機能設計仕様書

作成者：Pyii Phyo Maung
Student ID: 1029322149

June 9, 2023

1　機能設計仕様

名前	担当の部分	推進状況
加藤利梓	phasecounter, ALU, shifter, pll, pipeline, 全体	デバッグまで終わった
神事倫紀	RegisterFile, register, Branch, ctl, removechattering, display	デバッグまで終わった
オースティン	pc, ram, sorting algorithm, new instructions, assembler	デバッグまで終わった
全員	各モジュール、アルゴリズムのデバッグと編集	1msより速く動くSIMPLEを作成する目的ができた。5段パイプライン化ができ、100MHzで動くSIMPLEプロセッサができた。

Table 1: 分担状況

2　ソーティングアルゴリズム

2.1　assembler.py

アッセンブラーファイルに拡張命令の分を追加した。

```
"ADDI": lambda rb, d:
        "10" + "001" + to_binary(rb, 3) + to_binary(d, 8, signed=True),
    "SUBI": lambda rb, d:
        "10" + "010" + to_binary(rb, 3) + to_binary(d, 8, signed=True),
    "CMPI": lambda rb, d:
        "10" + "011" + to_binary(rb, 3) + to_binary(d, 8, signed=True),
    "B": lambda d:
        "10" + "100" + "000" + to_binary(d, 8, signed=True),
    "SLI": lambda rb, d1, d2:
        "10" + "101" + to_binary(rb, 3) + to_binary(d1, 4, signed=True)
        + to_binary(d2, 4, signed=False),
```

2.2　基数ソートのアルゴリズム

基数ソートのアルゴリズムを何回もアセンブリコードで書き、さらに速くなるため、サイクル数が少なくなるため、頑張った。以下は作成したソートの

アルゴリズムである。

2.2.1 基数ソート1ビットずつ

```
LI 1,1
LI 3,1
SLL 3,15
LI 0,1
SLL 0,11
SUB 0,1
LI 2,1
SLL 2,10
LI 5,0
LD 4,0,2
AND 4,3
BNE 1
ADD 5,1
ADD 2,1
CMP 2,0
BLT −7
LI 6,1
SLL 6,11
LI 2,1
SLL 2,10
ADD 6,2
SUB 6,5
MOV 5,6
LI 6,1
SLL 6,11
LD 4,0,2
LD 7,0,2
AND 4,3
BE 2
ST 7,0,6
ADD 6,1
BNE 2
ST 7,0,5
```

```
ADD 5,1
ADD 2,1
CMP 2,0
BLT -12
LI 2,1
SLL 2,10
LI 7,1
SLL 7,11
LD 4,0,7
ADD 7,1
ST 4,0,2
ADD 2,1
CMP 2,0
BLT -6
LI 2,1
SLL 2,11
LI 0,1
SLL 0,10
ADD 0,2
LI 4,0
ST 4,0,2
ADD 2,1
CMP 2,0
BLT -5
HLT
```

プログラムの最初のブロックは、後の操作で使用されるバイナリ マスクを設定します。 AND 演算でビットを順次シフトおよびマスクすることにより、整数の 2 進表現内の 1 の数をカウントするループを確立します。 レジスター 5 に保管されるこのカウントは、バイナリー・マスクを構成します。ループはすべてのビットが処理されるまで継続します。

2 番目のブロックは、ビット シフト操作を実行して、最初のブロックで作成されたマスクを目的の位置に移動します。 また、1 のカウントを元の値から減算して修正し、この結果をレジスター 6 に保管します。

3 番目のブロックは並べ替えアルゴリズムの中核です。 メモリから値をフェッチし、前に作成したマスクを使用してそれらをマスクして特定のビットを分離します。 特定の条件を満たすと、値を取得し、それが 1 か 0 かに応

