情報工学実験 A（ハードウェア）報告書

32 ビットマイクロプロセッサの設計

氏名: 赤松 佑哉 (akamatsu, YUYA)
学生番号: 09B23595
E-mail: p5785hcf@s.okayama-u.ac.jp

出題日: 2024 年月日
提出日: 2024 年月日
締切日: 2024 年月日

概要

本実験では，ハードウェア記述言語および CAD（Computer Aided Design）ツールを利用したマイクロプロセッサ設計を通じて，論理回路，コンピュータアーキテクチャ，およびコンピュータシステムに関する理解を深めることを目的とする．特に，デジタル回路の設計から高位のシステム構築に至る一連の工程を体験することで，ハードウェアとソフトウェアの連携の重要性を理解することを狙いとする．実験内容としては，まずハードウェア記述言語と CAD ツールの使用方法を学び，それらを用いた簡単な論理回路の設計を行う．最終的な目標は，32 ビットマイクロプロセッサの設計である．実験ではまず，ハードウェア記述言語（HDL）と CAD ツールの使用方法を学び，簡単な論理回路の設計と検証を通して基本的な回路設計の流れを習得した．次に，それらの知識を応用し，最終的には 32 ビットマイクロプロセッサの設計を目指す．プロセッサ設計においては，命令のデコード，レジスタファイルの設計，ALU（算術論理演算器）の構築，制御信号の生成といった基本機能の実装を行った．また，基本的な CPU の演算機能の実現に加えて，除算や乗算といったやや複雑な演算機能の拡張実装や，加算演算における効率化の手法の導入など，発展課題にも取り組んだ．プロセッサ設計の実験では，アセンブラ，シミュレータ，およびハードウェア設計支援システムなど，複数のツールを使用する．それぞれのツールごとに個別の設定を行わず，すべてのツールに共通する設定ファイルが用意されておりそれらを使用し実験をした．

1 設計したプロセッサの概要

実際の設計においては，命令を逐次実行する非パイプライン型プロセッサ，処理を高速化した改良版プロセッサ，そして 5 ステージのパイプラインを導入した高効率プロセッサの，3 パターンのアーキテクチャを比較・実装した．それぞれのプロセッサを `FPGA` にマッピングし，最大動作周波数や消費リソース，処理性能などの観点から詳細な評価を行った．これにより，設計方式の違いが性能や実装規模にどのように影響するかを体系的に比較することができた．以下に，p32 の命令セットの構成，内部構造の特徴，および処理方式の概要について説明する．

(1) 命令セット

p32 は，MIPS アーキテクチャの一部を踏襲した命令セットを備えており，R 形式，I 形式，J 形式という 3 種類の命令フォーマットを採用している．基本的な算術演算，論理演算，分岐・ジャンプ，ロード／ストア操作といった主要な命令を網羅しつつも，複雑な命令は省略することで，プロセッサ設計における命

令の取り扱いを簡素化している．以下の表に，p32 に実装されている代表的な命令をその機能ごとに分類して示す．

表 1　p32 の主な命令セット

種類	命令内容（例）
ロード／ストア命令	`lw`，`sw`，`lb`，`sb`：メモリとレジスタ間のデータ転送を行う命令であり，主にデータの読み書きに用いられる．
演算命令	`add`，`sub`，`and`，`or`，`xor`，`nor`，`sll`，`srl`，`sra`，`addi`，`subi`：算術演算や論理演算，ビットシフト演算を行う命令群で，レジスタ間または即値との演算に対応する．
条件分岐命令	`beq`，`bne`：2 つのレジスタ値を比較し，条件に一致する場合に指定されたアドレスへ分岐する．
ジャンプ命令	`j`，`jal`，`jr`，`jalr`：指定されたアドレスへ無条件にジャンプする命令群で，`jal` や `jalr` はリンク機能を持ち，サブルーチンの呼び出しに用いられる．
その他	`lui`，`syscall`：上位ビットへの即値ロードやシステムコールの実行など，特殊な機能を提供する補助的な命令．

