九州大学 CPC Lab 研究会
2021 年度 第 13 回 研究会資料 TR2021-013

プロセッサ設計演習

ウン クアン イー
九州大学 工学部 電気情報工学科 4 年

2021 年 07 月 05 日

1 はじめに

今後, コンピュータアーキテクチャに関する研究を行う上で, プロセッサの動作原理に対する理解とプロセッサの設計手法の会得が不可欠である. そこで, プロセッサの動作原理を理解することを目的とし, 命令パイプライン構造を持つプロセッサを設計・実装した. プロセッサはメモリから命令をフェッチして実行することを繰り返し, プログラムを実行する. また, 命令パイプライン構造を持つプロセッサは, フェッチした命令を複数のステージに分けて実行するような構造を持つため, 1 クロックサイクル内に複数の命令を並行して実行することが可能である [2].

プロセッサの設計・実装は, Verilog-HDL というハードウェア記述言語を用いて行った. さらに, プロセッサのクロックサイクル数と最小動作クロック周期を削減するために, 分岐予測の導入, ならびに, クリティカルパスの短縮を行った. 分岐予測の導入により, テストプログラムの実行クロックサイクル数が平均で 26.60% 減少した. また, ボトルネックとなるステージの処理の一部を他のステージで行い, クリティカルパスの短縮をすることで, 最小動作クロック周期が短縮前に比べて 33.33% 減少した.

本稿では, 第 2 章でプロセッサの仕様について述べる. 第 3 章でプロセッサの機能検証の方法, ならびに, プロセッサの性能評価と論理合成の方法と結果を示す. 第 4 章でプロセッサの性能改善方法として, 分岐予測, ならびに, クリティカルパスの短縮について述べる. 最後に第 5 章でまとめを行う. 付録 A に設計したプロセッサの設計図 (図 6) を載せている. プロセッサのソースコードは GitHub [1] 上に置いてある.

2 プロセッサの仕様

2.1 命令セット

設計したプロセッサがサポートしている命令の一覧を付録 B の表 9 に示す. この命令セットは, RISC-V 32I 命令セットの一部である. RISC-V 32I にあり、このプロセッサがサポートしていない命令は, `fence`, `fence.i`, `sfence.vma`, `ebreak`, `uret`, `sret`, `wfi` である. 機能検証と性能評価のプログラムでは, フェンスを使うことがないので, `fence`, `fence.i`, `sfence.vma` を実装しない. また, 今回のプロセッサは, デバグ, ユーザモード, スーパーバイザモード, 外部割り込みをサポートしていないため, `ebreak`, `uret`, `sret`, `wfi` を実装しない.

2.2 外部インタフェース

図 1 にプロセッサの外部インタフェースを示す. 信号線名の後ろに # が記述されている信号線は負論理であり, # が記述されていない信号線は正論理である.

図 1 のそれぞれの信号線の説明は以下の通りである.

- IAD (Instruction ADdress Bus)
  命令メモリへの 32bit のアクセスアドレスパス.