じてメモリに格納し直します。このプロセスは、すべての要素が特定のビットについて評価されるまで繰り返されます。

4 番目のブロックは、整数のソート済みリストを反復処理し、整数を最終的なソート順に並べ替えます。 メモリから値をロードし、新しい位置に書き込み、すべての要素が処理されるまで繰り返します。 このステップで並べ替え操作が完了します。

最後のブロックは使用済みメモリをクリアし、並べ替えプロセス中に使用されたメモリ セグメントをリセットしてゼロにします。 この手順は、残留データが今後の操作に影響を与えないようにするため、または別の並べ替えの準備をするために重要です。 この後、プログラムは停止します。

要約すると、このアセンブリ コードは、バイナリ マスクを利用して特定のビット位置に基づいて整数を並べ替える並べ替えアルゴリズムを実装します。 ソートされる実際の値、およびソートに使用される特定の基準は、メモリとレジスタの初期内容、およびこのアセンブリ言語の正確な仕様によって異なります。

2.3 基数ソートコピーなし、sortedとr-sorted検知入れ済み

```
SLI 0,1,10
SLI 1,1,11
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 4,5
BLT 2
BE 29
B 9
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 0,1
BNE 1
HLT
CMP 4,5
BLE -8
B 29
LD 4,0,0
```

```
LD 5,1,0
ADDI 0,1
CMP 0,1
BNE 11
SLI 0,1,10
SUBI 1,1
LD 4,0,0
LD 5,0,1
ST 4,0,1
ST 5,0,0
ADDI 0,1
SUBI 1,1
CMP 0,1
BLE -8
HLT
CMP 5,4
BLE -18
B 10
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 0,1
BNE 1
HLT
CMP 4,5
BE -8
BLE -37
B -29
LI 3,1
SLI 0,1,10
SLI 1,1,11
MOV 2,1
SUBI 2,1
ADD 2,0
SLI 6,1,15
LI 7,0
```

```
LD 4,0,0
LD 5,0,0
AND 4,3
BNE 2
ST 5,0,1
ADDI 1,1
BE 2
ST 5,0,2
SUBI 2,1
AND 5,6
BE 1
ADDI 7,1
ADDI 0,1
CMP 1,2
BLE -15
ST 7,0,3
SLL 3,1
SLI 0,1,11
SLI 7,1,10
SLI 6,1,11
SUBI 6,1
LD 4,0,0
LD 5,0,0
AND 4,3
BNE 2
ST 5,0,7
ADDI 7,1
BE 2
ST 5,0,6
SUBI 6,1
CMP 0,2
ADDI 0,1
BLT -12
SLI 0,1,10
SLI 2,1,11
ADD 0,2
```

```
SUBI 0,1
LD 4,0,0
LD 5,0,0
AND 4,3
BNE 2
ST 5,0,7
ADDI 7,1
BE 2
ST 5,0,6
SUBI 6,1
CMP 1,0
SUBI 0,1
BLT -12
SLL 3,1
SLI 0,1,10
SLI 1,1,11
ADD 2,0
SUBI 2,1
LD 4,0,0
LD 5,0,0
AND 4,3
BNE 2
ST 5,0,1
ADDI 1,1
BE 2
ST 5,0,2
SUBI 2,1
CMP 0,6
ADDI 0,1
BLT -12
SLI 0,1,11
SUBI 0,1
LD 4,0,0
LD 5,0,0
AND 4,3
BNE 2
```

```
ST 5,0,1
ADDI 1,1
BE 2
ST 5,0,2
SUBI 2,1
CMP 7,0
SUBI 0,1
BLT -12
SLI 5,1,14
AND 5,3
BE -67
SLL 3,1
SLI 0,1,11
SLI 7,1,10
LI 4,1
LD 6,0,4
ADD 6,7
LD 4,0,0
LD 5,0,0
AND 4,3
BE 2
ST 5,0,7
ADDI 7,1
BNE 2
ST 5,0,6
ADDI 6,1
CMP 0,2
ADDI 0,1
BLT -12
SLI 0,1,10
SLI 5,1,11
ADD 0,5
SUBI 0,1
LD 4,0,0
LD 5,0,0
AND 4,3
```

```
BE  2
ST  5,0,7
ADDI  7,1
BNE  2
ST  5,0,6
ADDI  6,1
CMP  1,0
SUBI  0,1
BLT  −12
HLT
```

