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

ムハマド ナウファル
九州大学 工学部 電気情報工学科 4 年
2020 年 7 月 05 日

1 はじめに

プロセッサ設計を通して verilog-HDL というハードウエア記述言語を用いて 5 段パイプラインの例外なし RISCV-32I プロセッサを設計し、基礎的なプロセッサの構造を理解することを目指した。設計されたプロセッサはいくつかのテストプログラムで検証を行い、正常な動作を確かめてからベンチマークプログラムで性能を調べた。その後、論理合成を行った。

本稿では，2 章でプロセッサの仕様について述べる。3 章ではプロセッサの構成を詳しく、4 章では検証結果および論理合成の結果について述べ、5 章では論理合成結果を基に改善したことについて、6 章ではまた改善できるものを述べる。7 章ではまとめを述べる。

2 プロセッサの仕様

2.1 プロセッサ名

この演習で設計したプロセッサの名前を,「grapy 」と名付けた。その由来は、最近ぶどうにハマってしまって、「grape 」だと独特はなく、「grapy 」にした。それぐらいの気持ちでこの設計演習に取り組んだということを伝えたい。

2.2 命令セット

本プロセッサがサポートする RISCV 32I の命令を表 1 に示す。命令は全て 32bit の幅長であり、それぞれある命令形式に従っている。命令形式には、R 形式、I 形式、S 形式、B 形式の 4 種類がある。また命令形式は図 1 に示してある。

31 27	26 25	24 20	19 15	14 12	11 7	6 0	命令
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

図 1: 命令形式

2.3 外部インタフェース

本プロセッサは 5 段パイプライン処理を導入することため、命令フェッチとデータアクセスの資源競合存在する。このような状態を避けるため、ハードウェア的なアプローチをとり、命令用メモリおよびデータ用メモリに対して同時にアクセス可能である。また，命令およびデータを扱うとき、幅長は全て 32bit で，メモリはバイト単位で格納されている。図 2 に外部インターフェイスとプロセッサの接続を示す。ここで，信号名の末尾に付いている # は，信号線が負論理 (Low でアクティブ) であることを意味する。

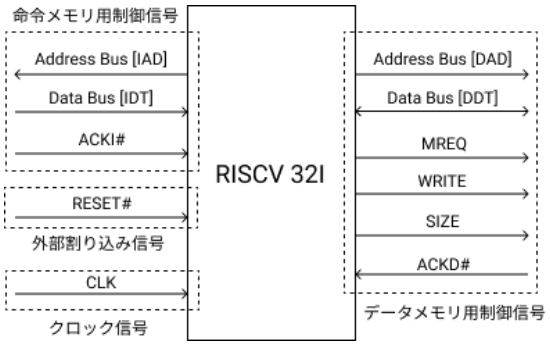


図 2: プロセッサの外部インタフェース

表 1: 命令セット

命令	形式	内容
lui	U	Load Upper Immediate
auipc	U	Add Upper Immediate to PC
jal	J	Jump And Link
jalr	J	Jump And Link Register
beq	B	Branch EQual
bne	B	Branch Not Equal
blt	B	Branch Less Than
bge	B	Branch Greater than or Equal
bltu	B	Branch Less Than Unsigned
bgeu	B	Branch Greater than or Equal Unsigned
lb	I	Load Byte
lh	I	Load Halfword
lw	I	Load Word
lbu	I	Load Byte Unsigned
lhu	I	Load Halfword Unsigned
sb	S	Store Byte
sh	S	Store Halfword
sw	S	Store Word
addi	I	ADD Immediate
slti	I	Set Less Than Immediate
sltiu	I	Set Less Than Immediate Unsigned
xori	I	eXclusive OR Immediate
ori	I	OR Immediate
andi	I	AND Immediate
slli	I	Shift Left Logical Immediate
srli	I	Shift Right Logical Immediate
srai	I	Shift Right Arithmetic Immediate

命令	形式	内容
add	R	ADD
sub	R	SUB
sll	R	Shift Left Logical
slt	R	Set Less Than
sltu	R	Set Less Than Unsigned
xor	R	eXclusive OR
srl	R	Shift Right Logical
sra	R	Shift Right Arithmetic
or	R	OR
and	R	AND