[1] https://github.com/kuanyi-ng/rv32i-processor

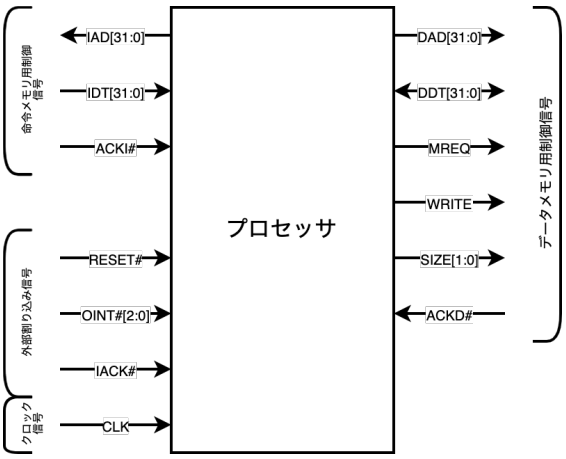


図 1: プロセッサの外部インタフェース

- IDT (Instruction DaTa Bus)
  命令メモリからの 32bit のデータパス.
- ACKI# (ACKnowledge from Instruction memory)
  命令メモリへのアクセスに対するアクノリッジ信号. 命令メモリにアクセスして, この信号がインアクティブであれば, 読み出し・書き込みが完了していないことを意味する.
- RESET#
  リセット信号.
- OINT#
  外部からの割り込みを示す信号. 外部からの割り込みがあった時に, 対応するビットがアクティブになる.
- IACK# (Interruption ACKnowledge)
  外部の割り込みを処理している時にアクティブになる信号.
- CLK (CLocK)
  クロック信号.
- DAD (Data ADdress bus)
  データメモリへの 32bit のアクセスアドレスパス.
- DDT (Data DaTa bu)
  データメモリからの 32bit のデータパス.
- MREQ
  データメモリに対するアクセス (読み出し・書き込み) をリクエストするための信号. データメモリへアクセスする前にリクエスト信号をアクティブにする必要がある.
- WRITE
  データメモリへの書き込みをリクエストする信号. データメモリにデータを書き込む時にこの信号をアクティブにする必要がある.
- SIZE
  データメモリへのアクセスサイズを示す.
  - word アクセス: SIZE = 00
  - halfword アクセス: SIZE = 01
  - byte アクセス: SIZE = 10
- ACKD# (ACKnowledge from Data memory)
  データメモリへのアクセスに対するアクノリッジ信号. データメモリにアクセスして, この信号がインアクティブであれば, 読み出し・書き込みが完了していないことを意味する.

命令メモリとデータメモリはシミュレーション環境で用意されているため, プロセッサの中に実装しない. メモリはリトルエンディアン方式を採用する. メモリのアドレスは 1B ごとに振り分けられ, 1 つのアドレスに 32bit のデータを保持することができる.

また, シミュレーション環境において, 外部からの割り込みが発生しないため, OINT 入力信号の処理と, IACK 出力信号の生成を行わない.

2.3 特権モードと CSR レジスタ

このプロセッサはマシンモードのみをサポートしている. そのため, 例外処理から戻る `mret`, `sret`, と `uret`, 3 つ命令のうち, `mret` 命令だけをサポートする. また, マーシンモードに合わせて使える CSR レジスタの一覧 [3] [4] を表 1 に示す.

2.4 例外・割り込み処理

このプロセッサが対応している例外・割り込み処理の優先順位は以下の通りである.

1. リセット
   外部信号によってプロセッサの内部状態 (レジスタとメモリ) をリセットする.

表 1: マシンモードの CSR レジスタ

CSR レジスタ名	内容
mvendorid	ベンダー ID
marchid	アーキテクチャ ID
mimpid	実装 ID
mhartid	ハードウェアスレッド ID
mstatus	マシンステータスレジスタ
misa	ISA と拡張機能
mie	マシン割り込み有効化
mtvec	マシン・トラップ・ベクトル
mcounteren	マシンカウント有効化
mscratch	マシン・スクラッチ
mepc	マシン例外 PC
mcause	マシン例外原因
mtval	マシン・トラップ値
mip	マシン割込処理待ち

2. 不正命令
   サポートしていない命令を解読した時に例外処理に移す.

3. 命令アクセス・ミスアライメント
   命令のアドレスは常に 4 の倍数であるため, 命令メモリに対してそれ以外のアドレスにアクセスする時に例外処理に移す.

4. ECALL 命令
   アプリケーション側がシステムコールを呼び出す時に ECALL 命令を使う. システムコールの呼び出しを例外処理で行う.

リセットがアクティブになる時, プロセッサのメモリ, レジスタファイル, パイプラインレジスタと PC を初期値にリセットする. また, 不正命令, 命令アクセス・ミスアライメント, または, ECALL 命令を検知した場合, 例外処理に移すために, PC の値を 0x0000_0000 に設定し, プログラマーが記述した例外処理のプログラムを実行する. なお, 例外処理前のプログラムに処理を移すために, 例外処理プログラムの最後に MRET 命令を使用する必要がある.

2.5 パイプライン処理

今回設計したプロセッサは, 1 つの命令を 5 つのパイプラインステージに分けて実行する. それぞれのステージの名前と役割は以下の通りである.

1. IF ステージ
   次に実行する命令を命令メモリから読み出す.

2. ID ステージ
   命令を解読して, EX ステージで行われる演算に必要な入力を用意する.

3. EX ステージ
   命令で必要な演算を行う.

4. MEM ステージ
   データメモリへのアクセス (読み出し・書き込み) を行う.

5. WB ステージ
   汎用レジスタへデータを書き込む.

命令のパイプライン処理では, それぞれのステージにおいて, どのように命令を実行するかを指定する制御信号が必要である. たとえば, EX ステージで ALU を使ってどんな演算を行うか, MEM ステージでメモリにアクセスするかどうか, WB ステージで汎用レジスタに値を書き込むかなどの制御信号がある. 本設計では, 各々のステージに必要な制御信号は各ステージで生成される.

2.6 データハザードとその解決法

パイプライン処理では, 異なるステージにおいて, 異なる命令が実行される. これにより, パイプライン処理をしない場合と異なる結果が得られることがある. その原因の 1 つはデータハザードである. データハザードには, 主に以下の 3 種類のハザードがある.

