CPU 実験最終レポート

7 班 05-231001 阿部 桃大

2024 年 2 月 23 日

1　7 班のコンピュータの構成

命令セットアーキテクチャ

命令セットアーキテクチャは、RISC-V をベースとしている。意図としては、コアにおいて高速実行が可能な回路を構成しやすいようにするためである。

実装した命令は、以下の通りである。

LUI	上位 20 ビットを即値として rd に格納
JAL	即値を pc に加算し、次の命令のアドレスを rd に格納
JALR	即値を pc に加算し、次の命令のアドレスを rd に格納
BRANCH	条件分岐命令
	BEQ, BNE, BLT, BGE, BLTU, BGEU
LOAD	メモリからデータを読み込む命令
	LW
STORE	メモリにデータを書き込む命令
	SW
OP-IMM	即値演算命令
	ADDI,SLLI, SLTI, SLTIU, XORI, SRAI,SRLI,ORI,ANDI
OP	演算命令
	ADD, SUB, SLL, SLT, SLTU, XOR, SRA, SRL, OR, AND
OP_FP	浮動小数点演算命令
	FADD,FSUB,FMUL,FDIV,FSQRT,FEQ,FLT,FABS,FCVTWS,FCVTSW
LOAD_FP	浮動小数点数を読み込む命令
	FLW
STORE_FP	浮動小数点数を書き込む命令
	FSW

以降は、独自命令である。

FADDI　浮動小数点数レジスタを対象に即値を加算する命令
FLUI　浮動小数点数レジスタを対象に上位 20 ビットを即値として rd に格納
IN/FIN　入力命令
OUT/FOUT 出力命令

また、コンパイラの出力は、以下の擬似命令も含む。これらの擬似命令は、アセンブラにおいて、上記の命令に変換される。

NOP　何もしない命令
LI　即値を rd に格納する命令
MV　rs1 の値を rd に格納する命令
NOT　rs1 のビット反転を rd に格納する命令
NEG　rs1 の符号反転を rd に格納する命令
SEQZ　rs1==1 を rd に格納する命令
SNEZ　rs1!=1 を rd に格納する命令
SLTZ　rs1<0 を rd に格納する命令
SGTZ　rs1>0 を rd に格納する命令
BEQZ　rs1==0 の時分岐する命令
BNEZ　rs1!=0 の時分岐する命令
BLEZ　rs1<=0 の時分岐する命令
BGEZ　rs1>=0 の時分岐する命令
BLTZ　rs1<0 の時分岐する命令
BGTZ　rs1>0 の時分岐する命令
BLE　rs1<=rs2 の時分岐する命令
BGT　rs1>rs2 の時分岐する命令
BLEU　rs1<=rs2 の時分岐する命令 (符号無し)
BGTU　rs1>rs2 の時分岐する命令 (符号無し)
J　ラベルにジャンプする命令
JR　レジスタの値にジャンプする命令
RET　ra の値にジャンプする命令
CALL　ラベルにジャンプし、次の命令のアドレスを ra に格納する命令
LA　ラベルのアドレスを rd に格納する命令
FLI　浮動小数点数の即値を rd に格納する命令
FMV　浮動小数点数の rs1 の値を rd に格納する命令

コアについて

コアは、fetch,decode,execute,memory access,write back の 5 段構成からなるパイプラインプロセッサである。浮動小数演算などの execute ステージで複数クロックを必要とする命令を実行する際や、メモリアクセスに複数クロック要する場合には、全てのステージにおいてストールが発生する。また、分岐予測は行っておらず、全て untaken として予測実行している。

フォアーディングを実装しているため、データハザードによるストールは減らすことに成功している。

レジスタは、通常の RISC-V の構成である汎用レジスタ 32 個、浮動小数点レジスタ 32 個を持っている。

メモリについて

キャッシュが存在し、以下のような構成になっている。

L1 キャッシュ 16B,1entry
L2 キャッシュ ダイレクトマップ, 256KB
tagsize: 9bit, indexsize: 14bit, offsetsize: 4bit