最初のブロックは、レジスタ 0 と 1 のビットをシフトすることによってインデックスを設定します。次に、値をメモリからレジスタ 4 と 5 にロードし、これらの値を比較します。 この比較により条件分岐が推進され、必要に応じて値を交換することで並べ替え手順が調整されるようです。

2 番目のブロックはループに入り、各メモリ空間を調べ、値をロードし、比較を実行します。 レジスター 0 がレジスター 1 と等しくない場合、プログラムは停止します。そうでない場合は、値の比較を続行し、必要に応じて順序を変更します。

プログラムの 3 番目のブロックは別のループを開始します。 インデックスを変更し、メモリから値をロードし、比較の結果に基づいてメモリ内の値を交換します。 値が希望の順序で並べ替えられるまで、この手順が繰り返されます。

4 番目のブロックは、ブロック 3 の並べ替え操作を繰り返しているように見え、並べ替えが正確であることを保証します。 ソートが確認された場合は、ブロック 5 の実行にジャンプします。

5 番目のブロックは別のインデックスのセットを確立し、ループを開始します。 このループでは、マスクを作成し、それを使用してメモリ内の値を操作します。 マスクの出力に応じて、値は異なる場所に保存されます。 このプロセスは、すべての値がチェックされ、それに応じて操作されるまで続行されます。

最後のブロックは、前のブロックで見られた操作を組み合わせたもののようですが、インデックスとマスクが異なります。 メモリ内の値を調べて操作し、マスクの出力に基づいて特定の場所に保存します。 これらの操作は、すべての値が処理されるまで継続されます。 これらの操作の後、プログラムは最終的に停止します。

要約すると、アセンブリ コードは、ビット表現に基づいてメモリ内の値

を操作する並べ替えアルゴリズムを実装しているようです。 ソートの具体的な基準は、メモリとレジスタの初期状態、およびこのアセンブリ言語の正確な仕様によって異なります。 このアルゴリズムは以前のアルゴリズムよりも複雑で、追加のメモリ操作層が含まれています。

2.4 基数ソート8ビットずつ

```
SLI 0,1,10
SLI 1,1,11
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 4,5
BLT 2
BE 29
B 9
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 0,1
BNE 1
HLT
CMP 4,5
BLE -8
B 29
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 0,1
BNE 11
SLI 0,1,10
SUBI 1,1
LD 4,0,0
LD 5,0,1
ST 4,0,1
ST 5,0,0
ADDI 0,1
```

```
SUBI 1,1
CMP 0,1
BLE -8
HLT
CMP 5,4
BLE -18
B 10
LD 4,0,0
LD 5,1,0
ADDI 0,1
CMP 0,1
BNE 1
HLT
CMP 4,5
BE -8
BLE -37
B -29
SLI 2,1,8
LI 0,0
LI 1,0
ST 0,0,1
ADDI 1,1
CMP 1,2
BLT -4
SUBI 2,1
MOV 0,2
SLI 1,1,10
SLI 3,1,11
SUBI 3,1
LD 4,0,1
AND 4,0
LD 5,1,4
ADDI 5,1
ST 5,1,4
CMP 1,3
ADDI 1,1
```

```
BLT -8
LI 0,0
SLI 5,1,11
LD 6,0,0
ADD 6,5
ST 6,0,0
ADDI 0,1
LD 5,0,0
ADD 6,5
ST 6,0,0
CMP 0,2
BLT -6
SLI 1,1,10
SLI 3,1,11
SUBI 3,1
LD 4,0,1
LD 6,0,1
AND 4,0
LD 5,0,4
ST 6,0,5
ADDI 5,1
ST 5,0,4
CMP 1,3
ADDI 1,1
BLT -10
ADDI 2,2
LI 0,0
LI 1,0
ST 0,0,1
ADDI 1,1
CMP 1,2
BLT -4
SUBI 2,2
MOV 0,2
SLI 1,1,11
SLI 3,1,10
```

```
SUBI 3,1
ADD 3,1
LD 4,0,1
SRL 4,8
AND 4,0
LD 5,0,4
ADDI 5,1
ST 5,0,4
CMP 1,3
ADDI 1,1
BLT -9
SLI 0,1,7
SLI 5,1,10
LD 6,0,0
ADD 6,5
ST 5,0,0
ADDI 0,1
LD 5,0,0
ST 6,0,0
ADD 6,5
CMP 0,2
BLT -6
SLI 1,1,7
SUBI 1,1
LI 0,0
LD 5,0,0
ST 6,0,0
ADDI 0,1
ADD 6,5
LD 5,0,0
ST 6,0,0
CMP 0,1
BLT -6
SLI 1,1,11
SLI 3,1,10
SUBI 3,1
```

```
ADD 3,1
LD 4,0,1
SRL 4,8
LD 6,0,1
AND 4,2
LD 5,0,4
ST 6,0,5
ADDI 5,1
ST 5,0,4
CMP 1,3
ADDI 1,1
BLT −11
HLT
```

