プロセッサ設計

2023 年 8 月 23 日

はじめに

コンピュータアーキテクチャに関連する研究を行うために、プロセッサの原理と設計プロセスを理解する必要があります。したがって、Verilog-HDL ハードウェア記述言語を使用して、RISC-V RV32I 命令セットのプロセッサを設計した。

1　プロセッサの仕様

1.1　命令セット

プロセッサの命令セットは、RISC-V の基本命令セット RV32I を選択した。ここで、「RV」は RISC-V を表し、「32」はこのアーキテクチャが 32 ビット幅を採用していることを示し、「I」は整数命令セットを表す。RISC-V はオープンな ISA であり、個人や組織はライセンス料を支払うことなく自由に使用することができる。さらに、その命令フォーマットはシンプルで統一されており、これによりプロセッサハードウェアの単純化が実現し、速度と消費電力の面で優れている。また、RISC-V は、そのコアの基本命令セットを拡張し、浮動小数点演算、乗算、アトミック操作などの機能を追加することができる。

RISC-V 命令セットは、R、I、S、B、U、J の 6 つの形式に分かれており、各命令セットの具体的な形式は Fig.1 に示されている。

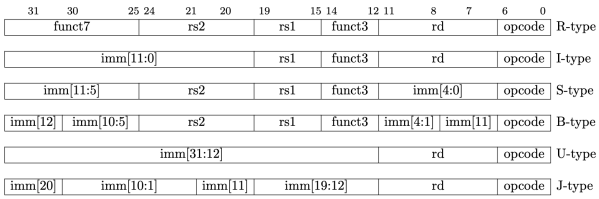

Figure 1: 命令形式

1.2　単一サイクルデータパス

単一サイクルのマイクロアーキテクチャは、1 サイクル内で命令のフェッチ (IF)、デコード (ID)、実行 (EX)、メモリアクセス (MEM)、および結果の書き戻し (WB) を行うもので、そのサイクルのサイズは最も遅い命令の時間に制約される。また、

命令の各フェーズを実行する際に、CPU の一部のコンポーネントがアイドル状態になる可能性があり、これによりリソースが非効率的に利用される。しかし、単一サイクルの CPU の設計は通常、よりシンプルであるという利点がある。単一サイクルのアーキテクチャの命令の実行順序は Fig.2 に示されている。

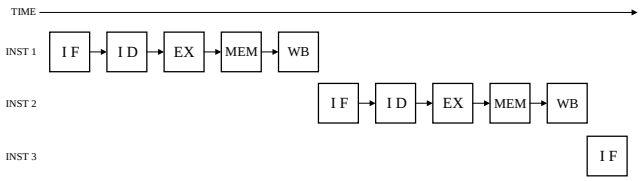

Figure 2: 単一サイクルの CPU の実行順序

1.3 パイプラインデータパス

パイプラインのアーキテクチャは、1 つの命令の複数の実行段階をそれぞれ異なるサイクルで実行するもので、異なる命令が異なるクロックサイクルで異なるハードウェアを使用するため、これにより CPU のハードウェアリソースをより効率的に利用することができる。このような構造により、CPU は高いスループットを持つことができ、また、各サイクルで命令の一部しか完了する必要がないため、CPU の動作周波数を向上させることができる。しかし、このような設計は CPU のハードウェアの複雑さを増加させるという側面もある。パイプラインのアーキテクチャの命令の実行順序は Fig.3 に示されている。

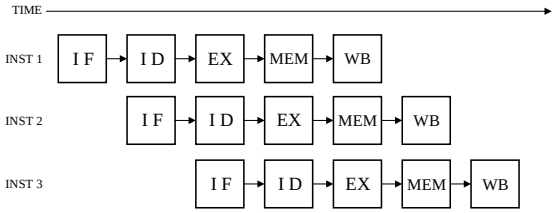

Figure 3: 5 段パイプラインの CPU の実行順序

1.3.1 IF (Instruction Fetch)

この段階で、CPU はプログラムメモリから実行する指令を取得する。PC の値に基づいて命令メモリから命令を読み取り、プログラムカウンタを更新して次の命令のアドレスを指す。

1.3.2 ID (Instruction Decode)

この段階で、CPU は取得した指令をデコードし、実行する操作と使用するオペランドを特定する。そして、一部の制御信号を生成する。さらに、指令がレジスタの値を使用する必要がある場合、レジスタファイルから読み取る。