L1 キャッシュは、最後に書き込みが行われた L2 キャッシュのページを保持している。

また、高速化のためにメモリのクロックは、コアのクロックの反転で動いている。

FPU について

FPU では、単精度浮動小数点数を扱う。また、FPU のレジスタは、通常のレジスタとは別になっている。

各命令に対し、以下のクロック数で演算が行われる。

命令	クロック数
FABS,FLT,FGT	2
FMUL	3
FADD,FSUB,FDIV,FSQRT,FCVTWS,FCVTSW	5

コンパイラについて

コンパイラは、mincaml をベースに OCaml で実装した。このコンパイラは OCaml のコードを入力として、それを上記の命令セットに変換する。

mincaml で行われる最適化に加えて、以下の最適化を行っている。

レジスタ割り当て　使わないレジスタを使うように割り当てる。
スタックの最適化　積む必要のないスタックを削除する。
グローバル変数の導入　グローバル変数を絶対アドレスに保持し、それを使うように変換する。

アセンブラについて

アセンブラは、二つのファイルに分かれている。一つは、アセンブリ言語を機械語に変換するアセンブラであり、もう一つは、機械語をバイナリに変換するアセンブラである。

アセンブリ言語を機械語に変換するアセンブラでは、擬似命令を含む命令列を我々の班の命令セットのみからなる命令列に変換する。

機械語をバイナリに変換するアセンブラでは、機械語をバイナリに変換する。このアセンブラは、機械語をバイナリに変換するだけでなく、機械語のアドレスをラベルに変換する。

シミュレータについて

シミュレータは、C++ で実装した。シミュレーターは、命令列と入力を受け取り、それを実行する。

シミュレータ (ISS) では、以下の実行統計情報を得ることができる。

実行結果　出力、結果 (a0 レジスタの値)、各レジスタの値 (オプション)
実行時間予測　実行命令数、予測実行クロック数、予測実行時間
メモリ性能　メモリアクセス回数、キャッシュミス率
分岐予測　分岐回数、分岐予測ミス率
ISS の性能　実行時間、MIPS

2　レイトレの実行結果

レイトレの実行時間は、256×256 で 378.5266sec,512×512 で 1340.1752sec であった。また、出力画像は、シミュレータの実行結果と完全に一致し、以下のような画像が出力された。

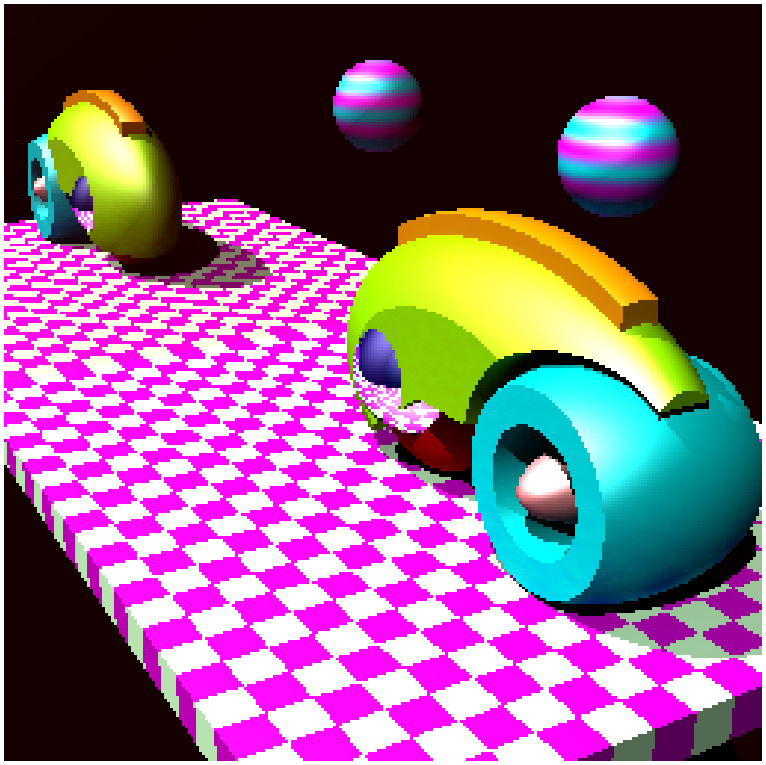

図 1　レイトレの出力画像 (256 ×256)

