONI 言語と MONI コードの関係が分かりやすいように局所的なものしか行っていない。だから、このような結果が出たのは当然だと考える。性能を求めるのではなく学習者がなぜこのような結果がでるかなど、いろいろな考察をさせることが本コンパイラの大きな役割のひとつだと考える。

また、パイプラインプロセッサではループ内にストールが起こる箇所が１箇所でもあると、ループし続ける間ストール回数が増加してしまう。そのためストール回数が大幅に増加していることがわかる。今回実装することができなかったパイプラインプロセッサ用の最適化コンパイラによって、ストール回数が減少する期待がある。

６． 教育システム上での利用方法の検討

本コンパイラの利用方法と最適化コンパイラの設計について述べる。以下に教育システム上での利用方法についての概要を示す。

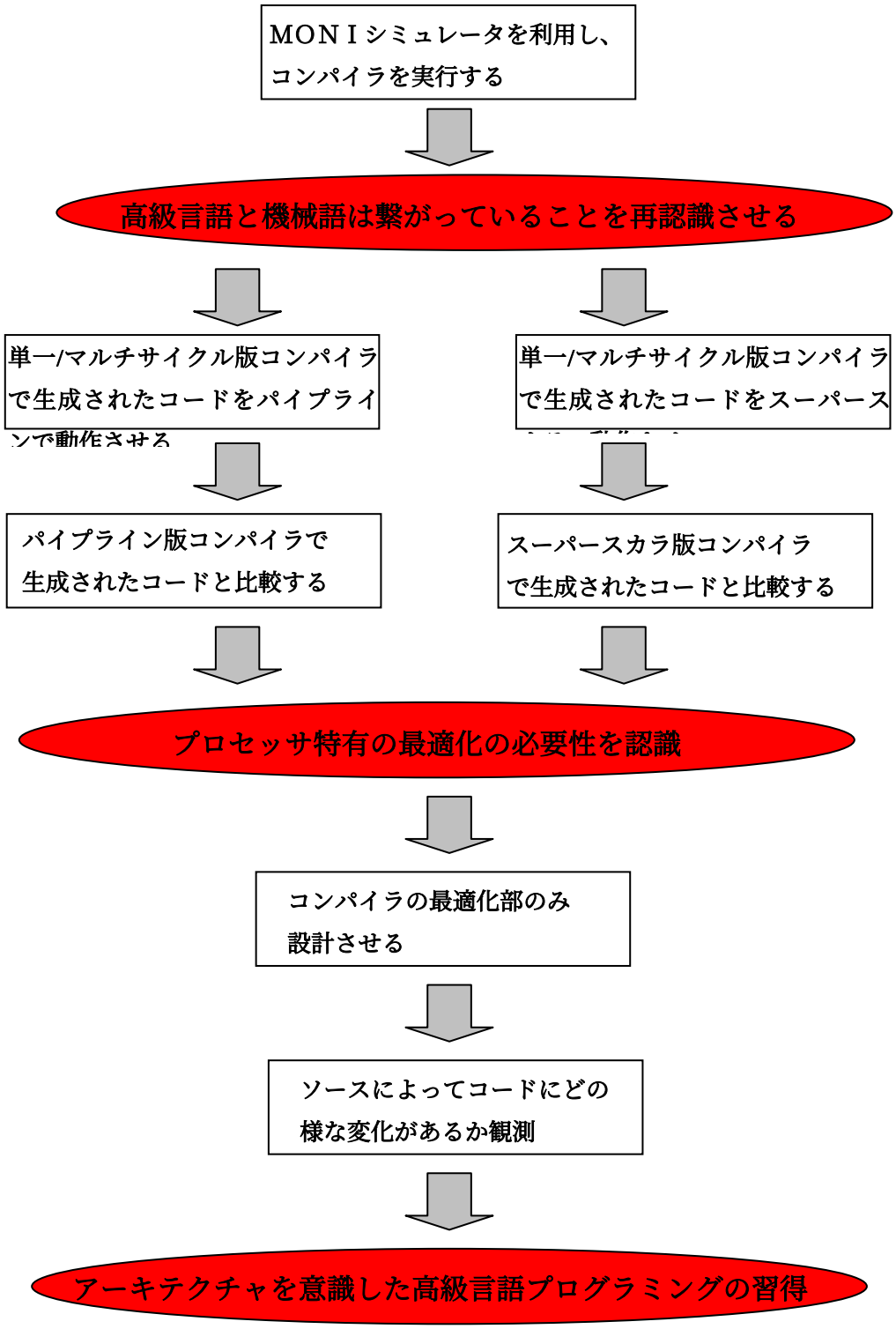

図１８：教育システム上での利用方法

① ＭＯＮＩシミュレータを利用し、コンパイラを実行する。
学習者が本研究で作成した最適化コンパイラを使用し、生成されたコードをＭＯＮＩシミュレータに投入させ、高級言語と機械語はつながっていることを再認識させる。そして、各プロセッサアーキテクチャの理解をさらに促せる。

② 単一/マルチサイクル版のコンパイラが生成したコードをパイプラインプロセッサとスーパースカラプロセッサで動作可能なようにコードを書き換え、ＭＯＮＩシミュレータで観測する。
その後、パイプライン版最適化コンパイラとスーパースカラ版最適化コンパイラでコードを生成し、ＭＯＮＩシミュレータで観測する。前者と後者を比較し、各プロセッサアーキテクチャ特有の最適化の必要性に気づかせる。

