アーキテクチャ拡張仕様書

作成者：Pyii Phyo Maung
Student ID: 1029322149

June 9, 2023

1 SIMPLE/B基本アーキテクチャからの拡張仕様の説明

- 5段パイプライン
- 不成立分岐予測
- 即値命令（ADDI, SLI, CMPI, SUBI）
- ハーバードアーキテクチャ
- 7SEGを用いた左レジスタデバッガー
- ソーティングアルゴリズム

1.1 5段パイプライン化

最新のCPU（Central Processing Unit）では、パイプラインという概念が、動作効率を高め、命令の処理速度を向上させるために重要である。パイプラインとは、CPUの使用率を最大化し、アイドル時間を最小化するために、命令の実行を重ねることを意味する。パイプラインの最も基本的な例として、RISC（Reduced Instruction Set Computer）アーキテクチャで主に使用されている5段パイプラインがあります： IF（命令フェッチ）、ID（命令デコード）、EX（実行）、MEM（メモリアクセス）、WB（ライトバック）です。

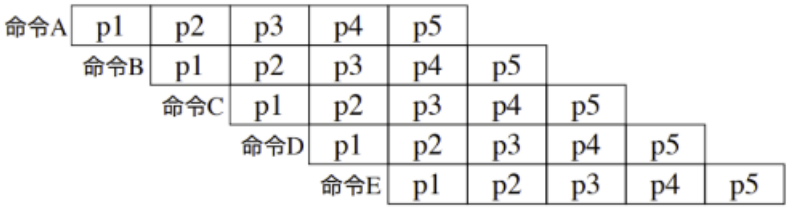
命令フェッチステージ（IF）は、パイプラインプロセスの最初のステージです。プログラムカウンタ（PC）が指すメモリ位置から命令をフェッチする。命令がフェッチされると、PCは次のシーケンシャル命令のアドレスに更新されます。これは、次のサイクルで正しい命令をフェッチすることを保証するために行われます。命令デコードステージ（ID）では、フェッチされた命令がデコードされ、CPUは命令の種類と実行すべきオペレーションを特定する。また、この段階では、命令が必要な場合には、レジスタファイルから必要なデータを読み出すことも行われる。

デコードステージに続いて、エクゼキューション（EX）ステージに到達します。ここでは、命令で指定された実際の演算が実行される。加算や減算などの算術演算や、AND、OR、NOTなどの論理演算が行われる。これらの演算の結果は、内部のレジスタに格納される。続くステージは、メモリアクセス（MEM）ステージである。このステージは、メモリからデータをロードしたり、メモリにデータを保存したりと、命令内容がデータメモリに関わ

る場合に使用されます。命令によっては、CPUがメモリからデータを読み込んだり、メモリにデータを書き込んだりすることもある。

パイプラインの最終ステージは、ライトバック（WB）ステージです。このステージでは、実行ステージまたはメモリアクセスステージからの結果が、レジスタファイルに書き戻されます。このステージで、パイプラインにおける命令のライフサイクルが完了する。パイプラインの設計は、各クロックサイクルで新しい命令の処理を開始することを目的としています。パイプラインが満杯になると、最大5つの命令が同時に処理され、それぞれが異なるステージで処理されます。したがって、5段パイプラインは、命令の実行を効率的に管理し、並列処理を促進することで、CPUの性能を大幅に向上させることができます。

- 5 ステージパイプラインプロセッサを作る逐次活性化
  - P1 命令フェッチ
  - P2 命令デコード、レジスタ読み出し
  - P3 演算
  - P4 主記憶アクセス
  - P5 レジスタ書き込み／PC更新