1. RAW (Read After Write) ハザード

2. WAR (Write After Read) ハザード

3. WAW (Write After Write) ハザード

今回設計したプロセッサは, 命令を順序通りに実行するイン・オーダー実行方式を採用している. そのため, イン・オーダー実行において発生しない WAR ハザードと WAW ハザードの対処を行う必要はない. この節では, RAW ハザードについて説明した後に, 今回のプロセッサ設計に採用された解決法について述べる.

2 つの命令の間にデータ依存性が存在する時, 先に実行される命令が汎用レジスタを更新する前に, 後

で実行される命令が同じレジスタの古い値を読み出してしまうことがある. これによって, 正確な演算結果を得ることができない. この現象を, RAW (Read After Write) ハザードという. 以下, RAW ハザードが発生する状況を記述する.

1. 命令 m はレジスタ xn を更新し, 命令 $m+1$ はレジスタ xn の値を用いた演算を行う時.
2. 命令 m はレジスタ xn を更新し, 命令 $m+2$ はレジスタ xn の値を用いた演算を行う時.

RAW ハザードを解決するために, データフォワーディングとパイプラインストールの 2 つの方法がある. RAW ハザードが発生する状況とその解決法を表 2 に示す.

2.6.1 データフォワーディング

データフォワーディングとは, EX (または, MEM) ステージにある命令 m の演算結果を ID (または, EX) ステージにある命令 $m+1$ (または, 命令 $m+2$) に渡し, 命令 $m+1$ (または, 命令 $m+2$) の EX (または, MEM) ステージで使用する方法である. この時, 命令 $m+1$ (または, 命令 $m+2$) はレジスタ xn の最新値を用いて演算を行うため, 正確な結果が得られる.

データフォワーディングにより, 上記の RAW ハザードが発生する状況の中で, 命令 m がロード命令以外の状況の対処ができる. 命令 m がロード命令の場合は, パイプラインストールを用いる.

2.6.2 パイプラインストール

パイプラインストールとは, パイプラインの各ステージにある命令を次のステージに進まないようにする方法である. データフォワーディングに必要なデータが生成されるまでに, パイプラインをストールすれば, データフォワーディングで解決できる状況が作られるため, RAW ハザードが解決できる.

2.7 制御ハザードとその解決法

パイプライン処理が, パイプライン処理をしない場合と異なる結果が得られる原因のもう 1 つは制御ハザードである. この節では, 分岐命令による制御ハザードについて説明した後に, 今回のプロセッサの設計で採用した解決法について述べる.

分岐命令が存在する時に, その命令の分岐方向によって次に実行する命令が決まる. 分岐方向とは, 分岐命令が分岐するか, 分岐しないかのことを意味する. パイプライン処理をしない場合, 分岐命令の実行を注意する必要はないが, 今回の 5 段パイプライン処理では, 分岐命令の分岐方向が EX ステージにならないと判明しない. その上, 分岐命令の後に続く命令がすでにパイプラインの IF ステージと ID ステージにおいて実行されている. 分岐方向が「分岐しない」ならば, 問題なく命令の実行を続けることができる. しかしながら, 分岐方向が「分岐する」ならば, IF ステージと ID ステージにある命令は実行してはいけない. IF ステージと ID ステージの命令を実行してしまうと, データメモリ, または, 汎用レジスタが更新されることがある. それにより, プログラムの実行結果が正しくなくなる.

2.7.1 パイプラインフラッシュ

制御ハザードの解決法の 1 つとして, パイプラインストールがある. 分岐命令の分岐方向が判明されるまでに, 分岐命令の後に続く命令がパイプラインに入らないように, パイプラインをストールする方法である. そして, 分岐方向が判明した後, パイプラインストールを解除し, 分岐方向を基に次の命令をフェッチする. しかしながら, パイプラインストールを採用すると, 分岐命令が発生する度に 2 クロックサイクル分が無駄になってしまう.

パイプラインストールの他に, パイプラインフラッシュによって制御ハザードを解決する方法がある. 分岐命令に続く命令がプロセッサの内部状態 (メモリと汎用レジスタ) に対する変更を無効化することを「パイプラインをフラッシュする」という. たとえば, 分岐命令の分岐方向が「分岐する」場合でも, 内部状態の更新に関連する制御信号を無効化したまま, IF ステージと ID ステージにある命令の実行を続ける.

その結果, 分岐結果が「分岐する」場合では変わらず 2 クロックサイクル分が無駄になるが, 分岐方向が「分岐しない」場合では, クロックサイクルの無駄が生じなくなる. パイプラインストールと比べると, 分岐によるクロックサイクル数のペナルティが少ないため, 今回の設計にパイプラインフラッシュ

表 2: RAW ハザードに関わる命令とその解決法

レジスタ xn を更新する命令	レジスタ xn を用いる命令	解決法
ロード命令以外	ストア命令	データフォワーディング (EX または MEM → ID)
ロード命令以外	ストア命令以外	データフォワーディング (EX または MEM → ID)
ロード命令	ストア命令	データフォワーディング (MEM → EX)
ロード命令	ストア命令以外	パイプラインストール

を採用した.

