CPU 実験 最終レポート

1　コンピュータの構成

1.1　ISA とアーキテクチャ

ISA　RISCV をベースとした 32bit 固定長命令を採用した. 浮動小数レジスタと整数レジスタの識別をなくしつつ, 合計 128 個レジスタに増設している. 最終調整の段階でレジスタファイルが大きくコアの負担になったため 64 個に減らす案もあったが, レジスタ数を減らした際に同程度の命令数を保てなかったため, コンパイラ係の都合で 128 レジスタのままになった. また, 小数を含めすべてのデータが 32bit であるため, バイトアドレッシングではなくワードアドレッシングに変更. これにより配列アクセスのインデックスをシフトする必要性がなくなり, またアドレス指定にかかるビット数も節約ができた. 頻出する依存関係を持つ命令の組を調査した結果として, 小数比較分岐と, レジスタで lw のオフセットを指定する命令が最終段階で追加された. 補遺 A に利用可能なすべての命令による表を掲載した.

コア　アーキテクチャとしては, 周波数 95MHz のインオーダースーパースカラプロセッサが採用された. 1 サイクルに 2 命令をフェッチし, デコードした命令を FIFO に入れて最大 2 命令を並列に実行できる. ただし, メモリアクセスは 1 サイクルに 1 つずつしか行えない. 命令フェッチ, デコードおよびフォワーディングをするフロントエンドが 5 ステージ, メモリアクセスと演算により命令の実行を行うバックエンドが 5 ステージと細分化された構成になっている. また, スーパースカラ化の影響によって 2 命令デコードしてから次の pc を決定するのでは間に合わないため, fetch pc を元に Branch Target Buffer を利用して分岐先予測をする. 各 BRAM に 2 ずつしかポートがない状況で 128 個ものレジスタファイルを 4-read2-write で実現するためには, Live Value Table として知られているテクニックが用いられている. これは BRAM 自体は潤沢に利用できることを前提として, n-read に関しては n 個に複製した RAM から 1 ポートずつ読むことにし, まず 1-write n-read の RAM を実現する. 次に m-write に関しては m 個の 1-write n-read RAM のうちにどこかに最新の値を書き込み, どれが最新の値を持っているのかを LUT で記録することで疑似的に m-write n-read の RAM を実現する.

コンパイラ　実装はすべて OCaml による. フロントエンドは mincaml をそのまま利用しつつ, 追加でいくつか最適化モジュールを作ったりグローバル配列導入のために一部仕様を変更したりなどした. シミュレータ完動および実機完動の段階のバックエンドは, mincaml の PowerPC 版を参考に書いた RISCV 用のバックエンドを用いた. 最終的にはコンパイラ係向け課題のレジスタ割当を実装する都合で SSA 形式へと変更したバックエンドを用いることとなった. 命令数削減に貢献した最適化は, グローバル配列導入, 共通部分式削除, ジャンプの連鎖の縮退と fall-through の利用, while 式導入, 分岐の畳み込みなどが挙げられる. 三角関数 (sin, cos, atan) および print_int の mincaml 言語によるソフトウェア実装はコンパイラ係が担当した.

FPU/メモリ　各段を 150MHz で動かしたときの段数と FPU モジュール一覧を以下の表にまとめた. sin, cos, atan はソフトウェア実装されているため, ハードウェアのモジュールにはない.

表 1: FPU モジュールと段数

段数	モジュール一覧
1	fmul, fsqr, itof, ftoi
3	fadd, fsub
5	finv, sqrt
6	floor
8	fdiv

キャッシュシステムは L1 キャッシュのみをダイレクトマップ, ラインサイズ 128, ライン数 16384 で実装. 素朴な状態遷移では, Read hit 発生後に idle 状態へと戻ってしまっていたので連続したメモリアクセスをする場合, キャッシュヒットであっても間に必ず 1 クロックが挟まってしまっていた. そこで, タグ比較状態でも CPU からのリクエストを受け付けられるようにすることで, この余分な 1 クロックを削減できるように改善された.

