後期実験：マイクロプロセッサの設計と実装

電気電子工学科3年
03210499
高原陽太

平成34 年1 月1 日

0.1 はじめに

今回のマイクロプロセッサ実験を通して、設計から実装まで自力で行った。そこで今回のレポートに関しては、基本課題と応用課題に取り組むに当たって工夫した点、苦労した点のそれぞれについて記述して行きたいと思う。

0.2 基本的な cpu の設計

まず実験 1-8 日目は基本的な cpu の設計と実装を行った。構造は wiki の資料を参考にして作成した。

工夫した箇所はストア命令を読み出す際に書き込む部分の指定をするための 4bit のフラグを指定するためのモジュールを事前に作成したことである。図 1 の store モジュールに対応している。このモジュールを作ったことによりデータメモリへの書き込みに必要な作業を分担することができた。

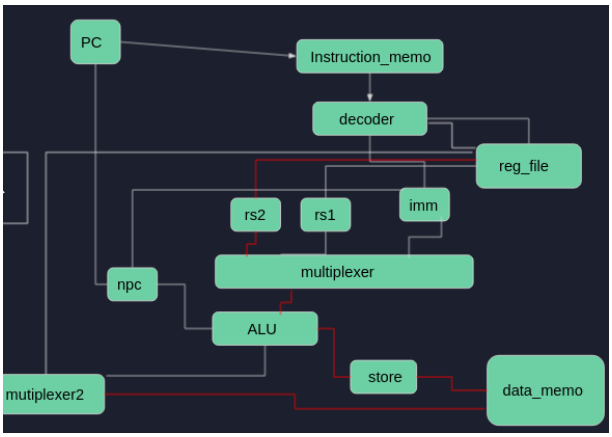

図 1: cpu の図

0.3 実験結果

実装した結果は以下の通りである。

最終的なスコアは図 2 に示されている通り、

$$Iterations/Sec = 42 \tag{1}$$

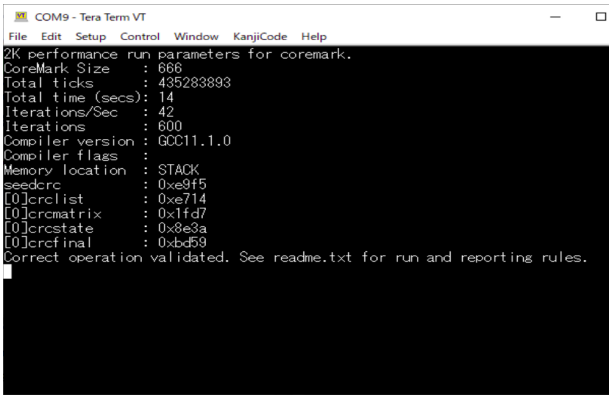

図 2: 最終的な CoreMark のスコア

であった。これは基本課題として設計した cpu についてできるかぎりクリティカルパスなどを考慮し、動作周波数を $30MHz$ として動かした際の実行結果である。

0.4　苦労した点

cpu を作る上で苦労した点は以下の通りである。
・ロードストア命令におけるメモリ番地の指定
・クロックの扱い

まずロードストア命令におけるメモリ番地の指定についてである。今回の RISC-V の仕様ではメモリのアドレシングの単位はバイトである。したがって、alu の演算結果として読み出されたアドレスではそのままメモリ中の 1 バイトを特定することにはならない。そこで読み込んだアドレスをメモリ中のデータのインデントとして利用するにはそれぞれのアドレスを 4 で割る必要があり、このことを理解するのに時間を要した。またデータ語中のバイトの並べ方がリトルエンディアンであるため、alu から受け取ったアドレスの下位 2bit を 4 で割った余りについてロード命令のメモリを読み出しを場合分けする必要があり、実装に苦心した。

また、クロックを用いてメモリを制御することに関しても工夫した。今回の cpu はシングルプロセッサであるので 1 クロックでメモリに関する処理についても終了させる必要がある。しかし、フリップフロップにしてしまうと資源消費が大きくなってしまう。そのためできるだけブロック