1.1.1 シンプルパイプラインに追加したコードの例

```
1     phase3ctl p3IFID(.ALUSrc1in(ALUSrc1&hazard),.ALUSrc2in(ALUSrc2&
          hazard),.ALUorshifterin(ALUorshifter&hazard),.AS_BCin(AS_BC
          &hazard&(~brch_sig_wire)),.MemReadin(MemRead&hazard),.SLIin
          (SLI),.Rain(Inst_wire1[13:11]&{3{hazard}}),.Rbin(Inst_wire1
          [10:8]&{3{hazard}}),.opcodein(opcode_wire&{4{hazard}}),
2       .clk(clk),.rst_n(rst_n),.ALUSrc1out(x0),.ALUSrc2out(x1),.
            ALUorshifterout(x2),.AS_BCout(x3),.MemReadout(x14),.
            SLIout(x16),.Raout(z5),.Rbout(z6),.opcodeout(y0));
3   phase4ctl p4IFID(.MemWritein(MemWrite&hazard&(~brch_sig_wire)), .
        Inputin(Input&hazard),.Branchin(Branch|{3{brch_sig_wire}}),.
        clk(clk),.rst_n(rst_n),.MemWriteout(x4), .Inputout(x5),.
```

```
        Branchout(z0));
4   phase5ctl p5IFID(.MemtoRegin(MemtoReg&hazard),.RegWritein(
        RegWrite&hazard&(~brch_sig_wire)),.RegDstin(RegDst_wire&{3{
        hazard}}),.clk(clk),.rst_n(rst_n),.MemtoRegout(x8),.
        RegWriteout(x9),.RegDstout(z2));
5   phase3ctl p3IDEX(.ALUSrc1in(x0),.ALUSrc2in(x1),.ALUorshifterin(x2
        ),.AS_BCin(x3&(~brch_sig_wire)),.MemReadin(x14),.SLIin(x16),.
        Rain(z5),.Rbin(z6),.opcodein(y0),
6         .clk(clk),.rst_n(rst_n),.ALUSrc1out(ALUSrc1out),.ALUSrc2out(
              ALUSrc2out),.ALUorshifterout(ALUorshifterout),.AS_BCout(
              AS_BCout),.MemReadout(x15),.SLIout(SLIout),.Raout(z7),.
              Rbout(z8),.opcodeout(opcodeout));
7   phase4ctl p4IDEX(.MemWritein(x4&(~brch_sig_wire)), .Inputin(x5),.
        Branchin(z0|{3{brch_sig_wire}}),.clk(clk),.rst_n(rst_n),.
        MemWriteout(x6), .Inputout(x7),.Branchout(z1));
8   phase5ctl p5IDEX(.MemtoRegin(x8),.RegWritein(x9&(~brch_sig_wire))
        ,.RegDstin(z2),.clk(clk),.rst_n(rst_n),.MemtoRegout(x10),.
        RegWriteout(x11),.RegDstout(z3));
9   phase4ctl p4EXMEM(.MemWritein(x6&(~brch_sig_wire)), .Inputin(x7)
        ,.Branchin(z1|{3{brch_sig_wire}}),.clk(clk),.rst_n(rst_n),.
        MemWriteout(MemWriteout), .Inputout(Inputout),.Branchout(
        Branchout));
10  phase5ctl p5EXMEM(.MemtoRegin(x10),.RegWritein(x11&(~
        brch_sig_wire)),.RegDstin(z3),.clk(clk),.rst_n(rst_n),.
        MemtoRegout(x12),.RegWriteout(x13),.RegDstout(z4));
11  phase5ctl p5MEMWB(.MemtoRegin(x12),.RegWritein(x13),.RegDstin(z4)
        ,.clk(clk),.rst_n(rst_n),.MemtoRegout(MemtoRegout),.
        RegWriteout(RegWriteout),.RegDstout(RegDstout));
```

この5段パイプライン化により、実行する速さが5倍ぐらいになった。

1.2 不成立分岐予測

分岐予測は、最新の高性能プロセッサにおいて、高い命令スループットを維持するために使用される重要な技術である。プロセッサが条件分岐命令に遭遇した場合、その分岐が行われる（すなわち、条件が真で制御がコードの別の部分にジャンプする）か、行われない（すなわち、条件が偽で制御が次の連続命令に進む）かを推測または「予測」する必要があります。