(2) 内部構造の概略

p32 プロセッサの内部構造は，代表的な RISC プロセッサの設計手法に基づいており，以下のような主要コンポーネントによって構成されている．

- **レジスタファイル**：32 本の汎用レジスタ（各 32 ビット幅）を搭載し，2 つの読み出しポートと 1 つの書き込みポートを持つことで，同時に複数のレジスタ操作が可能．
- **実行ユニット（ALU・シフタ）**：加算，減算，論理演算（AND, OR, XOR など），およびシフト操作（論理シフト，算術シフト）を 1 サイクルで実行可能とする．拡張として乗算器や除算器も追加可能であり，将来的な機能拡張も視野に入れて設計されている．
- **制御ユニット**：命令のオペコードやファンクションコードを解析し，各ステージに必要な制御信号（レジスタ選択，ALU 制御，メモリ制御など）を生成する．
- **メモリインターフェース**：命令メモリとデータメモリを明確に分離したハーバードアーキテクチャを採用しており，同時アクセスによる性能向上を図っている．
- **メモリアーキテクチャ**：メモリデータの並びはビッグエンディアン形式で管理され，ネットワーク系や組込み系プロセッサに共通する仕様と親和性がある．

(3) 処理方式の概略

p32 プロセッサでは，設計初期段階では単純な逐次実行方式を用いていたが，最終的には以下に示す 5 ステージのパイプラインアーキテクチャを採用することで，命令の同時並行処理と高スループット化を実現した．

- **IF（Instruction Fetch）**：命令メモリから次に実行すべき命令を取得する．
- **ID（Instruction Decode）**：命令の構文解析を行い，同時にレジスタファイルからオペランドを読み出す．

- **EX（Execute）**：ALU やシフタによって算術・論理演算を実行するステージ．条件分岐命令では条件評価も行われる．
- **MEM（Memory Access）**：ロードやストア命令においてデータメモリへのアクセスを行うステージ．
- **WB（Write Back）**：演算結果やメモリから取得したデータをレジスタファイルに書き戻す．

このようなパイプライン処理を導入することにより，複数の命令を重ねて実行することで命令スループットが向上し，プロセッサ全体の実行効率が大幅に改善された．また，データハザードの発生に対処するため，フォワーディング（バイパス）機構を導入して，依存関係のある命令間でデータを直接転送できるようにしている．さらに，制御ハザードや構造ハザードに対しては，必要に応じて NOP 命令を挿入するなどの手法を取り入れる必要がある．

2　実施状況の報告

今回問われた課題について自分の実施状況について以下の表にまとめる．

表 2　プログラミング課題，設計課題および発展課題の実施状況

課題		状況
(プログラミング課題)		
1. 【プログラミング課題 1】N 個の語の加算		(2) 完了
2. 【プログラミング課題 2】N 語のメモリコピー		(2) 完了
3. 【プログラミング課題 3】乗算		(2) 完了
(設計課題 2)		
4. 【設計課題 2-1】32 ビット加算器	`add32`	(2) 設計完了
5. 【設計課題 2-2】32 ビット ALU	`alu32`	(2) 設計完了
6. 【設計課題 2-3】32 ビットシフタ	`shift32`	(9) 非担当
(発展課題 2)		
7. 【発展課題 2-1】32 ビット整数乗算器	`mult32`	(9) 非担当
8. 【発展課題 2-1】32 ビット整数除算器	`div32`	(2) 設計完了
(設計課題 3)		
9. 【設計課題 3-1】レジスタファイル	`regs32x32`	(2) 設計完了
10. 【設計課題 3-2】実行ユニット	`p32ExecUnit`	(2) 設計完了
11. 【設計課題 3-3】デコードユニット	`p32DecodeUnit`	(2) 設計完了
(設計課題 4)		
12. 【設計課題 4-1】プロセッサ	`p32m1`	(2) 設計完了
13. 【設計課題 4-2】プロセッサ	`p32m2`	(2) 設計完了
14. 【設計課題 4-3】プロセッサ	`p32p1`	(2) 設計完了
(発展課題 4)		
15. 【発展課題 4-1】改良	`add32_cla`	(2) 設計完了
16. 【発展課題 4-2】乗算機能の実装	`p32m12`	(2) 設計完了
17. 【発展課題 4-3】自由課題	`p32m12_v2`	(1) 設計中

3　課題に関する報告

3.1　プログラミング課題に関する報告

3.1.1　設計課題 2-1：32 ビット加算器（add4, add32）

本課題では，32 ビットの加算処理を行う加算器 add32 を設計するにあたり，まず基本単位となる 4 ビット加算器 add4 を構成し，それを 8 個直列に接続することで add32 を構成した．