3 プロセッサの構成

3.1 パイプライン処理

本プロセッサでは、逐次実行で実装するのあれば、かなりのクロック数を無駄にするため、複数の命令を同時に並列的に実行するためには 5 段パイプライン処理を導入し、それらの 5 ステージの詳細は以下のように構成される。

- IF ステージ
  取得する命令アドレスを計算し，命令メモリから次々と実行すべき命令をフェッチする。
- ID ステージ
  フェッチされた命令をデコードし，必要とする制御信号を解釈する。また、デコードされた命令種類に対し、即値の拡張を行う。ここで、ストール検出「詳細は 3.2.2」とデータ依存検出「詳細は 3.2」も行う。
- EX ステージ
  命令の種類から，行うべき演算を実行するここで、ID ステージで解釈された命令の種類が分岐命令であり、分岐条件が満たしていれば、分岐先のアドレスも計算する「詳細は 5.2」。
- MEM ステージ
  主にロード命令またはストア命令を扱うステージである。メモリアドレスを指定し，データメモリからデータをロードもしくはデータを

ストアする。ただし、ロードされたデータを命令の種類により拡張する。

- WB ステージ
  レジスタへの書き込みを行う。

3.2 データハザードとその解決方法

本プロセッサは 5 段パイプラインで実装されているため、いくつかの問題が発生する。パイプライン化をすることで、EX ステージで実行している命令と以前に実行していた命令にデータの依存関係があり得る。つまり、データハザードが生じ、逐次に実行したときとの最終結果が変わり、結果の保証ができなくなる。本プロセッサでは、ハザードは主に２つの場合があり、LOAD 命令の場合と LOAD 命令でない場合である。詳細は、3.2.1 にある。ハザード検出の原理は、３つのパイプライン、IF/ID パイプライン、ID/EX パイプライン、EX/MEM パイプラインを基に検出する。それぞれのパイプラインには現在各ステージで実行している命令を格納しているため、その命令から関係するレジスタを取り出し、検出を行う。従って、パイプラインストールおよびデータフォワーディングという概念を導入する必要がある。ここで、注意すべきところは、ハザード検出には２つ同様なユニットからできているため、書き込みのレジスタは重なる可能性がある。よって、場合分けする必要があり、そのどれかを１つしか選べない状態にする。

3.2.1 データフォワーディング

フォワーディングとは、EX ステージでの命令と前の命令にハザードが生じたときに、前命令の EX ステージからの計算結果を EX ステージにフォワードする。ただし、LOAD 命令には EX ステージではなく、MEM ステージでデータメモリから取り出した値を EX ステージで実行している命令にフォワードすること注意すべきである。本プロセッサで実装したデータフォワーディングは図 4 に示す。

3.2.2 パイプラインストール

パイプラインストールが必要になった理由は、フォワーディングだけでは解決できないハザードに対抗

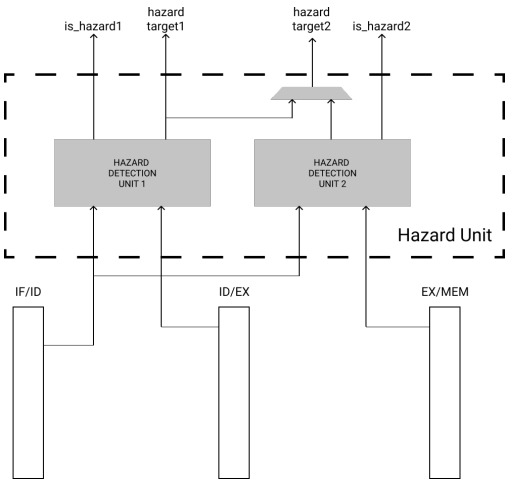


図 3: データフォワーディング

するためである。例えば、EX ステージでの命令は前の命令にデータ依存があり、前の命令が LOAD 命令であるとき、EX ステージではなく、MEM ステージからのフォワーディングになる。つまり、１段を空けないといけない状態になっている。本プロセッサで実装したストールは図 5 に示す。ストール信号を受ける IF ステージは現在の PC の値に対し、4 を引き、前の命令をフェッチすることになる。IF/ID パイプラインでは、その信号を受けると、プロセッサの状態を影響するような信号をゼロの信号を出力する。ただし、レジスタファイルに対する書き込みは負論理の信号なので、１の信号を出力ことになる。