3　シミュレータについて

この節では、私の担当であるシミュレータについて説明する。

概要

全体的な実装の方針として、高速実行できることを重視している。そのため、パイプライン処理は行わず、命令は一つずつ実行するようにしている。

実行時間の正確なシミュレート及び実機でのパラメータ調整を行うため、メモリアクセスと分岐予測の性能を計測している。

シミュレータの実行過程は、以下のようになっている。

1. 命令列を構造体に読み込む
2. 現在の PC に応じた命令を実行し、レジスタやメモリを更新する
3. 各統計情報、PC を更新し、2 に戻る
4. 命令列の終端に到達したら、終了する
5. 終了時に統計情報を出力する

ファイル分割

シミュレータは、以下のようにファイルを分割している。

iss.cpp	シミュレータ本体
cal.h/cpp	FPU 演算の実装
hardware.h/cpp	pc, クロック, メモリ, レジスタ, 分岐予測器などのハードウェア関連の実装
order.h/cpp	命令, オペコード, レジスタ名の管理
print.h/cpp	エラー出力、実行列の出力

なお、iss.cpp 以外の各ファイルは、ライブラリとして扱い、アセンブラの実装でも使用している。

FPU 演算の実装

FPU 演算はナイーブに実装すると、実機での実行結果から誤差が生じてしまう。そのため、FPU 演算は、実機での verilog 実装と全く同じロジックの関数を実装することで、実機での実行結果と一致するようにしている。

また、この実装による出力結果と、デフォルトの単浮動小数演算計算の計算結果を何億ケースも比較し、我々の FPU 実装の誤差が要件を満たしていることを確認している。

メモリアクセスの計測

メモリアクセスに関しては、キャッシュの挙動を正確にシミュレートするようにしている。すなわち、メモリアクセス関数が呼ばれるたびに、キャッシュを参照し、状態に応じて更新およびキャッシュミスのカウントを行っている。

また、シミュレーター上では L2 キャッシュはセットアソシアティブ方式で実装している。実機の実装であるダイレクトマップ方式は、セットアソシアティブ方式の 1way の特殊な場合であるため、シミュレーションは問題なく行える。

シミュレーター上で tag size, index size, offset size,ways を変更することで、最適なキャッシュの構成を探索することができる。上に述べた実機のメモリ構成は、このシミュレーターを用いて探索した結果である。

分岐予測の計測

最適化の途中で分岐予測器の実装を試していたので、分岐予測器の挙動をシミュレートできるようにしている。

分岐予測器は Gshare 方式を実装している。Gshare 方式は、分岐履歴を PC の値と XOR して、その結果をインデックスとして使って飽和カウンタを参照する方式である。

実機での実装においてクリティカルパスが長くなってしまい、実行時間が遅くなってしまうため、最終的には全て untaken として予測実行するようにしている。シミュレータでも同様にした。

ストール、フラッシュの実装

パイプライン処理を行っていないため、ストールやフラッシュが発生する場合を網羅することで、実機と同様のクロック数遷移をするようにしている。

我々のコアにおいてストールが発生するのは、以下の場合である。

- 実行に複数クロックを要する命令 (FPU 演算) を実行する場合
- メモリアクセス (どのキャッシュにヒットするかによってクロック数が変わる)
- 分岐予測ミス
- lw 命令によるデータハザード
- jal, jalr 命令による分岐