2.8 プロセッサの名前: opt

今回のプロセッサ設計演習では, 1 つずつモジュールを作成して結合するように, 一歩ずつ前へ進めればいいと意識して取り組んでいた. この思いをプロセッサの名前にしたいと思った.

「1 つステップずつ進む」の英語が「One Step At A Time」で, これを数学ぽっく言い換えれば,「1 単位時間あたり 1 ステップ」になる. その英語が「One step Per Time」であり, 頭文字の「opt」を取ってプロセッサの名前にした.

3 プロセッサの機能検証, 性能評価と論理合成

3.1 機能検証

付録 C の表 10 で示したプログラムを用いてプロセッサの機能検証を行う. 機能検証は Verilog-HDL で記述したプロセッサに対し, 論理シミュレーター xmverilog と波形ツール SimVision を用いてシミュレーションを行った. テストプログラムが正しく実行され、正確な出力が得られたことを確認した.

3.2 性能評価と論理合成

設計したプロセッサの性能を, プログラム実行のクロックサイクル数, 最小動作クロック周期, 面積, ならびに, 消費電力で評価する.

ベンチマークプログラム MiBench [1] の一部を用い, プログラム実行のクロックサイクル数を求めた.

表 3: ベンチマークプログラムの実行クロックサイクル数 (改善前)

ベンチマークプログラム	クロックサイクル数
stringsearch	10594
bitcnts	56040
dijkstra	4079473

なお, ベンチマークプログラムが扱うデータのサイズが `large`, `small`, `test` の 3 種類があるが, 評価に用いたのは `test` サイズのプログラムである. 今回の性能評価に使われたプログラムの一覧を付録 C の表 11 に示す.

次に, 論理合成ツール Design Compiler を用いて論理合成を行い, 最小動作クロック周期, 面積, ならびに, 消費電力を測定した.

最小動作クロック周期の求め方について説明する.

1. タイミング制約を 10.00ns と設定して論理合成を行い, そのの結果 slack (与えたタイミング制約と最大遅延時間との差) が正であることを確認する.

2. タイミング制約を 9.00ns, 8.00ns, . . . のように 1.00ns ずつ下げ, slack が負になるタイミング制約を見つける.

たとえば, タイミング制約を 5.00ns に設定した時に slack が負になったら, 作成したプロセッサの最小動作クロック周期は 6.00ns になる.

各ベンチマークのプログラムの実行に必要なクロックサイクル数を表 3 に示す. また, プロセッサの最小動作クロック周期, 面積, ならびに, 消費電力の測定結果を表 4 に示す.

表 4: 論理合成の結果 (改善前)

最小動作クロック周期 [ns]	面積 [μm^2]	消費電力 [mW]
6.00	357534.7228	7.5732

4 プロセッサの性能改善方法と評価

4.1 プログラムの実行時間と改善方法

プロセッサの速度性能面を向上する方法の 1 つは, プログラムの実行時間を短くすることである. プログラムの実行時間は式 (1) で求められる.

$$\begin{aligned} &\text{プログラムの実行時間} \\ =&\text{プログラムの実行クロックサイクル数} \\ &\times \text{プロセッサの動作クロック周期} \end{aligned} \quad (1)$$

式 (1) より, プログラムの実行クロックサイクル数, プログラムの動作クロック周期, または, 両方を小さくすると, プログラムの実行時間が短くなることが分かる.

今回設計したプロセッサのプログラムの実行クロックサイクル数を減少させるために, 動的分岐予測を導入した. 動的分岐予測の詳細は, 4.2 節で述べる.

分岐予測を実装することで, プログラムの実行クロックサイクルが, 平均で 26.60% で減少した. しかしながら, プロセッサの最小動作クロック周期が 6.00ns から 9.00ns に増加した. 分岐予測実装前と実装後のプロセッサを比較した時に, 分岐予測を実装する前のプロセッサの方が, プログラムの実行時間が短いことが分かった. そこで, 論理合成の結果を基に, クリティカルパスの短縮を試みた. クリティカルパスの短縮方法については, 4.3 節で述べる.

4.2 動的分岐予測

プロセッサが動的分岐予測機能を持つ場合, 分岐する時に無駄になるクロックサイクル数を減らすことができる. この節では, 今回の設計における動的分岐予測の実装方法と評価結果について述べる.

