プロセッサ設計演習

ウン クアン イー

2021.06.22

1　はじめに

現代のマイクロプロセッサの動作原理を理解することを目的とし, 基礎である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. 更に, プロセッサの最大遅延時間と面積を削減するために, 分岐予測を導入して, パイプラインの各ステージの役割分担を見直した. 分岐予測の導入によって, テストプログラムの実行クロックサイクル数が (数字を入れる) 減少した. パイプラインの各ステージの役割分担の見直しによって, ボトルネックとなるステージの処理が他のステージに行われるようにすることで, 最大遅延時間が (数字を入れる) 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について説明し, 第 4 章で, プロセッサの性能評価と論理合成の結果を示す. 第 5 章で, プロセッサの性能改善方法として, 分岐予測とクリティカルパスの短縮について述べる. 最後に, 第 6 章でまとめを行う.

2　プロセッサの仕様

2.1　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ことを示している.

図 1 にある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 32 [bit] のアクセスアドレスパス.
- IDT (Instruction DaTa Bus)
  命令メモリからの 32 [bit] 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

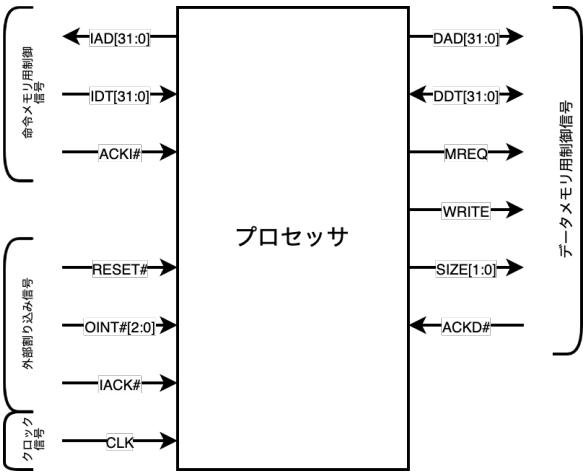

図 1　プロセッサの外部インタフェース

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 32 [bit] のアクセスアドレスパス.
- DDT (Data DaTa bu)
  データメモリからの 32 [bit] 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. ただし, 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 1 [byte] ごとに振り分けられ, 1 つのアドレスに 32 [bit] 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 出力信号への出力を行わない.

2.2 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は表 1 に示している. 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.

2.3 例外・割り込み処理

このプロセッサが対応している例外・割り込み処理を優先順位の高い順に以下に示す.

1. リセット
2. 不正命令
3. 命令アクセス・ミスアライメント
4. ECALL 命令

2.4 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, 実行される. 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.

2.5 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ている. 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. この章では,

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

表 1　命令セット

データハザードの 1 種である RAW (Read After Write) ハザードについて説明してから, 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に, 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場合を記述する.

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場合.
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場合.

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.

2.5.1 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. この時に, 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って, 上記の RAW ハザードが発生する場合の中で, 命令 m がロード命令以外の場合の対処ができる. データフォワーディングで解決できなかった場合は, パイプラインストールを用いる.

2.5.2 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用意できるまでに, それ以降の命令を待たせる.

パイプラインストールによって, データフォワーディングが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.

RAW ハザードが発生する状況とその解決法を表 2 にまとめる.

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/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/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

表 2　RAW ハザードに関わる命令とその解決法

2.6 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. この章では, 分岐命令による制御ハザードについて説明してから, 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, その命令の分岐結果によって次に実行する命令が決まる. 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, 分岐命令の分岐結果が EX ステージにならないと判明されない. それに, 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. そこで, 分岐結果が「分岐しない」ならば, 問題なく命令の実行を続くことができる. しかし, 分岐結果が「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ったら, プログラムの実行結果が正しくなくなる.

2.6.1 パイプラインストール

パイプライン処理で, 分岐命令がある時に, IF ステージと ID ステージにある命令を実行しない方法の 1 つとしてパイプラインストールがある. 分岐命令の分岐結果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, 分岐結果が分かってから, パイプラインストールを解除し, 分岐結果を元に選んだ次の命令をフェッチし, 分岐命令以降の命令の実行を再開する. しかし, パイプラインストールを採用すれば, 分岐命令がある度に, 2 つのクロックサイクルが無駄になってしまう. パイプラインストールよりクロックサイクルの無駄が少ない方法があるので, パイプラインストールを採用しなかった.

2.6.2 パイプラインフラッシュ

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. 例えば, 分岐命令の分岐結果が「分岐する」であっても, IF ステージと ID ステージにある命令の実行を続ける. ただし, それらの命令がメモリと汎用レジスタに対する書き込みを無効化した状態の元で, 実行を続行する. その結果, 分岐結果が「分岐しない」場合では, クロックサイクルの無駄なく, 分岐命令に続く命令がそのまま実行されていく. しかし, 分岐結果が「分岐する」場合では, 2 つのクロックサイクルが無駄になってしまう. パイプラインストールと比べたら, 分岐によるクロックサイクル数の無駄が少ないため, 今回の設計にパイプラインフラッシュを採用した.