1.3.3 EX (Execution)

この段階で、CPU は指令の操作を実行する。算術および論理指令の場合、ALU は計算を実行する。load および store 指令の場合、有効アドレスを計算する。分岐指令の場合、分岐を取るかどうかを判断し、ターゲットアドレスを計算する。

1.3.4 MEM (Memory Access)

この段階で、プロセッサが load 指令を実行している場合、計算されたアドレスからデータを読み取る。store 指令を実行している場合、計算されたアドレスにデータを書き込む。

1.3.5 WB (Write Back)

この段階で、プロセッサは EX 段階または MEM 段階で生成された結果をターゲットレジスタに書き戻す。

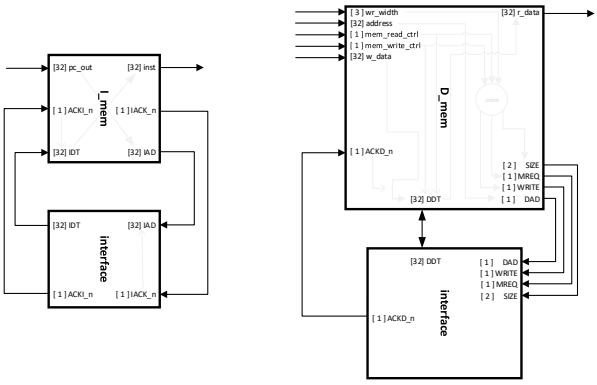

Figure 4: Imem と Dmem インターフェース

1.4 インターフェース

設計する際、最初は top test ファイルを手に入れられなかったため、一部の制御信号は[3]に基づいて設計された。その結果、一部の信号の定義は top test ファイルと差異がある。そこで、Imm と Dmm の信号変換モジュールを設計した。そのモジュールは Fig.4 に示されている。

2 工夫した点

2.1 一部の分岐判定の前倒し

パイプラインのアーキテクチャの CPU では、B タイプの命令の判断は通常 EX 段階で実行される。どのような分岐予測方法を使用しても、予測失敗が発生した場合は ID 段階と IF 段階の命令をフラッシュする必要があり、これには 2 つのサイクルが無駄になる。しかし、分岐命令の beq と bne は、判断プロセスが比較的簡単で ALU を必要としないため、beq と bne のデータ比較を ID 段階に前置することができる。これにより、これら 2 つの命令については、分岐予測失敗が発生した場合、1 つのサイクルの命令をフラッシュするだけでよい。また、load-use(beq/bne) タイプのデータハザード後の 2.3 節で説明する。

2.2 データハザード

プログラムを実行する際、ある命令が前の命令の結果に依存するケースがある。パイプライン構造では各段階で異なる命令を実行するため、問題が起こる可能性がある。 データハザードには、データフォワーディングやデータの書き込みおよび読み取りのタイミングを変更することで解決でき

る 4 つのケースがある。 命令が EX フェーズで EX フェーズでの別の命令の実行結果を使用する場合、EX/MEM レジスタのデータをフォワードする。命令が EX フェーズで MEM フェーズでの別の命令の実行結果を使用する場合、MEM/WB レジスタのデータをフォワードする。命令が EX フェーズで、WB フェーズでの別の命令の実行結果を使用する場合、クロックの後半でデータを書き戻すため、データのフォワードは不要である。これら 3 つのケースは Fig.5 に示されている。

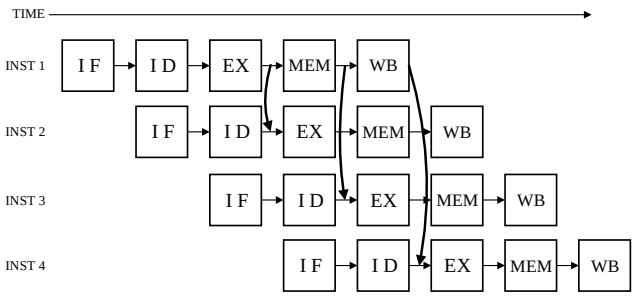

Figure 5: 以上の 3 つのケースのフォワード

load-use(store) 型のデータハザードが発生した場合、load 命令の rd が store 命令の rs1 とは異なり、rs2 と同じであれば、ストールする必要はなく、MEM/WB レジスタのデータを MEM フェーズにフォワードするだけでよい。この状況は Fig.6 に示されている。