4.2.1 分岐予測の対象命令

分岐予測の対象命令を, 無条件分岐命令と条件付き分岐命令 (それぞれ表 9 にある J 形式と B 形式の命令) とする. 無条件分岐命令は必ず分岐し, 条件付き分岐命令よりも予測しやすいため, 分岐予測の対象に含めることにした.

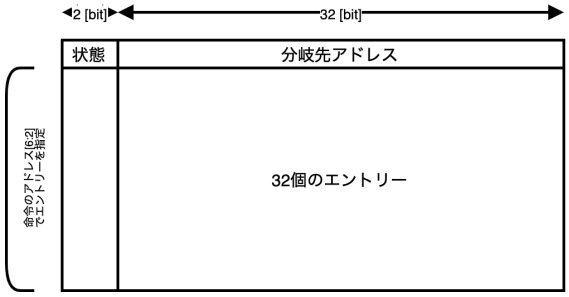


図 2: 予測器の参照テーブル

表 5: 予測器の状態信号と分岐方向の予測

状態信号	分岐方向の予測
00 (STRONG_NOT_TAKE)	分岐しない
01 (WEAK_NOT_TAKE)	分岐しない
10 (WEAK_TAKE)	分岐する
11 (STRONG_TAKE)	分岐する

4.2.2 分岐方向と分岐アドレスの予測方法

無条件分岐命令と条件付き分岐命令の予測しやすさに違いがあるため, 命令ごとに予測できるローカル予測器を採用した [2]. 実装では, エントリー数が 32 の参照テーブル (図 2) を用意した. 命令のアドレスの下位 2 ビットは常に `00` であるため, 参照テーブルのエントリーは, 対象命令のアドレスの 6 ビット目から 2 ビット目までの値で指定する. そして, 1 つのエントリーに 2bit の状態信号と 32bit の分岐先アドレスを保持する. 状態信号の値と分岐方向の予測を表 5 に示す.

命令メモリからフェッチした命令が予測の対象命令である時に, 参照テーブルのエントリーの状態信号と分岐先アドレスを基に, 分岐方向と分岐先アドレスを予測する. なお, 予測の対象命令ではない時に,「分岐しない」と予測する.

4.2.3 予測器の参照テーブルの更新

分岐方向, または分岐先アドレスの結果が判明すると, 予測が成功したかどうかを基に, 予測器の参照テーブルの更新を行う. 参照テーブルの該当エントリーに対し, 状態信号を図 3 のように更新し, 分岐先

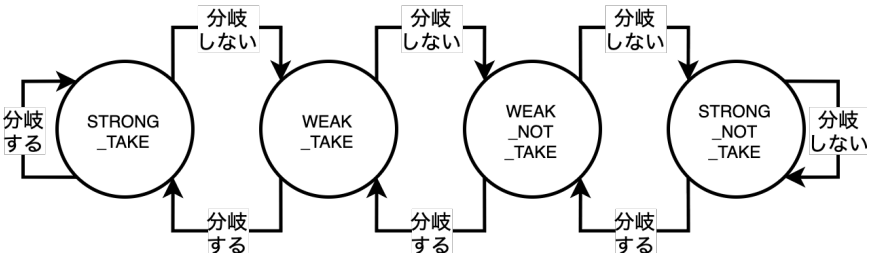


図 3: 2bit 予測器: 状態遷移図

表 6: 分岐予測実装前後のプログラム実行クロックサイクル数

ベンチマークプログラム	クロックサイクル数 (分岐予測実装前)	クロックサイクル数 (分岐予測実装後)
stringsearch	10594	6966
bitcnts	56040	44680
dijkstra	4079473	3048011

アドレスを ALU で計算された分岐先アドレスに更新する.

4.2.4 分岐予測の評価と論理合成

エントリー数が 32 個の参照テーブルをもつ分岐予測器を搭載したプロセッサを, MiBench ベンチマークプログラムで実行クロックサイクル数とミス率を測定した後に, 論理合成を行った. 実行クロックサイクル数の結果を表 6 に, ミス率の結果を表 7 に, 論理合成の結果を表 8 に示す.

参照テーブルのエントリー数を 1 から 4096 まで変化させて同じ項目で評価を行ったところ, エントリー数が増えるほど予測失敗率が減っていくことがわかった. しかしながら, 33 個以上のエントリー数の参照テーブルを持つプロセッサの論理合成にかかる時間が長かったため, 採用しなかった. よって, 32 個以下のエントリー数の参照テーブルを持つプロセッサの中で, 予測失敗率が最も小さいエントリー数が 32 の参照テーブルを選択した.

4.3 クリティカルパスの短縮

論理合成の結果から, クリティカルパスが EX ステージにあることが分かった. そのクリティカルパスは図 4a で示す.