RAM の書き方にすることで資源消費量を抑えようとした。そこで命令メモリとデータメモリに読み出したり書き込むタイミングを別々にした。

具体的には命令メモリの読み出しをクロックの立ち上がりで行い、データメモリに関する読み出し書き込みはクロックの立ち下がりで行うようにした。

0.5 発展課題

・RV32M への対応
・クリティカルパスの短縮
・function 文による記述

まず、乗算器除算器の実装を行った。基本的には decoder と alu についての書き換えのみを行った。decoder については仕様書を見る限り 25bit 目の値が 1 かどうかで判別できるので、できるだけコードが簡略できるようにした。alu に関しては符号付きかどうかに注意しながら実装した。その際に例外的な数値などは define.vh にまとめて記載することで間違いを減らすようにした。
RV32M への対応に関して最も苦労したのは実行時間増加に伴うクリティカルパスの延長であった。乗算器が非常に遅いためそれによって今まで動いていた動作周波数で稼働しなくなってしまった。そのため遅延しても稼働するようにクリティカルパスの短縮と function 文による記述という工夫を行った。

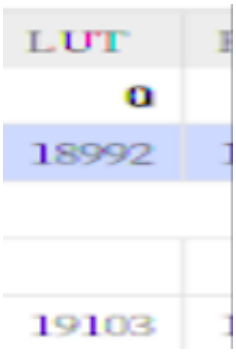

図 3: 最初の消費資源量

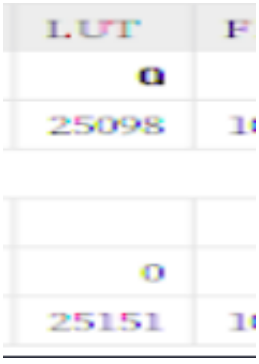

図 4: RV32M 後の消費資源量

また乗算器除算器を取り入れたことにより消費資源量も増えてしまっ

た。これも回路のコードに余分な箇所、重複している箇所があることに起因していると考えた。

下図がクリティカルパスである。

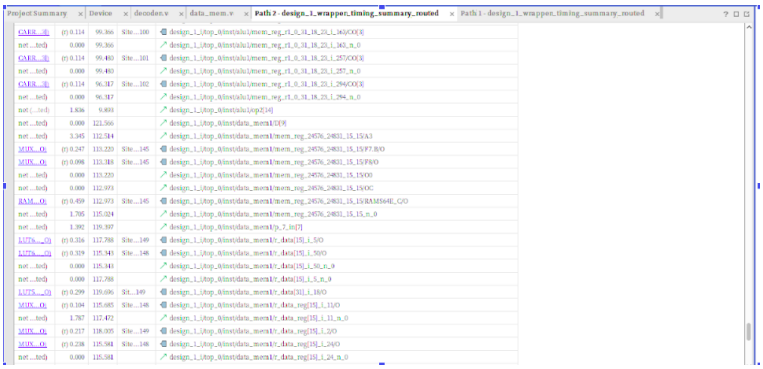

図 5: クリティカルパス

大まかに見ると図の cpu の赤線部分のような経路がクリティカルパスになっていることが分かる。データメモリから読み出してくるロード命令が一番遅いと考えた。そこでできるだけデータメモリに関する処理を簡単にすることを考えた。考えたのが function 文による記述である。自分のコードを見返す限り、データメモリだけでなく decoder や alu についても always 文の中で巨大な case 文を用いて記述してしまったことで回路が複雑化してしまい、Vivado が最適化をできなくなってしまったのではないかと考えた。