他の load-use の場合、ストールが必要である。発生する条件は以下に示す。

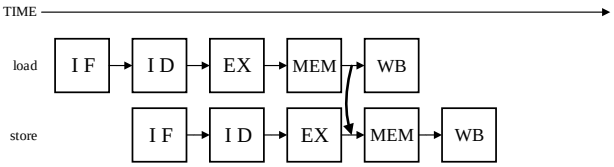

Figure 6: load-use(store) 型のフォワード

1. ID/EX.MemRead and ID/EX.Regwrite and (ID.MenWrite and ((ID/EX.rd=ID.rs1) or (ID/EX.rd=ID.rs2)))

2. IF/ID.isJALR and (ID/EX.MemRead and ID/EX.Regwrite and ((EX/MEM.rd=ID.rs1)))

3. (not LoadBneq) and ID.Bneq and(EX/MEM.MemRead and EX/MEM.Regwrite and((EX/MEM.rd=ID.rs1) or (EX/MEM.rd=ID.rs2)))

4. (not LoadBneq) and ID.Bneq and(ID/EX.Regwrite and ((ID/EX.rd=ID.rs1) or (ID/EX.rd=ID.rs2)))

ストールの具体的な実装方法は、次のサイクルで PC レジスタと IF/ID レジスタの出力を変更せずに保持し、ID/EX レジスタをクリアすることである。

2.3 制御ハザード

設計では分岐が発生しないと仮定されている。したがって、ID ステージでジャンプが必要と判断された場合、現時点で IF ステージにある命令をフラッシュする必要がある。EX ステージでジャンプが必要と判断された場合、現時点で IF ステージと ID ステージにある命令をフラッシュする必要がある。フラッシュの具体的な実装方法は、IF/ID レジスタまたは ID/EX レジスタをクリアすることである。

一般的な load-use 型のデータハザードについて、CPU は ID ステージと EX ステージの信号を検出して stall が必要かどうか判断するため、use のステージは通常 EX ステージであるため、データフォワーディングによってデータを取得するためには 1 サイクルの stall が必要である。ただし、load-use(beq/bne) 型のデータハザードの場合、ID ステージでデータを取得するには 2 サイクルの stall が必要で、これは EX ステージで判断する場合に無駄になるサイクル数と同じであるため、この種の bne または beq も EX ステージで判断される。下の表では、Table 1 は stall を 2 回行う実行順序で、Table 2 は EX ステージに配置される実行順序である。

Table 1: 実行順番の対比

time	IF	ID	EX	MEM	WB
1	$inst_1$	**beq**	**load**		
2	$inst_1$	**beq**	~~beq~~	**load**	
3	$inst_1$	**beq**	~~beq~~	~~beq~~	**load**
4	$\mathbf{inst_{tgt}}$	~~$inst_T$~~	**beq**	~~beq~~	~~beq~~

time	IF	ID	EX	MEM	WB
1	$inst_1$	**beq**	**load**		
2	$inst_1$	**beq**	~~beq~~	**load**	
3	$inst_2$	$inst_1$	**beq**	~~beq~~	**load**
4	$\mathbf{inst_{tgt}}$	~~$inst_2$~~	~~$inst_T$~~	**beq**	~~beq~~

3 性能評価

実行されるテスト・プログラムは、Mibench フォルダ内の bitcnts、dijkstra、stringsearch で、それぞれ test、small、large の 3 つのバージョンがある。コンパイル時に OPTIMIZE レベルを 3 に設定し、プログラムの実行に必要なサイクル数を Table 2 に示す。

Table 2: クロックサイクル数 (O=3)

	test	small	large
bitcnts	53158	38128163	570943108
dijkstra	4084141	37445372	189157568
stringsearch	10713	92120	2112971

論理合成の結果を Table 3 に示す。

Table 3: 論理合成結果

最大遅延時間 ns	消費電力 mW	面積 μm^2
4.56	13.7230	277570

4 まとめ

今回の RISC-V プロセッサ設計演習では、まず、単一サイクル構造の RISC-V プロセッサを設計した。その後、単一サイクルのプロセッサに基づいて各機能を分割し、新しいモジュールを追加した。最終的には、5 段階のパイプライン構造のプロセッサを設計した。演習を通じて、プロセッサの構造や原理についての理解を深めた。

参考文献

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 6th edition, 2017.

[2] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas.* Strawberry Canyon, 1st edition, 2017.

[3] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, 6th edition, 2020.