ただし、こうのような方法では１つの問題があり、その解決方法も含め、6.1 に詳しく述べる。

3.3 制御ハザードその解決方法

ここでのプロセッサではを考え、制御ハザードとは分岐命令またはジャンプ命令にしか起こらないハザードである。ジャンプ命令は ID ステージでわかるため、次から実行する命令はすでにフェッチされているため、ジャンプした後も、その命令は ID ステージで実行することになる。よって、実行結果に影響を及ばさないためには、パイプラインフラッシュという概念を導入する。

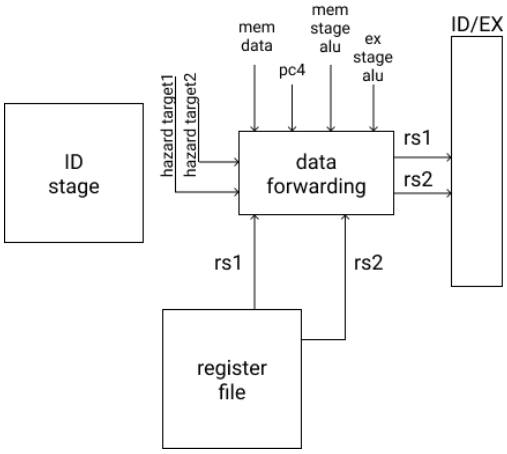


図 4: データフォワーディング

3.3.1 パイプラインフラッシュ

パイプラインフラッシュとは、ジャンプまたは分岐の制御信号がアクティブであれば、パイプラインのある値にゼロまたは、意味をなさい信号（プロセッサに変化を持たさない信号）に書き換えるようにする。最低限、その命令を実行してもプロセッサの状態（レジスタおよびメモリへの書き込み）が変化しなければ良い。

4 検証結果

実装したプロセッサを cadence 社のツール NC-Verilog を用いて性能検証する前に、正常な動作ができるように、シミュレーションを行い、用意されたテストプログラムを利用し、全ての命令が正しく動作することを確認した。シミュレーションをする際、1 クロックサイクルはあたり 10ns として実行した.

以上の機能検証が終わり、性能評価のために用いたテストプログムラ、MiBench である。先に、用意されたベンチマークプログラムは正しい出力が得られるかどかうかの test プログラムで判断する。表 2 に示すように、bitcount、dijkstra、stringsearch の 3 種類から、それぞれ test、small、large の 3 種類を持ち、ただし、large には問題があるため、ここで、large の検証結果を出さないことにする。総クロック数を求め、性能評価の基準として扱う。

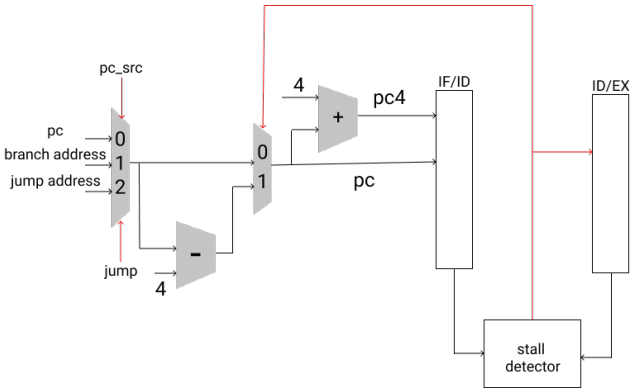


図 5: パイプラインストール

論理合成には Design Compiler というツールを用いて、最小周期の制約、面積、および消費電力を求めた。

表 2: ベンチマークの実行結果

ベンチマーク名	実行時間 [ns]
bitcnts:test	2000285
bitcnts:small	1417166995
dijkstra:test	107521545
dijkstra:small	985645445
stringsearch:test	351885
stringsearch:small	3007895

表 3: 論理合成の結果

最大遅延時間 [ns]	5.15
面積 [μm^2]	265559
消費電力 [mW]	4.7185

5 改善点

5.1 面積の減少

プロセッサの面積を削減できると、一般的にチップあたりのコストを下げることができる。その影響で消費電力も低下することになる。よって今回はプロセッサの面積が削減できる改善方法を考えた.