- **add4 モジュール**：入力 a, b（各 4 ビット）およびキャリー入力 cin を受け取り，4 ビットの和 sum およびキャリー出力 cout を出力する．各ビットの桁ごとにキャリービットを順次計算し，次の桁に伝播させることで加算処理を行っている．加算処理には，論理ゲートによるビットごとの組み合わせ論理を用いた．
- **add32 モジュール**：add4 を 8 段接続し，入力 a, b（各 32 ビット）およびキャリー入力 cin に対して，32 ビットの加算結果 sum，最終キャリー出力 cout，および符号付き演算におけるオーバーフローを示す overflow を出力する．各 add4 は 4 ビットずつ加算を行い，キャリーを次段に渡す構成となっており，全体としてリップルキャリー加算器に相当する．
- **オーバーフロー検出**：符号付き加算におけるオーバーフロー検出は，入力 a と b の最上位ビット（MSB）と加算結果の MSB との組み合わせにより判定している．

以上の構成により，基本的な加算器の動作を段階的に設計・理解することができた．また，より高速な加算器としてキャリールックアヘッド加算器も別途設計・実装・動作確認を行っており，p32 プロセッサにおける加算器選定に際して，速度とリソース使用量のトレードオフについて評価を行う材料とした．(第 3.4 章参照)

3.2　設計課題 2-2：32 ビット ALU（alu32）

本課題では，基本的な算術・論理演算を扱う 32 ビット ALU（算術論理演算器）alu32 を設計・実装した．本 ALU は，入力として 2 つの 32 ビット値 a および b を受け取り，演算結果 out，オーバーフロー検出信号 overflow，およびゼロフラグ zero を出力する．

- **算術演算**：加算（op_add）および減算（op_sub）においては，設計課題 2-1 で構成した add32 モジュールを用いて実装している．減算は 2 の補数の性質を利用し，入力 b のビット反転とキャリーイン 1 を与えることで実現している．
- **論理演算**：AND，OR，XOR，NOR の各演算は，それぞれビットごとの論理演算により直接実現されている．XOR 演算は，ビット単位の論理式（(a & b̃) | (ã & b)）により実装されている．
- **オーバーフロー検出**：加算と減算においてのみ，符号付き演算に基づいたオーバーフロー検出を行っている．論理演算に関してはオーバーフローが発生しないため，常に 0 を出力している．
- **ゼロフラグ**：すべての演算において，結果が全ビット 0 の場合に zero 信号が 1 となるように判定している．

この設計により，RISC プロセッサで要求される基本的な算術および論理演算を網羅的にサポートする ALU を実現することができた．シフト演算はグループの他の班員が作成し実装，動作確認済みである．

3.3 プロセッサ設計課題に関する報告

1. **シミュレーションによる動作確認**
   Quartus Prime に付属する ModelSim（あるいは同等のシミュレータ）を用いて、Verilog HDL で記述したプロセッサをシミュレーションした．基本命令（`add`, `sub`, `lw`, `sw`, `beq`, `j` など）を含むテストベンチを作成し，命令の逐次実行とレジスタ・メモリの値の変化が期待通りであることを確認した．
2. **テストプログラムによる確認**
   プロセッサの命令実行機能の動作確認として，アセンブリプログラム `sum10.s` を用いてテストを行った．このプログラムは，引数として与えられた整数（ここでは 10）に対して，その値までの整数を再帰的に加算する関数 `sum` を呼び出す構成となっており，以下のような特徴を持つ．
   - 関数呼び出しと戻り：`jal`, `jr` 命令を用いた再帰関数呼び出し
   - スタック操作：レジスタの保存／復元に `sw`, `lw` を用い，`fp` を使用してフレームベースのスタック管理を実装
   - 分岐命令の使用：条件分岐に `bne`, 比較に `slt` を使用
   - 算術命令の確認：`addi`, `add`, `ori` による加算・即値処理を含む

   このテストプログラムは，引数 $a_0 = 10$ に対して，$1 + 2 + \cdots + 10 = 55$ を求めることを目的としており，実行後にはその結果がレジスタ `$v0` に格納される．
   シミュレーション上での動作確認では，ModelSim を用いて命令のデコード・実行・メモリアクセスが期待通りに行われることを波形レベルで検証した．関数呼び出しにおいて `ra`, `fp` の保存／復元，および戻りアドレス制御が正しく行われていること，正常に動作したときのサンプル実行結果がシミュレーションの期待結果と一致することを確認した．