上記のクリティカルパスを短縮するために, 以下のことを行った.

1. ID ステージでの ALU のオペランドの生成 (4.3.1 小節)
2. パイプラインレジスタでのパイプラインフラッシュ (4.3.2 小節)

この 2 つの方法により, クリティカルパスが図 4b のように短くなった.

4.3.1 ID ステージでの ALU のオペランドの生成

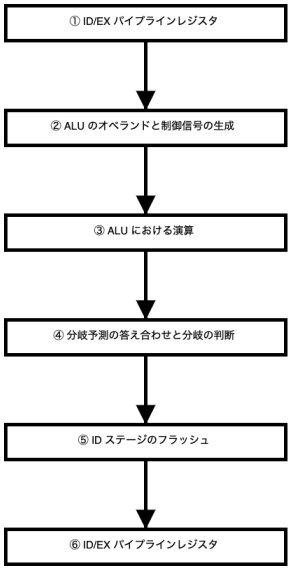
EX ステージで演算を行う前に, 以下のオペランドと制御信号を用意する必要がある.

1. オペランド
   (a) ALU の演算のオペランド
   (b) 比較専用 ALU の演算のオペランド
2. 制御信号
   (a) ALU に対して演算の種類を指定する制御信号
   (b) 比較専用 ALU に対して比較の種類を指定する制御信号

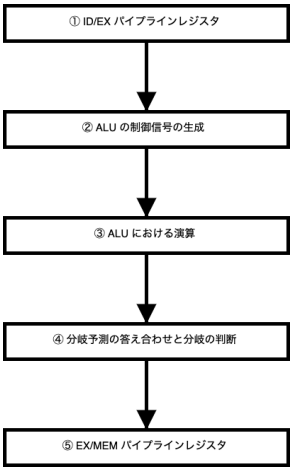
論理合成の結果から, 演算のオペランド (1a と 1b) の生成に必要な時間が, クリティカルパスの高い割合を占めることがわかった. これを改善するために, 演算のオペランドの生成回路を ID ステージに移動させた.

表 7: 分岐予測の予測失敗率

ベンチマークプログラム	分岐予測対象命令数	予測失敗命令数	予測失敗率 [%]
stringsearch	2113	131	6.20
bitcnts	9930	690	6.95
dijkstra	869932	12886	1.48



(a) 改善前



(b) 改善後

図 4: 改善前後のクリティカルパス

その結果, 図 4a の 2 の「ALU のオペランドと制御信号の生成」が 4b の 2 の「ALU の制御信号の生成」になり, プロセッサの最小動作クロック周期を 9.00ns から 8.00ns まで減らすことができた. それでも, プログラムの実行時間がまだ分岐予測を導入する前より長いため, 次の改善を実施した.

4.3.2 パイプラインレジスタでのパイプラインフラッシュ

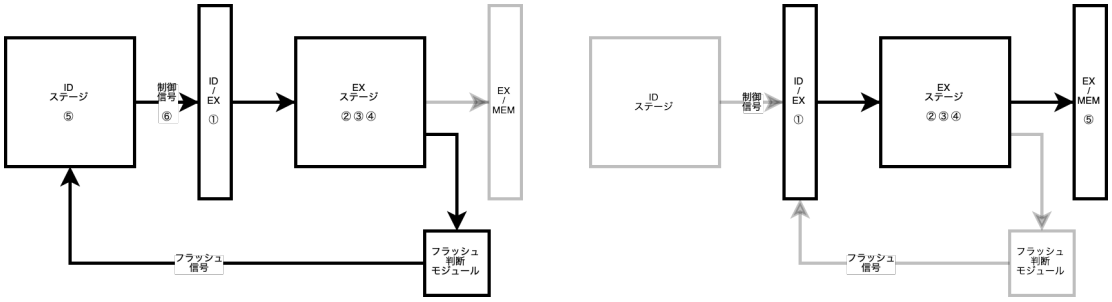
実装していたパイプラインフラッシュの回路では, プログラムの実行の中で分岐が起きた時に, パイプラインレジスタではなく, IF ステージと ID ステージのフラッシュをステージ内で行っていた (図 5a). この実装方法により, ID ステージにおいて, データメモリと汎用レジスタに対する書き込み信号を生成するタイミングは分岐結果が出た後のタイミングになる. その結果, 分岐結果が分かるまでに ID ステージの信号の生成ができなく, 無駄な時間が生じてしまう.

これを改善するために, パイプラインのフラッシュをステージ内ではなく, パイプラインレジスタで行うようにした (図 5b). 改善後の回路では, たとえば ID ステージで「データメモリに対して書き込む」信号が生成されても, EX ステージの分岐結果が「分岐する」なら, その信号が ID/EX パイプラインレジスタに保存される前に, 「データメモリに対して書き込まない」へと無効化される.