5.1.1 マルチプレクサー

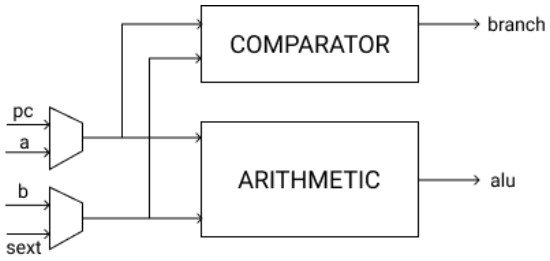
なぜマルチプレクサーの実装にこだわるか、論理合成をする時に気づいたことは、実際のプロセッサに「don't care 」の信号を論理的に考えると、この信号に対し、演算「比較、足し算など 」は無意味だと考えられる。例えば、分岐命令を考えると、WB ステージにどのような値をレジスタファイルに書き込もうが、レジスタファイルへの書き込み制御信号を１「書き込まない」にすると、結局、その値は変化を持たさないので、この場面でのマルチプレクサーへの制御信号をどんな値にしても、結果的に、変わらないと思い、「」信号をプロセッサの状態への変化がない程度に０また１の信号を出力することにした。

この原理を用いて、他のモジュールも書き換えることができた。さらに、この変化により、if-else 文も Verilog-HDL にある conditional operator を用いて、簡単ができた。従って、面積を 376773μm^2 から 268331μm^2 まで、約 29% の面積が削減できた。

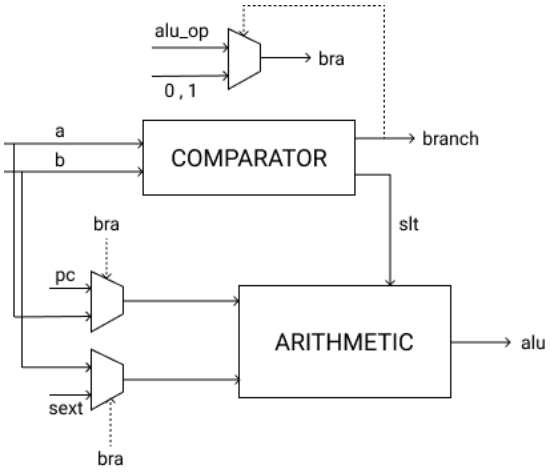
5.2 クリティカルパースの短縮

データフォワーディングを 1 つのモジュールとして実装する前には、ID ステージにある分岐先アドレスの加算機がクリティカルパースであった。さらに、分岐命令の時には、図 6a に示すように、ALU にある加算機は使われていないため、資源の無駄遣いかと考え、分岐条件の検証と同時に加算機を用いて、分岐先アドレスの計算も一緒に利用できないかと考えた。

その解決は図 6b に示している。よって、ID ステージにあるクリティカルパースは解決できた。ただし、データフォワーディングを１つのモジュールとして、実装してから、新たなクリティカルパースとして、問題になった。これの改善方法は 6.2 に詳しく述べる。



(a) 改善前の ALU



(b) 改善後の ALU

図 6: ALU での改善

6 改善できる点

6.1 パイプラインストール

3.2.2 で述べている方法には、FPGA ボード上で実装としたら、メモリとのやりとりが問題となるだろう。今回は全てシムレーション上で機能検証およびベンチマークを行ったため、メモリへの書き込みと読み込みには、半クロックサイクルでデータの扱いができるため、

これはパイプラインストーリより、ステージストールと認識しても良いと思い、確かに、問題がないが、効率が悪いだけである。これを実装する時に、パイプラインの出力をワイヤだと認識してしまったことが、自分のミスであった。

ここで、メモリへのアクセスが１クロックサイクル以上がかかるとしたら、全てのパイプラインストールが必要となっている。

本プロセッサで実装パイプラインストールは少し変える必要がある。それは、ストール信号を受ける時、今レジスタが格納しているデータをそのまま出力しただけで解決できると考えられる。もちろん、ストール検出にはメモリアクセスの成功の検出とい

う機能を追加すべきである。さらに、PC の値を一時的に停止しないといけない。

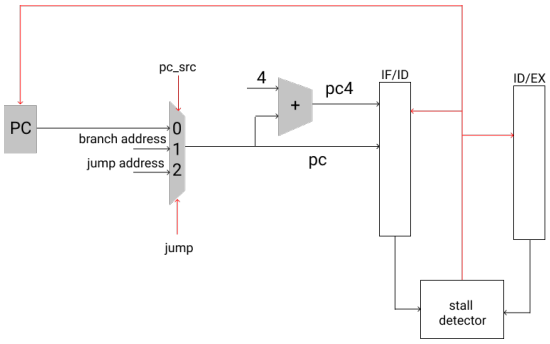


図 7: 改善後のパイプラインストール

6.2 クリティカルパースの短縮

図 4 を見れば、今のフォワーディング機がクリティカルパースになっているのは、パプラインに入る他の信号と比べたら、多少の論理ゲートを通過しないといけないパースになっているからだと考えられる。