設計したプロセッサは，Intel 社の FPGA デバイス `Cyclone IV E (EP4CE115F29C7)` をターゲットとして，論理合成・配置配線・静的タイミング解析を実施した．以下にターゲットデバイスの，各要素についてまとめた表 3 を以下に示す．

表 3　ターゲット FPGA デバイスとそのリソース量

ターゲットデバイス	Intel Cyclone IV E (EP4CE115F29C7)
Logic Element (LE) 数	114,480
レジスタ数	114,480
メモリ容量 (ビット)	4,981,312
9 ビット乗算器	432

正常に，作成したプロセッサが動作していることを確認したのちに各プロセッサについて動作周波数，ロジックエレメントの数等の記録を以下の表 4 にまとめた．

設計した 3 種類のプロセッサ（p16m1，p16m2，p16p1）に対して，Intel 社製 FPGA（Cyclone IV E: EP4CE115F29C7）をターゲットとした論理合成・配置配線・静的タイミング解析を行い，得られた各種諸量（最大動作周波数，論理素子使用量，レジスタ数など）を表 4 に示した．さらに，各プロセッサの実行性能（命令数，サイクル数，Fmax に基づく実行時間）についても比較を行い，設計上の改良点が結果にど

表 4　FPGA への論理合成等で得られた諸量のまとめ

モジュール	最大動作周波数 Fmax		LE 数 (使用率)	CF 数 (使用率)	レジスタ数 (使用率)
	85 °C Model	0 °C Model			
プロセッサ `p16m1`	56.85	61.64	3764 (3%)	3558 (3%)	1379 (1 %)
プロセッサ `p16m2`	53.41	58.45	3,842(3%)	3,609 (3%)	1377 (1%)
プロセッサ `p16p1`	40.52	43.95	4,089 (3%)	3,995(3%)	1,416 (1%)

（Fmax の単位は MHz）

のように反映されたかを定量的に考察できるようにした.

まず，すべてのプロセッサにおいて加算器には，基本課題で設計した単純なリップルキャリー加算器ではなく，発展課題として実装した**キャリールックアヘッド加算器（Lookahead Carry Adder）**を採用している点に注目したい．この加算器は，ビットごとにキャリー伝播を待たずに並列に計算する構造を持ち，ALU のクリティカルパスを大幅に短縮することができる．実際，Fmax の向上にもつながっており，各プロセッサで 60 MHz 前後の最大動作周波数が得られているのはこの発展加算器の貢献によるものである.

p16m1（単純マルチサイクル型）は最も基本的な構成であり，各命令のステージが順番に実行される方式である．443 命令を実行するのに 2215 サイクルを要し，最終的な実行時間は 0.21 μs であった．この構成は設計・実装が比較的容易である一方，命令ごとの平均サイクル数は約 5.0 と高く，効率性の面では他構成に劣る.

p16m2（改良型マルチサイクル）は制御ユニットと状態遷移の最適化を通じて命令の種類に応じた実行ステージ数を柔軟に設定できるようになっており，443 命令に対して 1584 サイクルと約 29% の削減を実現している．Fmax はやや低下（58.45 MHz）しているが，それでも実行時間は 0.18 μs と短縮され，制御回路の改良が明確に性能向上につながっている.

p16p1（5 ステージパイプライン型）は最も複雑な構成であり，命令の IF〜WB までを重ねて実行することで，大幅なスループット向上を実現している．命令数が 508 と他構成より多いにもかかわらず，総サイクル数はわずか 516 であり，1 命令あたりの実行時間が著しく短縮されている．Fmax 自体は 43.95 MHz と他より低めではあるが，実行時間は最短の 0.065 μs であり，パフォーマンス指標として最も優れている結果となった.

他の設計者との比較において，すべてのプロセッサで + 演算子や高位合成ライブラリを用いず，明示的に加算器を設計したことから，採用した加算器の構造がプロセッサ全体の性能に大きく影響を与えることが明らかとなった．私は発展課題として，基本課題で用いた単純なリップルキャリー加算器ではなく，キャリールックアヘッド（CLA: Carry Lookahead Adder）を実装した．その結果，リップルキャリー加算器を用いた設計と比較して，論理合成結果における最大動作周波数（Fmax）が明確に高く，処理性能の向上が確認された．一方で，より高速な並列 Prefix 加算器（例：Sklansky 型や Kogge-Stone 型）を実装した設計と比較すると，私の設計はわずかに Fmax や実行時間で劣っていた．この結果は，プロセッサの設計改善において，加算器が重要な構成要素であることを強く示唆しており，制御回路やパイプライン構成の工夫よりもまずは加算器の効率化が性能向上に直結することを再認識した.