以上のケースで、それぞれ無駄になるクロック数分だけクロックを進めることで、実機と同様のクロック数遷移をするようにしている。

工夫点

シミュレータの実装において行った工夫点は、以下の通りである。

オプションの実装

シミュレータには、以下のオプションを実装している。

-p	命令列の出力
-r	レジスタの出力
-t	ISS 実行時間の出力
-d	デバッグモード
-rt	レイトレ実行用の形式

こうして必ずしも必要でない情報を出力しないことで、実行時間を短縮している。

1 命令実行あたりの処理の削減

繰り返し実行されることになる部分を重点的に最適化した。

- 命令列は先に構造体に読み込むことで、デコードの時間を削減
- シフト演算で書ける部分は、シフト演算を利用
- パイプライン処理なしでクロック数シミュレートを行うことで、処理を削減

最適化オプション

実行結果が変わらない範囲で、最適化オプションを用いている。makefile には、-Wall -O3 -std=c++17 を指定している。

パラメータの調整

パラメータ調整をしやすくするために、各パラメータはヘッダファイルで define などで定義し、一括で変更できるようにしている。

実行方法の整備

実際にテストを行う際には、アセンブリ言語から実行を行いたいことが多い。そのため、Makefile を整備し、アセンブラ、シミュレータを一括で実行できるようにしている。

他にも、コンパイラ係のデバッグにおいては特にコンパイラの出力をシミュレータに入力することが多いため、コンパイラ、アセンブラ、シミュレータの実行を一括で行えるようにシェルスクリプトを整備している。

実行結果

シミュレータの実行結果は、以下の通りである。

```
simulator result:
result:         256
order count: 16139132855
clocks: 29417656707
estimated time: 367.721[s] (6m7s)
memory performance:
memory access:  6187964979
l1 cache miss:  4299008145
l1 cache miss rate:     69.4737[%]
cache miss:     6757621
cache miss rate:    0.109206[%]

branch performance:
branch count:   766831548
predict fail:   336853007
fail rate:     43.9279[%]
```

```
simulator performance:
simulating time:      120.29[s]
MIPS:     134.168
```

予測実行時間は 367.721[s] であり、実機での実行時間 378.5266[s] との誤差は 3% 以内である。

4 改善点

最後に、我々の実装において改善点として考えられることを述べる。

全体

他の班と比較して、我々の班の実行は時間がかかっていた。主な要因となっていたのは、まず命令数の多さである。これは、コンパイラにまだまだ最適化の余地があることを示している。特に、メモリアクセスが 161 億命令のうちの 62 億命令を占めていることから、メモリアクセスの最適化が求められる。私はコンパイラについてそこまで詳しくないので、詳しいことはわからないが、ラベル呼び出しの際のレジスタ退避などが無駄になっている可能性がある。また、jal 命令や jalr 命令を多用しているため、ストールが多く発生していることも要因の一つである。

また、命令セットやコアの構造を最適化しきれていない部分もあった。例えば、レジスタを増やすことで、レジスタ退避を減らすことができるかもしれない。他にも、分岐予測器がないこと、データハザード以外の様々なケースでストールが発生してしまうなど、コアの構造においても改善の余地がある。

シミュレータ

シミュレータについては主に 2 つ改善点がある。

まず、実行時間予測が実機との誤差が 3% 以内であるが、それでも誤差がある。原因としては、ストールやフラッシュが起こる場合として網羅できていないことが考えられる。様々なケースでテストを行いストールやフラッシュが発生する条件を網羅することで、実機との誤差をさらに減らすことができる。

また、統計データをとらないことでもっと高速に実行できるようにすることも考えられる。現在のシミュレータでは、メモリアクセス、分岐予測の統計データは必ず取るようになっており、処理が増える原因となっている。これらの実装を行う前は現在より 3 ～ 5 倍程度高速であったため、こういった統計データをとらないコードも別に用意すると、そのくらい高速に動作することが期待でき、コンパイラ係のデバッグに役に立つと思われる。