無効」または「誤予測」分岐予測は、プロセッサの予測が分岐条件の実際の結果と一致しない場合に発生します。例えば、プロセッサが「分岐しない」と予測していたのに、実際に分岐条件が成立して分岐してしまった場合、それは「誤予測分岐」です。この場合、誤った予測に基づいてフェッチされデコードされた命令は破棄され、プロセッサは正しい経路から再び

フェッチを開始しなければならない。これはパイプラインストールやブランチペナルティと呼ばれ、時間とリソースの損失となり、プロセッサの性能に大きな影響を及ぼしかねません。

分岐予測アルゴリズムは、このような予測ミスを最小限に抑えるように努めています。その手法は、単純な静的手法（always predict taken, always predict not taken）から、コードの実行時動作を観察して予測を行う、より洗練された動的手法まで様々です。動的予測手法には、1つの分岐の履歴を利用して予測を行う1レベル予測手法と、複数の分岐の履歴を利用して予測を行う2レベル予測手法があります。

これらの技術を駆使しても、完璧な予測方法は存在せず、無効な分岐予測も発生します。分岐予測の精度を向上させることは、コンピュータアーキテクチャーの重要な研究分野であり、小さな改善でもプロセッサー全体の性能を大幅に向上させることができるからです。

1.3 即値命令（ADDI, SUBI, CMPI, SLI）

拡張命令として、ADDI,SUBI,CMPI,SLIという4つの命令を追加した。ADDI命令は、ソースレジスタの内容と即値データの両方に対して加算を実行します。その結果をデスティネーションレジスタに格納します。dは8ビットです。SUBI 命令は、ソースレジスタの内容と即値データの両方に対して引算を実行します。その結果をデスティネーションレジスタに格納します。dは8ビットです。CMPI命令は，汎用レジスタ（GPR）RAの内容と8ビット符号なし整数SIを符号なし整数として比較し，条件レジスタフィールドBFのいずれかのビットをセットする。SLI命令はあるレジスタにロード（LI）し、その値をすぐに左論理シフトする。

名前	op1	op2	rb	d	操作
ADDI	10	001	3-bit input	8-bit input	r[rb]=r[rb]+signext[d]
SUBI	10	010	3-bit input	8-bit input	r[rb]=r[rb]-signext[d]
CMPI	10	011	3-bit input	8-bit input	r[rb] - signext[d]
SLI	10	101	3-bit input	3-bit register input and 5-bit d input	r[Rb]=signext[d]= shift_left_logical (r[Rd],signext[d])

Table 1: 拡張命令

コンピュータアーキテクチャーの分野では、特にアセンブリ言語において、ADDI、SUBI、CMPI、SLIなどの命令が、効率と性能の向上に大きな役割を果たしました。これらの演算により、演算処理に必要なアセンブリコー

ドの量やレジスタの数が大幅に削減され、処理速度やサイクル効率の大幅な向上につながった。

ADDI（加算即値）、SUBI（減算即値）、CMPI（比較即値）は、定数値（即値）とレジスタで直接演算できる命令の一例である。これに対して、従来のADD、SUB、CMP命令では、オペランドごとに2つのレジスタが必要でした。ADDI、SUBI、CMPIは、命令自体に即値が組み込まれているため、第2オペランドを保持するための余分なレジスタが不要になります。このようにレジスタの使用量を減らすことで、レジスタの割り当てを合理化し、オペランドをフェッチするためのメモリトラフィックを減らし、プロセッササイクルを節約することができます。

SLI（Shift Left Immediate）も同様に、シフト操作を簡略化します。従来は、シフト量をレジスタにロードする必要があり、追加命令と空きレジスタが必要でした。SLIでは、シフト量を命令に統合することで、必要なアセンブリコードとレジスタの使用量を削減することができます。

さらに、この「即時」命令は、コード・シーケンスの最適化にも役立ちます。例えば、ADDIがない場合、定数を使った加算演算を行うには、定数をレジスタにロードし、それを別のレジスタに加算するという別々の命令が必要になります。ADDIは、この2つの演算を1つの命令にまとめ、命令数を減らすことで、実行に必要なサイクル数を削減します。