以上の結果を踏まえると，設計課題および発展課題を通じて実施した各種の改良（加算器の高速化，制御ロジックの最適化，パイプライン構成の導入など）は，論理合成結果および実行性能に明確に定量的な効果として現れており，プロセッサ設計におけるアーキテクチャ選択と低レベル回路最適化の重要性を理解できた.

3.4 追加課題や発展課題に関する報告

3.4.1 32bit 除算器の実装

本発展課題では，32 ビットの符号なし除算器（div32_v2）を自作し，プロセッサに組み込んだ．本除算器は，**逐次シフト減算法（Restoring Division）**に基づく構成となっており，1 クロックサイクルごとに商の各ビットを計算していく設計である．以下に，動作の各ステージを示す：

- 初期化：被除数 a を 64 ビットに拡張し（上位 32 ビットにゼロ），除数 b およびループカウンタ stat をセットする（1 サイクル）。
- ループステート（st0）：
  - 上位ビット（a0[63:32]）から b を減算し，結果の符号によって商のビットを決定（0 または 1）。
  - 結果に応じて a0 を更新（減算値 or 未変更）し，次のシフト処理へ進む。
  - stat を 1 ビット右シフトすることで，ループ回数（全 32 回）を制御。
- 終了ステート（st1）：
  - 商（下位 32 ビット）と剰余（上位 32 ビット）を out_en を通じて出力。

本除算器は初期化に 1 サイクル，ループ処理に最大 32 サイクルを必要とするため，1 回の除算処理におおよそ 33 サイクルを要する．従って，演算器としてのスループットは高くはないが，回路構成が比較的単純であり，面積効率が良く FPGA 実装にも適している．

また，逐次処理であるため並列性には欠けるものの，演算ごとの結果の精度やデバッグの容易さの観点からは実装・検証しやすい構成となっている．今後，さらなる高速化を目指す場合は，非復帰除算器（non-restoring division）や SRT 除算器，さらには Newton-Raphson 法や Goldschmidt 法を用いた除算回路への拡張が考えられる．

3.4.2 プロセッサの改良

プロセッサの性能改善を検討する中で，Quartus Prime による静的タイミング解析のレポートを詳細に確認したところ，**加算を含む命令（例：add, addi, sub）がクリティカルパスとなっている**ことが判明した．これは，ALU 内の加算器におけるキャリー伝播遅延が，クロック周波数の上限を制約していることを意味しており，性能ボトルネックとして顕在化していた．プロセッサの ALU における加算処理は，命令の多くに共通して含まれる基本演算であり，その遅延はプロセッサ全体のクロック制約に直結する．従来のリップルキャリー加算器（Ripple Carry Adder, RCA）では，1 ビットごとにキャリー信号が逐次伝搬するため，全体の遅延がビット数に比例して増加しやすく，特に 32 ビット幅では遅延が顕著となる．この問題を解決するため，従来のリップルキャリー加算器（Ripple Carry Adder, RCA）に代えて，**32 ビット桁上げ先見加算器（Carry Lookahead Adder, CLA）**を自ら実装した．この加算器は，キャリービットの伝播をグループ単位で並列に計算する構造を持ち，全体の計算遅延を大幅に短縮できる．しかし 32 ビットともなると論理演算が複雑となる．桁上げの式は規則性があるためまず，任意ビット目のキャリーを中間変数だけで表した式を出力する C++ プログラムを作成した．いかにそのコードを示す．

```
1: #include <iostream>
2: #include <fstream>
3: #include <string>
4:
5: using namespace std;
6:
7: // キャリー c_i の論理式を文字列で生成する関数（セミコロンなし）
```

```
 8: string generate_carry_expr(int i) {
 9:     if (i == 0) return "c0 = c0"; // 初期キャリー
10:
11:     string expr;
12:     for (int k = 0; k < i; ++k) {
13:         if (!expr.empty()) expr += " | ";
14:         expr += "(";
15:         for (int j = i - 1; j > k; --j) {
16:             expr += "p" + to_string(j) + " & ";
17:         }
18:         expr += "g" + to_string(k) + ")";
19:     }
20:
21:     // 最後の項: p[i-1] & ... & p0 & c0
22:     expr += " | (";
23:     for (int j = i - 1; j >= 0; --j) {
24:         expr += "p" + to_string(j) + " & ";
25:     }
26:     expr += "c0)";
27:
28:     return "c" + to_string(i) + " = " + expr;
29: }
30:
31: int main() {
32:     ofstream outfile("CLA.txt");
33:     if (!outfile) {
34:         cerr << "CLA.txt を開けませんでした。\n";
35:         return 1;
36:     }
37:
38:     for (int i = 0; i <= 31; ++i) {
39:         string expr = generate_carry_expr(i);
40:         outfile << expr << "\n";
41:     }
42:
43:     outfile.close();
44:     cout << "a.txt に c0〜c31 の式を出力しました。\n";
45:     return 0;
46: }
```