最終的にキャッシュの構成がダイレクトマップに決定されるまで, セットアソシアティブで置き換えポリシーを疑似LRU, (厳密な)LRU, FIFO などとする多様な組み合わせがシミュレータ及び実機を用いて試行された. しかし同じキャッシュサイズで比較したとき, セットアソシアティブではキャッシュヒット時のレイテンシが 1 から 3 クロックに伸びるデメリットを埋め合わせるほど, 高いヒット率を実現するポリシーはなかったためダイレクトマップキャッシュが本番では採用されることとなった.

シミュレータ　実装はすべて C 言語による. シミュレータの main 部分である simulator.c, ISA の設定などを司る config.c, キャッシュのシミュレートを担う cache.c, fpu のエミュレートを行う fpu.c(+finv,fsqrt で参照するテーブルの fpu_mem.txt, fpu_mem_sqrt.txt) とデバッグ用のログ出力関数を収めた logger.c から構成されている. オプション-b を指定することでアセンブラとして動作させることもできるようになっている. 他のオプションとしては, オーソドックスなステップ実行と行数指定によるブレークポイント機能に加え, レジスタやメモリの更新ログを出力したりレジスタ番号とその値によりブレークポイントを設定したりなどのデバッグ支援用のオプションがコア係やコンパイラ係の希望により適宜追加されていった. また, キャッシュの各種ポリシーによるヒット率を調査するモードや, 各命令ごとの実行回数およびメモリアクセスが起きたアドレスなどの統計情報を取得するモードがアーキテクチャの決定や改良に役立てられた.

1.2　実行時間

最終発表会における記録を小数第 1 位までで表 2 に示した.

表 2: レイトレーシングの実行時間

ピクセルサイズ	実行時間 [s]
128 × 128	27.6
256 × 256	86.5
512 × 512	299.9

2　コンパイラ

2.1　単位取得条件について

コンパイラ係の CPU 実験の単位取得条件として 1. 班のプロセッサ用コンパイラを開発する, 2. コンパイラ実験で要求される課題の条件を満たすことが要求されている. また, コンパイラ実験の単位取得条件としては, 1. 自作 CPU (or シミュレータ) 上でレイトレプログラムを完動させる, 2. 共通課題を 6 つ以上提出, 3. コンパイラ係向け課題を 1 つ以上提出かつ口頭諮問を受ける, 4. プロセッサ担当の出す課題をクリアすることが必要とされている.

CPU-1 およびコンパイラ-1 に関しては, コンパイラ係面談においてコンパイルからシミュレータ動作までのデモンストレーションをし, 最終発表会にて開発したコンパイラによって出力されたアセンブリを用いて実機上でも完動することを示したことにより条件が満たされていると考えられる. コンパイラ-2 は第 1,2,3,4,5,7,11 回課題の計 7 つの共通課題提出を持って条件を満たした. コンパイラ-3 は第 9 回のレジスタ割当課題を提出の上, 2/6 に口頭諮問を受けて条件を満たした. コンパイラ-4 についてはコア係と協議して仕様に従ったり実装を追加したりすることを約束した要件に関してはすべてきちんと実装したため条件を満たしていると考えている. これによりコンパイラ実験側の条件をすべて満たしたため, CPU-2 の条件も達成され, すべての単位取得条件を満たしたと考えられる.

2.2　構成

mincaml へ新たに追加したモジュール名と, その内容を表 3 へまとめた. SSA 形式で使える命令は実際のアーキテクチャにある命令 + Φ関数 + Save/Reload となっている. mincaml の中間表現では, Add/Sub/Mul/Div や IfLE/IfEq のようにフォーマットが同じで同様の処理内容を書く命令でもヴァリアントが 1 段しかないためそれぞれに対してパターンマッチを書かなければならない部分が不満であった. そこで, SSA 命令の中間表現としては命令のフォーマット別のヴァリアントを外側, オペコード部分だけを内側の引数なしヴァリアントとすることで冗長な場合分けをなるべく排した.