これらの命令の利点は、コードとレジスタの使用量を即座に削減できることにとどまりません。命令とレジスタの操作回数を減らすことで、パイプラインのストールやハザードの可能性を減らすこともできます。また、先行する命令に結果が依存するような命令が少なくなるため、制御フローが単純化され、分岐予測の精度も向上します。

結論として、ADDI、SUBI、CMPI、SLIなどの命令は、アセンブリ言語プログラムの最適化と性能向上に大きな効果があります。算術演算を簡略化し、リソースの使用量を減らすことで、より効率的な実行とプロセッサの高速化に寄与する。

1.4　ハーバードアーキテクチャ

ハーバード・アーキテクチャとは、命令とデータの記憶と経路を物理的に分離したアーキテクチャのことで、私たちのグループのカスタムメイド・コンピュータの設計のインスピレーション源となっています。この明確な分離が、私たち独自のニーズに合わせて計算速度や効率を飛躍的に向上させる大きなメリットとなっています。

ハーバード・アーキテクチャの最大の特長は、プログラム・メモリとデー

タ・メモリに同時にアクセスできることです。この並列性により、命令のフェッチとデータのロードやストアを同時に行うことができ、命令の実行にかかる時間を大幅に短縮することができます。このような計算速度の大幅な向上は、私たちの考え抜かれた設計の賜物です。さらに、データと命令を分離することで、干渉の可能性を排除し、システムの信頼性を高めています。

さらに、データメモリと命令メモリを明確に分離することで、それぞれのメモリに異なるビット幅を割り当てることができ、特定の要件に対応できる柔軟性を備えています。例えば、大量のデータを比較的単純な命令で処理するシステムでは、データメモリの幅を命令メモリよりも広くすることができます。このような選択的な最適化により、システム全体の性能と効率が向上し、ハーバード・アーキテクチャをベースとしたカスタム設計の利点が明らかになりました。つまり、命令メモリとデータメモリを明確に分けることで、ハーバード・ベースのシステムは高速化、適応性、効率化を実現し、私たちのグループとして構築した特注コンピュータの価値を証明しているのです。