実際に function 文で記述し直したところ、decoder と alu に関しては記述量を半分以下にまで減らすことができた。

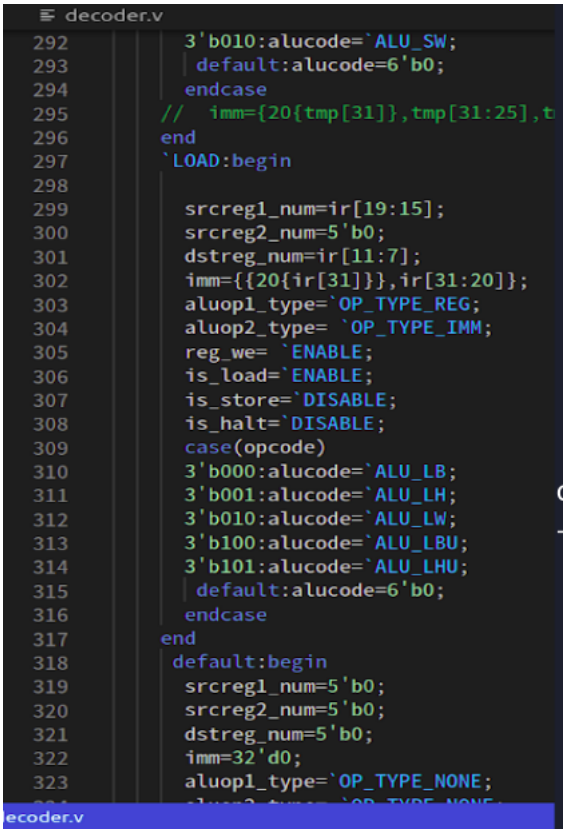

図 6: 当初の decoder のコード

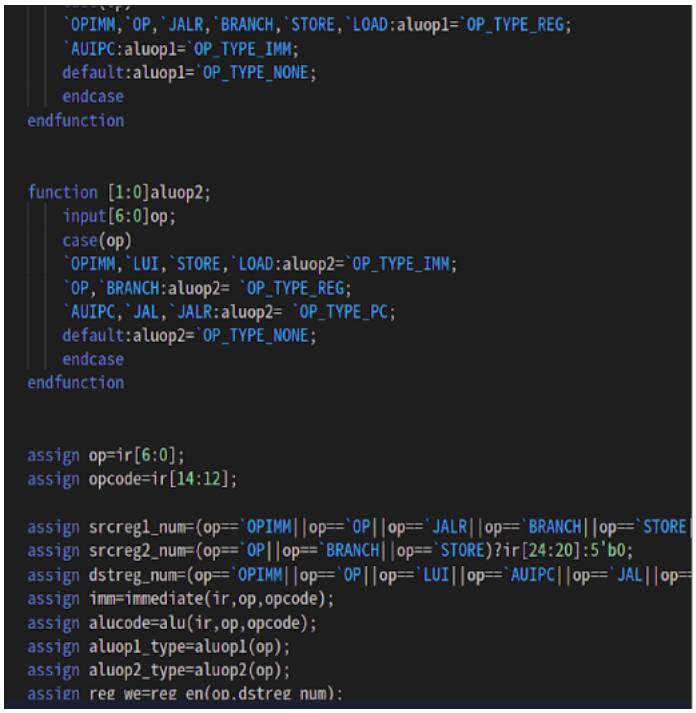

図 7: 簡略化した decoder のコード

しかし reg として存在しているメモリを参照する必要があるデータメモリを function 文で書くことができなかった。function 文と同様の機能を導入するには always(*) でも記述することができるがそれだとブロック RAM にならないためデータメモリに関しては always 文と case 文を用いて記述せざるを得なかった。
データメモリに関しては上記のようにストアのフラグを立てる箇所を分離するなどしてできるだけ処理を減らすことでクリティカルパスの短縮を試みた。

0.6 今後の展望

単に function 文でモジュールを簡略化させるだけでは効果が得られない箇所もあるので、回路の演算を共通化させるなどしてより無駄のない

コードを書くこととクリティカルパスの短縮のため、より機能を細分化してクリティカルパスに関連するモジュールを分割するなどをしてより高い周波数に対応できるようにしたい。