表 3: 新たに実装したモジュールと内容一覧

モジュール名	内容
フロントエンド	
Array_access	エイリアス解析
Mloc	コンパイラ実験第 1 回 プログラムの元の行数を持つメタ情報を管理するデータ構造
(non-SSA) バックエンド	
GlobalsConvert	グローバル配列用の初期化関数置換
SSA バックエンド	
Ssa	SSA 形式の命令のデータ構造設定
Dfa	Asm.t から SSA 命令列と CFG へ変換 + グローバル配列番号付け
Bpschedule	分岐で同じ値を使う場合に分岐前へ定義を移動, レジスタ負荷低減, 分岐の畳み込み 1
Regssa	レジスタ割当, 分岐の畳み込み 2, mv と値定義の一体化
Schedule	BB 内スケジューリング
Genoutlib	create_array などのライブラリ関数の出力
Emitssa	空ブロックへのジャンプ縮退, ループブロックの出力順変更, アセンブリの出力

2.3 工夫点

以降は主に実装した最適化についての説明である. 実装方法やデータ構造の持ち方などに特色があるようなコンパイラにはならなかった.

レジスタ割当 コンパイラ係向け課題として SSA 形式によるレジスタ割当を実装した. この際にポイントとなったのは関数呼び出し規約の扱いと callee save register の導入である.

SSA 形式における干渉グラフは弦グラフ (完全グラフの一種) となり, 制約がなければ多項式時間で最適な彩色が可能となる. 具体的には dominator tree の dfs をしながら貪欲に割り当てることで最適解がえられると知られている. 一方で現実的には関数呼び出し規約によって, 一部のノードに対して色の制約がある状況を考えることになる. [1] によると各色 1 ノードまでの制約なら多項式時間で最適解を出せるため, 関数呼び出し時点で生存区間分割をすることで制約条件を加味した最適な割当が実現できるようだ. 実際の手順としては独立な干渉グラフごとに彩色し, 間の変数の受け渡しに Φ 関数を用いる. しかし, これはレジスタ割当実装当初に複数引数をとる Φ 関数に対応しておらず, この点を含めて上記理論に基づく実装を完全に実現するのは難しいのではないかと思ったため, 関数引数の定義に割当してほしいレジスタをヒントとして与えるだけに留めることにした. 仮にこの時点で関数引数とする際とは異なるレジスタに割り当てられた場合, 関数呼び出し直前に然るべきレジスタ mv を挿入して呼び出し規約に従う. 引数ヒントによる 256×256 レイトレプログラムの命令数はインライン数 0 で 4.1%, インライン数 600 で 2.4% 減少した.

callee save register の導入は, レイトレにループへと変換できる末尾再帰が多いので効果的であると予想し行った. ループ途中でサブルーチン呼び出しをする場合でもループの前後で退避/復帰すれば十分でありループ途中にこのような非本質的なコードが増えることを防ぐ効果がある.

分岐関連 インライン展開を進めると, 主に関数の返り値として Φ 関数に書き込まれる値が定数であり, 以降の処理を決定するための後続の条件分岐が taken なのか untaken なのかが静的に定まるケースが頻出するようになった. そこで, このような部分コードに対して無駄な比較を避けて分岐先に直接ジャンプできるように CFG を変更する最適化をした. これを 2.2 の表では分岐の畳み込みという名称で掲載している. またこの最適化の処理は Φ 関数で書きこむ値が条件分岐のみに利用されるのか, さらにその先のコードで生存しているのかによってどの段階で適用可能かと, 削除できる命令の範囲に差が生じる. 仮に分岐のみで利用する場合は, 分岐先へ直接ジャンプさせる際に Φ 関数を同時に消せることができる. この場合はレジスタ割当前に行うことができるので, Bpschedule モジュール内で行う. 分岐後も変数が生存する場合は Φ 関数を削除できない. 問題点となるのは, Φ 関数で書き込みするレジスタが不明であるという部分であるので, レジスタ割当をして Φ 関数をレジスタコピーに落とし込んでからであれば分岐の畳み込みが可能である. これはレジスタ割当後に Regssa モジュールで行う.

また, この最適化を行うと分岐の then ブロックと else ブロックの入次数がそれぞれ 1 ずつではなくなるケースが生じてくる. else ブロックに対しては必ず分岐ラベルをつけてジャンプを挿入することになるが, then ブロックに対しては可能ならば fall-through を利用してジャンプを不要にしたい. このような事情があるので, もし then ブロックの入次数が 1 より大きく, else ブロックの入次数が 1 であれば分岐の条件を逆転させ, then と else を交換する.