1.5 7SEGを用いたレジスタデバッガー

```
1     module display(
2   input clk,rst_n,
3   input [15:0] reg_0,reg_1,reg_2,reg_3,reg_4,reg_5,reg_6,reg_7,reg_8
        ,reg_9,reg_10,reg_11,reg_12,reg_13,reg_14,reg_15,
4   input [7:0] ctl,
5   output reg [7:0] disp_1,disp_2,disp_3,disp_4,disp_5,disp_6,disp_7,
        disp_8,
6   output [8:0] sl_out);
7   wire [15:0] wire_reg0,wire_reg1,wire_reg2,wire_reg3,wire_reg4,
        wire_reg5,wire_reg6,wire_reg7,wire_reg8,wire_reg9,wire_reg10,
        wire_reg11,wire_reg12,wire_reg13,wire_reg14,wire_reg15;
8   wire [7:0] disp_reg0_1,disp_reg0_2,disp_reg0_3,disp_reg0_4,
        disp_reg1_1,disp_reg1_2,disp_reg1_3,disp_reg1_4,disp_reg2_1,
        disp_reg2_2,disp_reg2_3,disp_reg2_4,disp_reg3_1,disp_reg3_2,
        disp_reg3_3,disp_reg3_4,disp_reg4_1,disp_reg4_2,disp_reg4_3,
        disp_reg4_4,disp_reg5_1,disp_reg5_2,disp_reg5_3,disp_reg5_4,
        disp_reg6_1,disp_reg6_2,disp_reg6_3,disp_reg6_4,disp_reg7_1,
        disp_reg7_2,disp_reg7_3,disp_reg7_4,disp_reg8_1,disp_reg8_2,
        disp_reg8_3,disp_reg8_4,disp_reg9_1,disp_reg9_2,disp_reg9_3,
        disp_reg9_4,disp_reg10_1,disp_reg10_2,disp_reg10_3,
        disp_reg10_4,disp_reg11_1,disp_reg11_2,disp_reg11_3,
        disp_reg11_4,disp_reg12_1,disp_reg12_2,disp_reg12_3,
        disp_reg12_4,disp_reg13_1,disp_reg13_2,disp_reg13_3,
        disp_reg13_4,disp_reg14_1,disp_reg14_2,disp_reg14_3,
```

```
        disp_reg14_4 , disp_reg15_1 , disp_reg15_2 , disp_reg15_3 ,
        disp_reg15_4 ;
 9  wire sl_clk_wire , sl_rst_wire ;
10  reg [5:0] t;
11  reg [8:0] sel;
12  assign wire_reg0 = reg_0;
13  assign wire_reg1 = reg_1;
14  assign wire_reg2 = reg_2;
15  assign wire_reg3 = reg_3;
16  assign wire_reg4 = reg_4;
17  assign wire_reg5 = reg_5;
18  assign wire_reg6 = reg_6;
19  assign wire_reg7 = reg_7;
20  assign wire_reg8 = reg_8;
21  assign wire_reg9 = reg_9;
22  assign wire_reg10 = reg_10;
23  assign wire_reg11 = reg_11;
24  assign wire_reg12 = reg_12;
25  assign wire_reg13 = reg_13;
26  assign wire_reg14 = reg_14;
27  assign wire_reg15 = reg_15;
28  assign sl_clk_wire = clk;
29  assign sl_rst_wire = rst_n;
30  number reg0(.data_sig(wire_reg0), .disp_out1(disp_reg0_1), .
        disp_out2(disp_reg0_2), .disp_out3(disp_reg0_3), .disp_out4(
        disp_reg0_4));
31  number reg1(.data_sig(wire_reg1), .disp_out1(disp_reg1_1), .
        disp_out2(disp_reg1_2), .disp_out3(disp_reg1_3), .disp_out4(
        disp_reg1_4));
32  number reg2(.data_sig(wire_reg2), .disp_out1(disp_reg2_1), .
        disp_out2(disp_reg2_2), .disp_out3(disp_reg2_3), .disp_out4(
        disp_reg2_4));
33  number reg3(.data_sig(wire_reg3), .disp_out1(disp_reg3_1), .
        disp_out2(disp_reg3_2), .disp_out3(disp_reg3_3), .disp_out4(
        disp_reg3_4));
34  number reg4(.data_sig(wire_reg4), .disp_out1(disp_reg4_1), .
        disp_out2(disp_reg4_2), .disp_out3(disp_reg4_3), .disp_out4(
        disp_reg4_4));
35  number reg5(.data_sig(wire_reg5), .disp_out1(disp_reg5_1), .
        disp_out2(disp_reg5_2), .disp_out3(disp_reg5_3), .disp_out4(
        disp_reg5_4));
36  number reg6(.data_sig(wire_reg6), .disp_out1(disp_reg6_1), .
        disp_out2(disp_reg6_2), .disp_out3(disp_reg6_3), .disp_out4(
        disp_reg6_4));
37  number reg7(.data_sig(wire_reg7), .disp_out1(disp_reg7_1), .
        disp_out2(disp_reg7_2), .disp_out3(disp_reg7_3), .disp_out4(
```

```
        disp_reg7_4));
38  number reg8(.data_sig(wire_reg8), .disp_out1(disp_reg8_1), .
        disp_out2(disp_reg8_2), .disp_out3(disp_reg8_3), .disp_out4(
        disp_reg8_4));
39  number reg9(.data_sig(wire_reg9), .disp_out1(disp_reg9_1), .
        disp_out2(disp_reg9_2), .disp_out3(disp_reg9_3), .disp_out4(
        disp_reg9_4));
40  number reg10(.data_sig(wire_reg10), .disp_out1(disp_reg10_1), .
        disp_out2(disp_reg10_2), .disp_out3(disp_reg10_3), .disp_out4(
        disp_reg10_4));
41  number reg11(.data_sig(wire_reg11), .disp_out1(disp_reg11_1), .
        disp_out2(disp_reg11_2), .disp_out3(disp_reg11_3), .disp_out4(
        disp_reg11_4));
42  number reg12(.data_sig(wire_reg12), .disp_out1(disp_reg12_1), .
        disp_out2(disp_reg12_2), .disp_out3(disp_reg12_3), .disp_out4(
        disp_reg12_4));
43  number reg13(.data_sig(wire_reg13), .disp_out1(disp_reg13_1), .
        disp_out2(disp_reg13_2), .disp_out3(disp_reg13_3), .disp_out4(
        disp_reg13_4));
44  number reg14(.data_sig(wire_reg14), .disp_out1(disp_reg14_1), .
        disp_out2(disp_reg14_2), .disp_out3(disp_reg14_3), .disp_out4(
        disp_reg14_4));
45  number reg15(.data_sig(wire_reg15), .disp_out1(disp_reg15_1), .
        disp_out2(disp_reg15_2), .disp_out3(disp_reg15_3), .disp_out4(
        disp_reg15_4));
46  assign sl_out = sel;
47  reg [25:0] count;
48  always @(posedge clk or negedge rst_n) begin
49   if (!rst_n)
50    count <= 26'h0;
51   else if (count == (26'd20_000_000 / 300))
52    count <= 26'h0;
53   else
54    count <= count + 26'h1;
55  end
56  always @(posedge clk or negedge rst_n) begin
57   if (!rst_n)
58    t <= 4'd0000;
59   else if (count == (26'd10_000_000 / 300)) begin
60    t <= (t + 1) % 36;
61     sel[7] <= (t == 2) ? 1:
62         (t == 4) ? 0:
63         sel[7];
64     sel[6] <= (t == 6) ? 1:
65         (t == 8) ? 0:
66         sel[6];
```

```