③ 学習者が実際にコンパイラを作成する。しかし、作成部分は最適化部のみである。その際に、実行回数を減らすなど具体的な目標をもたせ、学習者間で競わせることでよりよい最適化プログラムの作成を促す。

④ 高級言語プログラミングによって生成されるコードにはどのような変化があるのか観測させる。そのことで同じコンパイラを使用したとしても、高級言語プログラムによって生成されるコードが違うことを認識させる。

⑤ ①～④をへてハード/ソフト・コラーニングシステムにおける最適化コンパイラ作成の目標である、アーキテクチャを意識した高級プログラミングの習得を目指す。

②の学習をさせるために本コンパイラでは、パイプラインプロセッサの最適化、スーパースカラプロセッサの最適化はピープホール最適化をすることに決定した。

また、本研究で開発した最適化コンパイラは教育用を目標としており、Ｃ言語のサブセットで、最適化も必要最低限しか行っていない。しかし、学習者が図１７のプロセスを辿り、アーキテクチャを意識した高級言語プログラミングが身につくことを仮定する。その後の学習プロセスとしてＭＯＮＩシミュレータにＣ言語フルセットを対象とし最適化も性能を重視したコンパイラを組み込み、ＭＯＮＩシミュレータを使用した組み込みソフトウェアを開発ができる環境を整えることも重要だと考える。

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プロセッサアーキテクチャ可変な最適化コンパイラの設計と教育システム上でのコンパイラの利用方法の検討を行った。ＭＯＮＩ命令セットアーキテクチャ、各プロセッサの理解を経て、コンパイラの作成を行った。プロセッサアーキテクチャ可変な最適化コンパイラの作成では、まず、教育用として必要十分なＣ言語のサブセットであるＣＭＯＮＩ言語の定義を行った。Ｃ言語プログラミングでは知ることの無かった、奥深い部分まで理解することができた。そして、コンパイラの作成では字句解析、コンパイラコンパイラである yacc を利用し開発した構文解析、意味解析、最適化、コード生成というコンパイラ一連の設計を行った。最適化では、単一/マルチサイクルプロセッサ用の最適化を設計した。パイプラインプロセッサとスーパースカラプロセッサ用の最適化は本論文で述べた最適化手法を用いて今後設計しなければならない。本コンパイラのプログラム規模は主要ソースプログラムで役５０００行程度のコンパイラである。

手動でアセンブリプログラミングをしたものと本コンパイラを用いて生成したものを比較し考察を行った。手動の方が結果はよいのだが、教育用を主眼に置いて提案、開発を行ってきたのでそれは当たり前の結果だと考えられる。性能よりむしろ、学習者になぜ性能が悪いのかなどを考えることを促し、より深い理解を得させることの方が重要だと考えている。だから、最適化手法は教育用として有意義なものであるかと性能がよいかという２つのトレードオフを考えなければならない。

また、本教育用コンパイラで図１８のような学習フローを経て目標に達した後には、C 言語のフルセットを対象とし、性能を重視した最適化コンパイラを設計しシミュレータに組み込むという課題が挙げられる。シミュレータに組み込むことで組み込みソフトウェアの開発環境が整い、より幅の広い実践的なシステムになると考える。

今後、ハード/ソフト・コラーニングシステムのＦＰＧＡボードコンピュータの設計、シミュレータ、最適化コンパイラの設計という３つの関係をより密にし、実際の教育現場でも十分利用可能なものになり、ハードウェアとソフトウェアの両方が分かる技術者が育つことを望む。

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、卒業生である Tran So Cong 氏、池田修久氏、大八木睦氏、Hoang Anh Tuan 氏、及び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]原田賢一:コンパイラ構成法,共立出版,1999.
[2]中田育男:コンパイラの構成と最適化,朝倉出版,1999.
[3] 池田修久,中村浩一郎,大八木睦,Hoang Anh Tuan,山崎勝弘,小柳滋: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 66 回全国代会 5T-6, 1-115,2004.
[4]大八木睦, 池田修久,山崎勝弘,小柳滋: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 66 回全国代会 5T-6, 1-117,2004.
[5]池田修久:ハード/ソフト・コラーニングシステムに上での FPGA ボードコンピュータの設計と実装 立命館大学大学院理工学研究科修士論文. 2004.
[6]大八木睦: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 立命館大学大学院理工学研究科修士論文. 2004.
[7]池田修久：ハードウェア記述言語による単一サイクル/パイプラインマイクロプロセッサの設計、立命館大学理工学部卒業論文、2002.
[8]大八木睦：ハードウェア記述言語によるマルチサイクル/パイプラインマイクロプロセッサの設計、立命館大学理工学部卒業論文、2002.
[9] 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.
[10] 中村 浩一郎 プロセッサアーキテクチャ教育用 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.
[11]John L.Hennessy,David A.Patterson 著、成田光章訳：コンピュータの構成と設計(上)(下)、日経ＢＰ社、1999.
[12]中田育男：コンパイラ、産業図書、1981.
[13]A.V.エイホ、R.セシイ、J.D.ウルマン著、原田賢一訳：コンパイラ(Ⅰ)(Ⅱ)、サイエンス社、1990.
[14]疋田輝雄、石畑清：コンパイラの理論と実現、共立出版、1988.
[15]Hoang Anh Tuan：Educational Microprocessor Design in a Hardware/Software Co-Learning System、立命館大学大学院理工学研究科修士論文. 2004.
[16]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.
[17]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.
[18]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.
[19]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.

[20]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.
[21]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.
[22]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.