ジャンプ関連　上記の最適化を行うと, 空ブロックへのジャンプを連鎖が生じるケースなどがある. 空ブロック b の出次数が 1 である場合に限って, $a \to b \to c$ のような辺があるならば $a \to c$ に縮退させる操作を Emitssa で行っている. また, ループの末尾ブロックに必ず無条件ジャンプが生じてしまうことに関しても分岐の条件を逆転させ, ブロックの出力順を調整することでループ内のジャンプを削減する最適化も行った.

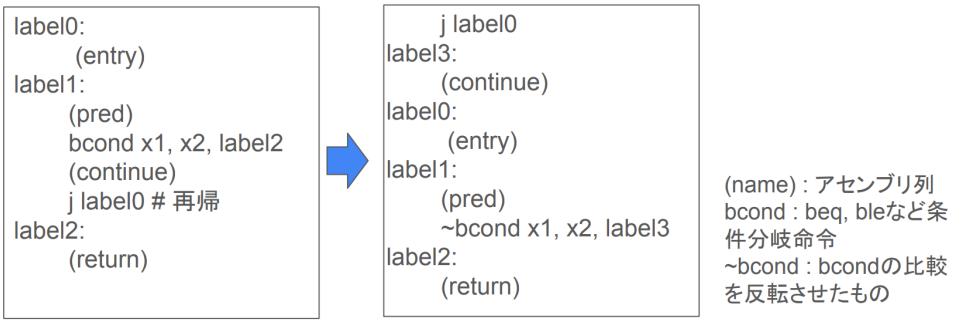

図 1: ループブロックの出力順変更

共通部分式削除　コンパイラ実験第 2 回の課題として実装した共通部分式削除では, 関数ごとに共通で利用する関数ラベルや整数小数即値が共通化される程度であった. このときは副作用の解析を実装する余裕がなかったので, 配列アクセスに関しての共通部分式は放置されたままであった. 最適化が進むにつれ, 全命令の中で lw の占める割合が 3 割を超えさらに増え続けた. メモリアクセスが実行時間時のボトルネックになることも踏まえ, 少しでもメモリ関連の命令を減らすべくエイリアス解析をして lw を減らしたいと思った. 余力があれば不要な sw を減らす最適化も行いたかったが, そこまでは手が回らなかった. 間で配列に書き込みがないことが確証できる lw 同士でしか共通部分式削除できないので, エイリアス解析をする必要が生じる. この際に難しいのは多次元配列やタプルの要素になっている配列である. これらはエイリアスグループを求めるのにオフセットを加味していく必要があるため, 複数次元にわたるものやタプルを挟んだ読み出しについては分析をあきらめ, 共通部分式削除もしないことにした. また, ループ変数などで複数の配列の実体に対してエイリアスとなりうるものについても厳密な分析が困難であるため最適化をしないことにした. 共通化できる候補に関しては, 配列アクセスの際に配列名と定数オフセットの組をキーとして読み出した値を代入した変数名を登録する. 間の書き込みをケアするためには, 二種類の方針がある. 1 つ目は書き込み可能性があり, 更新後の値も確定しないので保守的に登録を削除する場合だ. これはサブルーチンで書き込まれる可能性がある場合と, 定数でないオフセットにより配列にアクセスする場合が該当する. 2 つ目は定数オフセットで配列書き換えをする場合であり, この場合は更新後の変数に登録を付け替える.

while 式　再帰関数を素朴にインライン展開すると実行効率が落ちることに 1 月末ごろにようやく気付いた. mincaml ではループアンローリングのように展開が起こるわけではなく, ループの最初の数回が展開されるので, 複数同じコードが生じる上に展開された部分でループが終了しなければ残りの実行のために展開前の関数の方へと飛ばなければならないため無駄が生じやすい. そこで, ループ構造を保ったままインライン展開をしたいという動機の元で while 式を導入することにした. これによってインライン数を増やしたときに命令数が劇的に減少した. 一方で, ループの導入によりプログラムの性質が変わり, 実装が難しくなったという面もあった. 例えば, レジスタ割当でループの初週とループバックしたときの環境の違いを考慮して適切にリロードを挿入したりする必要が生じた. また, 配列アクセスの共通部分式削除でもループ内で lw する場合はループの初回ではループ前の lw と共通の値の場合であっても, ループの残りで更新される場合は共通部分式削除できない. この点を踏まえてループ内外をまたいで配列の共通部分式削除は起こらないように変更した.