67      sel[5] <= (t == 10) ? 1:
68          (t == 12) ? 0:
69          sel[5];
70      sel[4] <= (t == 14) ? 1:
71          (t == 16) ? 0:
72          sel[4];
73      sel[3] <= (t == 18) ? 1:
74          (t == 20) ? 0:
75          sel[3];
76      sel[2] <= (t == 22) ? 1:
77          (t == 24) ? 0:
78          sel[2];
79      sel[1] <= (t == 26) ? 1:
80          (t == 28) ? 0:
81          sel[1];
82      sel[0] <= (t == 30) ? 1:
83          (t == 32) ? 0:
84          sel[0];
85      sel[8] <= (t == 34) ? 1:
86          (t == 0) ? 0:
87          sel[8];
88      disp_1 <=
89        (t == 1)? disp_reg0_4:
90        (t == 5)? disp_reg2_4:
91        (t == 9)? disp_reg4_4:
92        (t == 13)? disp_reg6_4:
93        (t == 17)? disp_reg8_4:
94        (t == 21)? disp_reg10_4:
95        (t == 25)? disp_reg12_4:
96        (t == 29)? disp_reg14_4:
97        (t == 34)? ctl:
98        disp_1;
99      disp_2 <=
100       (t == 1)? disp_reg0_3:
101       (t == 5)? disp_reg2_3:
102       (t == 9)? disp_reg4_3:
103       (t == 13)? disp_reg6_3:
104       (t == 17)? disp_reg8_3:
105       (t == 21)? disp_reg10_3:
106       (t == 25)? disp_reg12_3:
107       (t == 29)? disp_reg14_3:
108       disp_2;
109     disp_3 <=
110       (t == 1)? disp_reg0_2:
111       (t == 5)? disp_reg2_2:
112       (t == 9)? disp_reg4_2:
```

```
113         (t == 13)? disp_reg6_2:
114         (t == 17)? disp_reg8_2:
115         (t == 21)? disp_reg10_2:
116         (t == 25)? disp_reg12_2:
117         (t == 29)? disp_reg14_2:
118         disp_3;
119       disp_4 <=
120         (t == 1)? disp_reg0_1:
121         (t == 5)? disp_reg2_1:
122         (t == 9)? disp_reg4_1:
123         (t == 13)? disp_reg6_1:
124         (t == 17)? disp_reg8_1:
125         (t == 21)? disp_reg10_1:
126         (t == 25)? disp_reg12_1:
127         (t == 29)? disp_reg14_1:
128         disp_4;
129       disp_5 <=
130         (t == 1)? disp_reg1_4:
131         (t == 5)? disp_reg3_4:
132         (t == 9)? disp_reg5_4:
133         (t == 13)? disp_reg7_4:
134         (t == 17)? disp_reg9_4:
135         (t == 21)? disp_reg11_4:
136         (t == 25)? disp_reg13_4:
137         (t == 29)? disp_reg15_4:
138         disp_5;
139       disp_6 <=
140         (t == 1)? disp_reg1_3:
141         (t == 5)? disp_reg3_3:
142         (t == 9)? disp_reg5_3:
143         (t == 13)? disp_reg7_3:
144         (t == 17)? disp_reg9_3:
145         (t == 21)? disp_reg11_3:
146         (t == 25)? disp_reg13_3:
147         (t == 29)? disp_reg15_3:
148         disp_6;
149       disp_7 <=
150         (t == 1)? disp_reg1_2:
151         (t == 5)? disp_reg3_2:
152         (t == 9)? disp_reg5_2:
153         (t == 13)? disp_reg7_2:
154         (t == 17)? disp_reg9_2:
155         (t == 21)? disp_reg11_2:
156         (t == 25)? disp_reg13_2:
157         (t == 29)? disp_reg15_2:
158         disp_7;
```

```
159       disp_8 <=
160         (t == 1)? disp_reg1_1:
161         (t == 5)? disp_reg3_1:
162         (t == 9)? disp_reg5_1:
163         (t == 13)? disp_reg7_1:
164         (t == 17)? disp_reg9_1:
165         (t == 21)? disp_reg11_1:
166         (t == 25)? disp_reg13_1:
167         (t == 29)? disp_reg15_1:
168         disp_8;
169     end else begin
170       t <= t;
171       sel <= sel;
172       disp_1 <= disp_1;
173       disp_2 <= disp_2;
174       disp_3 <= disp_3;
175       disp_4 <= disp_4;
176       disp_5 <= disp_5;
177       disp_6 <= disp_6;
178       disp_7 <= disp_7;
179       disp_8 <= disp_8;
180     end
181   end
182 endmodule
183
184 module SEVENSEG_LED (
185   input [3:0] a,
186   output [7:0] output_signal);
187   assign output_signal = (a == 4'b0000) ? 8'b1111_1100:
188             (a == 4'b0001) ? 8'b0110_0000:
189             (a == 4'b0010) ? 8'b1101_1010:
190             (a == 4'b0011) ? 8'b1111_0010:
191             (a == 4'b0100) ? 8'b0110_0110:
192             (a == 4'b0101) ? 8'b1011_0110:
193             (a == 4'b0110) ? 8'b1011_1110:
194             (a == 4'b0111) ? 8'b1110_0000:
195             (a == 4'b1000) ? 8'b1111_1110:
196             (a == 4'b1001) ? 8'b1111_0110:
197             (a == 4'b1010) ? 8'b1110_1110:
198             (a == 4'b1011) ? 8'b0011_1110:
199             (a == 4'b1100) ? 8'b0001_1010:
200             (a == 4'b1101) ? 8'b0111_1010:
201             (a == 4'b1110) ? 8'b1001_1110:
202             8'b1000_1110;
203
204 endmodule
```

```
205
206 module number(
207  input [15:0] data_sig,
208  output [7:0] disp_out1,disp_out2,disp_out3,disp_out4);
209  wire n_wire_clk,n_wire_rst;
210  wire [7:0] disp_wire1,disp_wire2,disp_wire3,disp_wire4;
211  wire [3:0] data_wire1,data_wire2,data_wire3,data_wire4;
212  assign data_wire1 = data_sig [3:0]; //___O
213  assign data_wire2 = data_sig [7:4]; //__O_
214  assign data_wire3 = data_sig [11:8]; //_O__
215  assign data_wire4 = data_sig [15:12]; //O___
216  SEVENSEG_LED l1(.a(data_wire1), .output_signal(disp_wire1));
217  SEVENSEG_LED l2(.a(data_wire2), .output_signal(disp_wire2));
218  SEVENSEG_LED l3(.a(data_wire3), .output_signal(disp_wire3));
219  SEVENSEG_LED l4(.a(data_wire4), .output_signal(disp_wire4));
220  assign disp_out1 =  disp_wire1;
221  assign disp_out2 =  disp_wire2;
222  assign disp_out3 =  disp_wire3;
223  assign disp_out4 =  disp_wire4;
224
225 endmodule
```