今回中間変数をデバッグ，可読性のため使用した．使用しなければサイクルを一つ減らせるためわずかに効率が向上するように思えるが実際には，回路面積が膨大になりフィッティングしみるとかえって遅くなる可能背もある．さらには，可読性が非常に悪くなるのでデバッグが非常に困難である，

本加算器を各種プロセッサ構成（p16m1, p16m2, p16p1）に統一的に導入した結果，いずれのプロセッサにおいても最大動作周波数（Fmax）が向上し，明確な性能改善が見られた．具体的には，p16m1 においてFmax が 60.98 MHz，p16m2 で 58.45 MHz，p16p1 でも 43.95 MHz を記録しており，いずれも加算器がボトルネックとなっていた状態から改善が図られたことが定量的に示された．

この経験を通して，演算器，特に加算器の設計がプロセッサ全体の性能に与える影響の大きさを実感するとともに，単体の回路改善がシステムレベルの最適化へと直結することを学ぶことができた．

3.4.3 乗算機能の実装

p32 実行ユニット内に 32 ビット乗算器（mult32）を組み込み、64 ビット結果を HI/LO レジスタに格納する方式を採用している。p32ExecUnit3 モジュールの乗算命令処理は、ex_mult 関数により実現され、以下のように動作する。

- 入力の 32 ビット値 a, b を mult32 に渡し、64 ビット乗算結果を得る。
- 64 ビット結果の下位 32 ビットを result へ、上位 32 ビットを result2 へ格納。

- フラグ `is_mult` を真に設定し、乗算処理中であることを明示する。

乗算処理中の `is_mult` フラグは、制御ユニットや後段パイプラインに対して乗算命令の特殊性を示すために用いられる。例えば、64 ビット結果の 2 つのレジスタへの分割格納や、結果取得までの待機制御などで利用される。これは乗算演算が，ほかの演算と比べて特定のマシンサイクルで演算終了とならないため，フラグを立てて計算中か明示する必要があるためである．階乗結果は最終的には HI/LO レジスタに保存されているので実行結果が正しいか確認し，正常に動作していることを確かめた．除算命令については実装中である．32 ビット除算器は第 3.4.1 章にて実装完了している．除算命令も同様に，そのアルゴリズムによってサイクル数が異なる．よって，実装方針としては乗算命令実装と同様である．まずフラグを定義して，関数の出力を out_en のように出力するのではなく乗算器のように，二つの３２ビット長の出力とした．実際には商，剰余を分けて出力している．それを `result`，`result2` に割り当てて，そこからレジスタへの格納までは乗算器と同様なので確認されたい．

4　検討・考察

5　工夫した点や特に力を注いだ点

私が特に力を注いだ点はやはりプロセッサの効率化，拡張である．私はまず逐次シフト減算法に基づく除算器の自作に取り組み，逐次演算による構成ながらも動作の正確性とデバッグ容易性を両立させる実装を実現した．さらに，加算命令群が性能上の律速要因となっていることを静的タイミング解析により特定し，これに対して桁上げ先見加算器（Carry Lookahead Adder）を独自に設計・導入することで，プロセッサ全体の最大動作周波数を向上させる成果を得た．また，乗算命令についても 64 ビット結果を適切に扱う機構を整備し，制御フラグを用いたパイプライン制御との整合性を確保するなど，システム全体の整合的な拡張を図った．これらの更なる詳細は第 3.4 章に記述しているので確認されたい．これら一連の実装・改良作業を通じて，演算器設計がプロセッサ全体の性能および機能性に与える影響の大きさを実感するとともに，低位レベルの回路設計と高位レベルのアーキテクチャ設計とが密接に連関していることを学んだ．

6　本実験の成果と実験を実施して得られたこと