spike シミュレータの利用　最適化とは無関係な話題であるが, CPU 実験を行う上の工夫の一つではあると考えられるためここで触れる. 多くの班が 1stISA として RISCV に近い, あるいは完全に従った命令セットを利用する. 当班では後者のケースであったため, コンパイラの RISCV 用バックエンドが完成し, 自班のシミュレータが機能し始めるまでの間に既存の RISCV シミュレータ spike を活用した. これはシミュレータの完成を待たずともコンパイラのデバッグを進められて時間を有効活用できるというメリットに加え, コンパイラとシミュレータの間でバグの要因を切り分けることにも役立ったと考えている. デバッグが最も難航するのはどこにバグの大元があるかわからない場合であると思っているため, 先にコンパイラのデバッグが終わると, 問題のないことがわかっているアセンブリをシミュレータ係のデバッグのために提供できるので, コンパイラが原因なのかシミュレータが原因なのかわからずに気を揉ませることがなくなって事がスムーズに運びやすくなるのではないかと思う. 一つ改善点があったとすればシミュレータ係に存在意義を疑わせてしまうことになりうるので, きちんと班員と協議したうえでこのような選択肢をとってみるのがよかったのかもしれない.

2.4 総括

大規模なプログラムを長期にわたって書き続けるということ自体が今回の CPU 実験で初めて体験したことだったように思う. そのような開発に不慣れであるために, プログラムのモジュール化, 計画的なコーディングと定期的なリファクタリングなどが不足しており最終的に無駄の多く読みにくい, 安全性の低いコンパイラになってしまったことが最も心残りである. また, デバッグに多くの時間を費やすことになったため, 先にコストを払って後のデバッグを楽にするような設計/支援ツールの開発を心掛けるべきであったというのも大きな学びである. とりあえず動かすために面倒な部分を避けて妥協した実装をしようという判断をした局面も少なくなかったが, 結局回り道をしても同じくらいの手間がかかった上に実装が余計に複雑になってしまった点もいくつかあったので, 目先の成果を優先せずできるだけ参考資料に忠実な実装を心掛けるほうが長い目で見れば結果的に良かったのではないかというように思われる.

補遺 A 命令セットアーキテクチャ

表 4: 命令セット一覧

op	funct1	funct6	Type	instruction
01000	1	-	I	lw rd, imm(rsi)
01001	0	-	I	addi rd, rs1, imm
01010	0	-	I	slli rd, rs1, imm
01011	0	-	I	srai rd, rs1, imm
01100	0	-	I	xori rd, rs1, imm
01111	0	-	I	jalr rd, rs1, imm
11100	-	-	S	sw rs2, imm(rs1)
1000_	-	-	B	beq rs1, rs2, imm
1001_	-	-	B	bne rs1, rs2, imm
1010_	-	-	B	blt rs1, rs2, imm
1011_	-	-	B	bge rs1, rs2, imm
1101_	-	-	B	bfle rs1, rs2, imm
1111_	-	-	B	bfeq rs1, rs2, imm
00001	-	000000	R	add rd, rs1, rs2
00010	-	000000	R	sll rd, rs1, rs2
00100	-	000000	R	xor rd, rs1, rs2
00101	-	000000	R	fabs rd, rs1
00110	-	000000	R	sub rd, rs1, rs2
00111	-	000000	R	feq rd, rs1, rs2
00111	-	000001	R	fle rd, rs1, rs2
00111	-	000010	R	fadd rd, rs1, rs2
00111	-	000011	R	fsub rd, rs1, rs2
00111	-	000100	R	fmul rd, rs1, rs2
00111	-	000101	R	ftoi rd, fs1
00111	-	000110	R	itof fd, rs1
00111	-	001000	R	fdiv rd, rs1, rs2
00111	-	001001	R	fsqrt rd, rs1
00111	-	001100	R	floor fd, fs1
00111	-	001100	R	floor fd, fs1
00111	-	101000	R	lwr rd, rs1, rs2
00111	-	100000	R	in32 rd
00111	-	110000	R	out8 rs1
11000	-	-	U	jal rd, imm
11001	-	-	U	lui rd, imm