2.7 プロセッサの名前

3 プロセッサの機能検証

今回設計したプロセッサの機能検証を以下のプログラムを用いて行った.

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

機能検証は, Verilog-HDL で記述したプロセッサに対して, 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. 上記の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.

4 プロセッサの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, と消費電力という面で評価する.

4.1 評価方法

MiBench ベンチマークプログラムを用いて, プログラム実行のクロックサイクル数を求めた. MiBench の中に, `stringsearch`, `bitcnts`, `dijkstra` といったプログラムが含まれている. 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, と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 $10[ns]$ と設定して論理合成を行い, 論理合成の結果にある 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 $9[ns], 8[ns], \ldots$ のように $1[ns]$ ずつ下げ, slack が負になるタイミング制約を見つける.

例えば, タイミング制約を $5[ns]$ 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 $6[ns]$ になる.

4.2 評価結果

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, と消費電力の測定結果を表 4 に示す.

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

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

最小動作クロック周期 $[ns]$	面積 $[\mu m^2]$	消費電力 $[mW]$
6	357534.722794	7.5732

表 4　論理合成の結果

5　プロセッサの性能改善方法と評価

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. プロセッサによるプログラムの実行時間は式 1 で求められる. 式 1 より, プログラムの実行クロックサイクル数, または, プログラムの動作クロック周期, または, 両方も減らしたら, プログラムの実行時間が短くなることが分かる.

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

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, 分岐予測を導入した. 分岐予測の詳細については, 5.1 章で述べる.

分岐予測を実装した後, プログラムの実行クロックサイクル数を, 平均的に 0.794 倍で減らすことができたが, プロセッサの最小動作クロック周期が $6[ns]$ から $9[ns]$ までに増加してしまった. 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, 論理合成の結果を元に, 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, 5.2 章で述べる.

5.1　分岐予測

5.2　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が以下の通りである.

ID/EX パイプラインレジスタ
$\rightarrow ALU$ の入力の用意
$\rightarrow ALU$ における演算
$\rightarrow$分岐の判断
$\rightarrow$パイプラインフラッシュの判断
$\rightarrow ID/EX$ パイプラインレジスタ

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. ALU の入力の用意を ID ステージで行う (5.2.1 章)
2. パイプラインフラッシュのタイミングの後回し (5.2.2 章)

5.2.1　ALU の入力の用意を ID ステージで行う

EX ステージで演算を行う前に, 以下の入力を用意する必要がある.

1. ALU に対して演算の種類を指定する制御信号
2. ALU の演算対象となる 2 つの $32[bit]$ の信号
3. 比較専用 ALU に対して比較の種類を指定する制御信号
4. 比較専用 ALU の比較対象となる 2 つの $32[bit]$ の信号

倫理合成の結果から, 信号が生成されるまでに必要な時間からみた時に, 信号 2 と 信号 4 がボトルネックになっていた. これを改善するために, 信号 2 と信号 4 の生成回路を ID ステージに移動させた. その結果, プロ

セッサの最小動作クロック周期を 9[ns] から 8[ns] 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の性能が，分岐予測を導入する前のに劣っているため，次の改善を実施した．

5.2.2 パイプラインフラッシュのタイミングの後回し

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．これにより，EX ステージに分岐命令がある場合，ID ステージの命令がデータメモリと汎用レジスタに対する書き込み信号の生成タイミングは分岐結果が出た後のタイミングになってしまう．分岐結果が分かるまでに ID ステージは待たないといけないため，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．改善後の回路では，例えば 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」となったら，その信号が ID と EX 間の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」と無効化される．この改善によってクリティカルパスが短くなり，プロセッサの最小動作クロック周期を 8[ns] から 5[ns] 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[ns] よりも短くなったので，分岐予測機能を今回のプロセッサの設計に導入してもいいと判断した．

5.3 性能改善後の性能と論理合成

分岐予測を実装した後と，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 5 と表 6 にまとめた．分岐予測の実装によるクロックサイクル数を実装前のと比較したら，平均的に 79.4% 減少したことがわかった．また，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 9[ns] までに増えたクロック周期を 5[ns] へと減少させたことができた．

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

表 5　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

プロセッサ	最小動作クロック周期 [ns]	面積 [μ m^2]	消費電力 [mW]
分岐予測実装前	6	357534.7228	7.5732
分岐予測実装後	9	936675.8334	10.6318
クリティカルパス短縮後	5	764805.1213	15.3567

表 6　性能改善前後の論理合成の結果

6 まとめ

まとめの内容

参考文献

参考文献の内容