この改善によってプロセッサの最小動作クロック周期を 8.00ns から 5.00ns まで減らすことができた. 改善によって得られたクロック周期は分岐予測導入前の 6.00ns よりも短くなったため, 分岐予測機能を今回のプロセッサに導入することにした.

4.3.3 クリティカルパスの短縮後の論理合成

クリティカルパスの短縮を行った後の性能評価の結果を表 8 に示す. クリティカルパスの短縮により, 分岐予測の実装によって 9ns までに増えたクロック周期を 5ns へと減少させることができた.



(a) 改善前　　(b) 改善後

図 5: パイプラインフラッシュ処理の回路とクリティカルパス

表 8: 性能改善前後の論理合成の結果

プロセッサ	最小動作クロック周期 [ns]	面積 [μm^2]	消費電力 [mW]
分岐予測実装前	6.00	357534.7228	7.5732
分岐予測実装後	9.00	936675.8334	10.6318
クリティカルパス短縮後	5.00	764805.1213	15.3567

4.3.4　さらに改善できる点

1 つのステージ内の処理負担が減少すれば, クリティカルパスが短くなることが期待でき, それによってプロセッサのクロック周期をさらに短くできると考えられる.

5　まとめ

今回のプロセッサ設計演習では, 5 段のパイプライン処理, 例外処理と分岐予測機能をもつプロセッサを設計した. テストプログラムを用いてプロセッサの動作確認を行い, プロセッサが正しく動作することを確認した. そして, MiBench ベンチマークプログラムを用いてプロセッサの性能評価を行い, 論理合成も行った. 実際にプロセッサを設計するにあたり, 細かいところまで考慮しなければ, 思い通りの動作が出ないことを痛感した. この演習を通じて, 講義で習ったプロセッサの動作原理をより深く理解することができたと思う. また, 困難なことをやろうとする時に, それを複数個の実現可能なタスクに分解して行っていくことで, モチベーションを保ちながら続けられることを学ぶことができた.

謝辞

プロセッサの設計環境 `woodblock` を維持してくれている先輩と教員たちに, テストプログラムの用意と質問への対応をしてくれた先輩に, 一緒に悩んで考えてくれた同期に感謝を伝えたい.

参考文献

[1] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 3–14, 2001.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Katey Birtcher, 6 edition, 2019.

[3] David Patterson and Andrew Waterman. RISC-V 原典オープン・アーキテクチャのススメ. 安達功, 2018.

[4] Andrew Waterman, Krste Asanovic, and SiFive Inc. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, 2019.