補遺 B　braid アーキテクチャのサポート

本番で利用されなかったモジュールの一つとして, braid.ml がある. これは [2] の論文で提案されたコアのアーキテクチャを採用することを目指した際に, コンパイラ側で OoO 実行のために必要な分析の一部を担うために追加した機能の一部である. 当該アーキテクチャのデザインとしては, 基本ブロックをいくつかの独立した実行の流れ braid に分割し, braid 内をスーパースカラ, braid 間を OoO 実行するというようになっている. また, 使用されるレジスタが braid 内に利用が限られる internal register と braid 間の情報の受け渡しに利用できる external register に二分されており, 分岐予測失敗時には external register と pc だけを復帰すればよいので従来の OoO アーキテクチャよりは低コストに抑えられることが特色である.

braid への分割はコンパイラが責任をもって行うことになっており, コア係の要請を受けて以下のグラフに示されたように素朴な braid への分割と external register に割り付けるべき値 (赤い辺) と internal register に割り付ければよい値 (黒い辺) を区別するところまでを実装していた. このアーキテクチャへの移行は残り日数と労力を鑑みて, 途中で挫折という形になった.

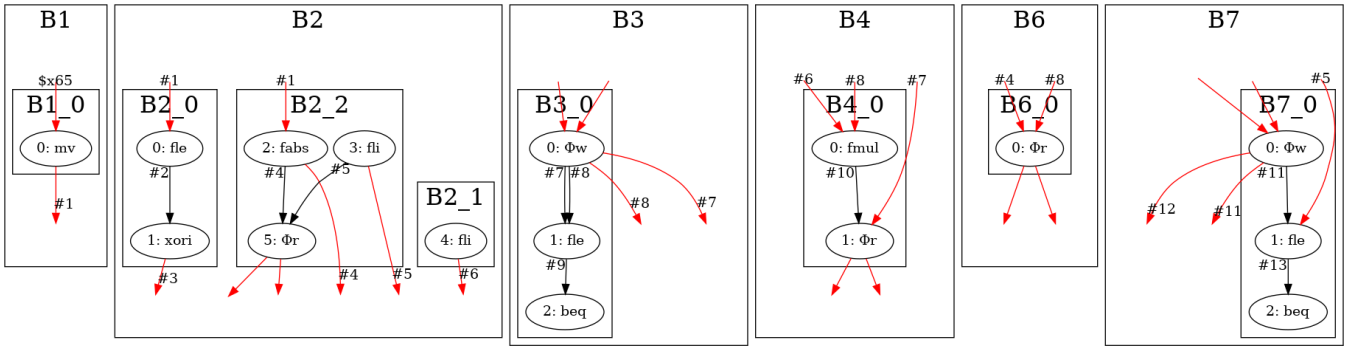

図 2: braid 分割した基本ブロックの例

補遺 C　PPM フォーマット変換

実行効率を考えると P6 形式を選択することになるが, デバッグをする人間には P3 形式の方が都合がいい. しかし, デバッグのためにいちいち P3 で実行し直すのも面倒であり, かつ出力まわりのバグの場合は P3 形式に変更することで出現するバグが変わることもある. よくよく考えてみると, P3 と P6 の情報量は同じなので互いに変換することができるということに気がついた. そこで, c のプログラムとして P6 から P3 に変換, そして指定した ppm ファイルと diff をとれるようにしてデバッグに役立てた. これは ppmConvert.c としてコンパイラのディレクトリに入っている.

参考文献

[1] Sebastian Hack. Register Allocation for Programs in SSA Form. *Universitätverlag Karlsruhe*, 2006. ISBN : 978-3-86644-180-4.

[2] Francis Tseng and Yale N. Patt. 2008. Achieving Out-of-Order Performance with Almost In-Order Complexity. SIGARCH Comput. Archit. News 36, 3 (June 2008), 3–12. https://doi.org/10.1145/1394608.1382169