この基数ソート アルゴリズムは、個々の数字 (より一般的には「基数」または「基数」) を処理することによって整数をソートします。 8 ビット整数 (0 - 255) は、256 個の可能な値があり、多くの異なる値を持つデータに対して効率的な方法となるため、この方法に適しています。

アルゴリズムは、各バイト値 (0 - 255) の出現をカウントすることから始まります。 これは、値のリストを反復処理し、「count」配列内の対応するインデックスのカウントをインクリメントすることによって実現されます。 これにより、各インデックス i の値が入力内のそのバイトのインスタンスの数を示す配列が得られます。

その後、アルゴリズムは、前のバイトのカウントを合計することによって、並べ替えられた出力内の各バイトの開始アドレスを計算します。 入力に対して 2 回目のパスを実行し、前のステップで計算された開始アドレスに従って出力配列に各値を配置し、そのアドレスをインクリメントします。 この操作により、ソートの安定性が維持されます。 これは最下位桁基数ソートであるため、下位 8 ビットが最初にソートされます。

次に、ソート アルゴリズムは上位 8 ビットに対して同じ手順を繰り返します。 ただし、数値が署名されていることが考慮されます。 コンピュータで符号付き整数を表す最も一般的な方法である 2 の補数では、負の数は最上位ビット (「符号ビット」) を設定 (1) することで表され、正の数は設定を解除する (0) ことで表されます。 したがって、2 番目のパスでは、アルゴリズムは最初に 16 番目のビットが 1 (負の数) である数値を処理し、次に 0 (正の数値) である数値を処理します。

この方法を使用すると、アルゴリズムは整数の固定サイズを利用して線形

時間で効率的に並べ替えることができ、データの範囲や項目数に関係なく、データを 2 回パスするだけで済みます。 そのため、バブル ソートや挿入ソート、さらには基数ソートの以前のバージョンなど、他のソート アルゴリズムと比較して、より少ないサイクルで高速に実行できます。 この基数ソート方法では、以前の実装で使用されていた広範なコピー、削除、およびカウント手順に加えて、比較および交換操作が回避されるため、サイクル数が削減されます。 代わりに、整数の内部バイナリ表現と、レジスタの代わりにメモリ自体を利用して、効率的な並べ替えを行います。

3　感想

皆が自分が得意な部分、自分がやりたい部分を選んでやりましたので自分がやりたいもの、作りたいものを上手く作れて満足したと思う。もちろん作ったものはモジュールとして分かれておりますが、それを全員で一緒に読み、理解できた上でデバッグ作業を行う。それで、全員が全体の動き、論理、などを全て分かるようになった。