簡潔な方法としては、フォワーディング機を EX ステージ持ち込めば、また、同じパースを作ることになるため、フォワーディングの原理を変えるべきだと考えられる。

考えられる改善は 8 に示すように、ALU のあたりにマルチプレクサーとして実装し、フォワーディングの原理を含めば、EX ステージのパースはあまり変わらず、クリティカルパースが短縮できると考えられる。

7 まとめ

実際に計画をし、設計を書いて実装することによってよりプロセッサへの理解が深めた。プロセッサ設計を通して, 理論的に学んだハザード, フォワーディング、ストールへの実装にぶつかる壁などを体験することができた。RISCV-32I にある例外処理の命令をサポートできないまま終わると、不満はあるが、この演習の目標であるプロセッサ設計への理解はしっかり守れたと思っている。しかし、設計したプロセッサを FPGA ボード上で実際に動かすことができなかったため、シムレーションと現実の違い見れなく、少々残念だと思っている。最後に、様々な場面で困難にあった時に手を差し伸べた同期の人と、サポートしてくれた先輩方に深く感謝している。

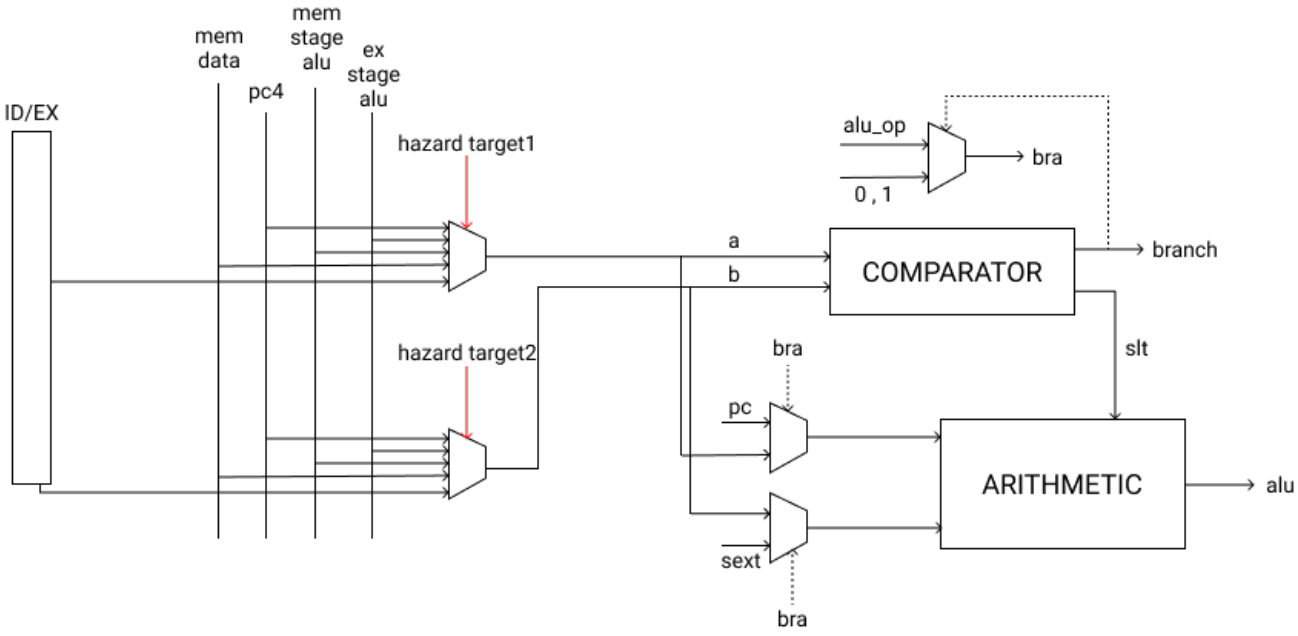


図 8: 改善後フォワーディング