付 録 A　プロセッサのブロック図

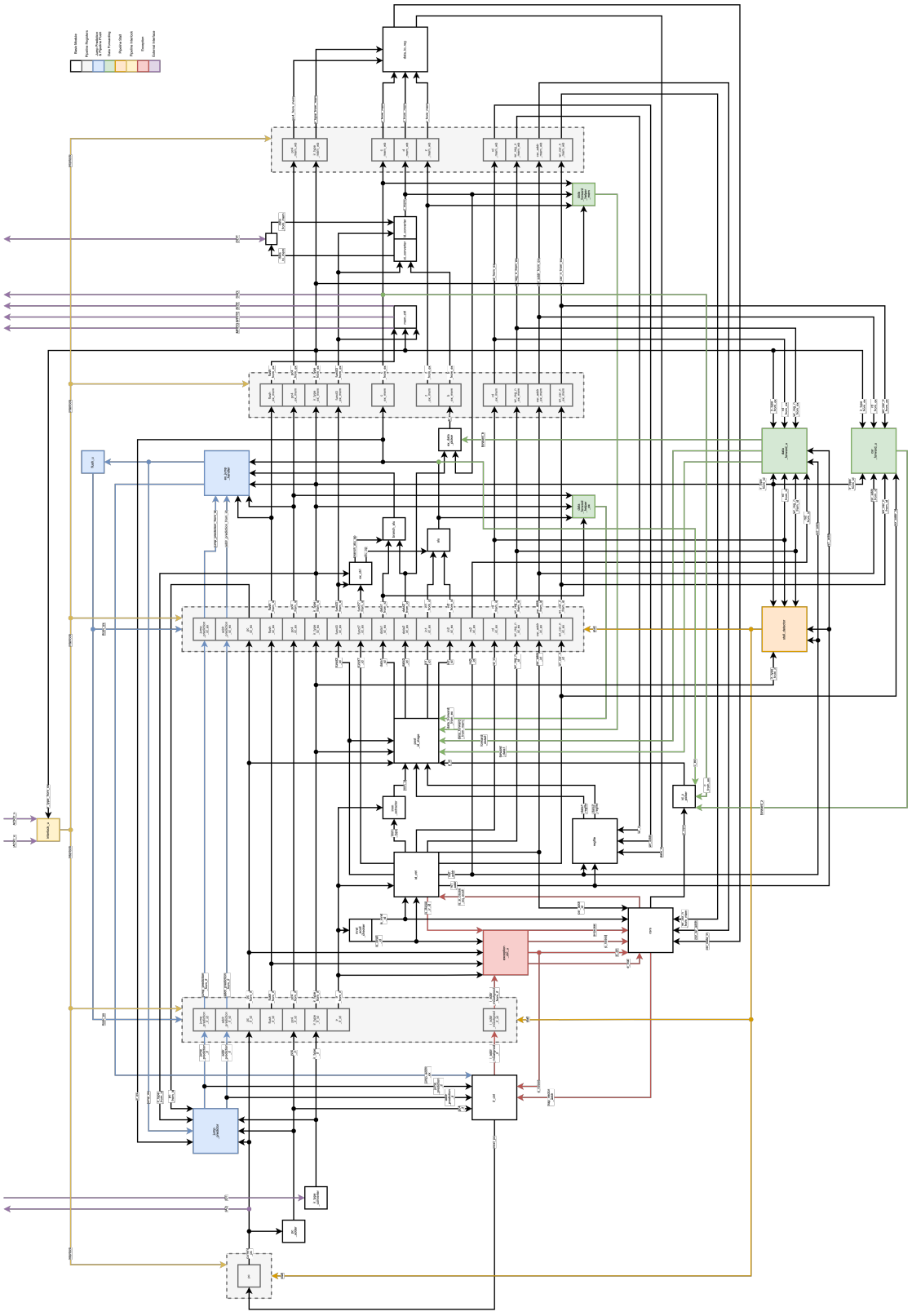


図 6: プロセッサのブロック図

付 録 B　サポートしている命令セット

表 9: 命令セット

命令	内容	形式
lui	load upper immediate	U
auipc	add upper immediate to pc	U
jal	jump and link	J
jalr	jump and link register	J
beq	branch equal	B
bne	branch not equal	B
blt	branch less than	B
bge	branch greater than or equal	B
bltu	branch less than unsigned	B
bgeu	branch greater than or equal unsigned	B
lb	load byte	I
lh	load halfword	I
lw	load word	I
lbu	load byte unsigned	I
lhu	load halfword unsigned	I
sb	store byte	S
sh	store halfword	S
sw	store word	S
addi	add immediate	I
slti	set less than immediate	I
sltiu	set less than immediate unsigned	I
xori	exclusive or immediate	I
ori	or immediate	I
andi	and immediate	I
slli	shift left logical immediate	I
srli	shift right logical immediate	I
srai	shift right arithmetic immediate	I

命令	内容	形式
add	add	R
sub	sub	R
sll	shift left logical	R
slt	set less than	R
sltu	set less than unsigned	R
xor	exclusive or	R
srl	shift right logical	R
sra	shift right arithmetic	R
or	or	R
and	and	R
ecall	environment call	I
csrrw	csr read and write	I
csrrs	csr read and set	I
csrrc	csr read and clear	I
csrrwi	csr read and write immediate	I
csrrsi	csr read and set immediate	I
csrrci	csr read and clear immediate	I
mret	machine-mode exception return	R

付 録 C　機能検証と性能評価用プログラム

表 10: 機能検証用プログラム

テストプログラム	言語	プログラム内容
load	アセンブリ	ロード命令の動作検証
store	アセンブリ	ストア命令の動作検証
p2	アセンブリ	演算と分岐命令の動作検証
trap	アセンブリ	`ECALL` 命令の動作検証
hello	C	Hello World! をコンソールに表示するプログラム
napier	C	Napier's Constant, e の値を 64 桁の精度で計算するプログラム
pi	C	pi の値を 64 桁の精度で計算するプログラム
prime	C	2 を含め, 40 個の素数を昇順に見つけるプログラム
bubblesort	C	100 個の整数を Bubble Sort でソートするプログラム
insertsort	C	100 個の整数を Insert Sort でソートするプログラム
quicksort	C	100 個の整数を Quick Sort でソートするプログラム

表 11: 性能評価用プログラム

テストプログラム	言語	プログラム内容
bitcnts	C	7 つの方法で与えられた数字の ビット数を求めるプログラム
stringsearch	C	ケース・インセンシティブ方式で 文字の検索するプログラム
dijkstra	C	ダイクストラアルゴリズムで与えられた グラフのノード間の最短距離を求めるプログラム