ある7SEGのLEDを4個ずつに分けてレジスタの値などを出力することにした。レジスタの値などが分かりやすくなり、アッセンブリーコードのデバッグがさらに簡単にできる。設定したのは以下の通りである。

```
1      display ds(.clk(clk20),.rst_n(rst_n),
2        .reg_0(reg_0),.reg_1(reg_1),.reg_2(reg_2),.reg_3(reg_3),.
             reg_4(reg_4),.reg_5(reg_5),.reg_6(reg_6),.reg_7(reg_7),
3        .reg_8(reg_8),.reg_9(reg_9),.reg_10(reg_10),.reg_11(reg_11),.
             reg_12(reg_12),.reg_13(reg_13),.reg_14(reg_14),.reg_15(
             reg_15),
4        .ctl(ctlcheck),
5        .disp_1(disp_1),.disp_2(disp_2),.disp_3(disp_3),.disp_4(
             disp_4),.disp_5(disp_5),.disp_6(disp_6),.disp_7(disp_7),.
             disp_8(disp_8),.sl_out(sl_out2));
6
7              assign reg_8 = IROut;
8    assign reg_9 = DROut;
9    assign reg_10 = opcodeout;
10   assign reg_11 = MDROut;
11   assign reg_12 = shifterIn;
12   assign reg_13 = dshiftout;
13   assign reg_14 = count[31:16];
14   assign reg_15 = count[15:0];
```

1.6 ソーティングアルゴリズム

Radixソートを8ビットずつやった。ある8ビットの値、それぞれの数をメモリーに追加し、追加して足し算した値からアドレスを計算している。それを2回やり、2回目はアドレスの値を1の値が最初から来るようにした。それでサイクル数が26万ぐらいから6万ぐらいまで減った。それにsorted、r-sorted検知を入れた。

```
 1  SLI 0,1,10
 2  SLI 1,1,11
 3  LD 4,0,0
 4  LD 5,1,0
 5  ADDI 0,1
 6  CMP 4,5
 7  BLT 2
 8  BE 29
 9  B 9
10  LD 4,0,0
11  LD 5,1,0
12  ADDI 0,1
13  CMP 0,1
14  BNE 1
15  HLT
16  CMP 4,5
17  BLE -8
18  B 29
19  LD 4,0,0
20  LD 5,1,0
21  ADDI 0,1
22  CMP 0,1
23  BNE 11
24  SLI 0,1,10
25  SUBI 1,1
26  LD 4,0,0
27  LD 5,0,1
28  ST 4,0,1
29  ST 5,0,0
30  ADDI 0,1
31  SUBI 1,1
32  CMP 0,1
33  BLE -8
34  HLT
35  CMP 5,4
36  BLE -18
37  B 10
38  LD 4,0,0
```

```
39  LD 5,1,0
40  ADDI 0,1
41  CMP 0,1
42  BNE 1
43  HLT
44  CMP 4,5
45  BE -8
46  BLE -37
47  B -29
48  SLI 2,1,8
49  LI 0,0
50  LI 1,0
51  ST 0,0,1
52  ADDI 1,1
53  CMP 1,2
54  BLT -4
55  SUBI 2,1
56  MOV 0,2
57  SLI 1,1,10
58  SLI 3,1,11
59  SUBI 3,1
60  LD 4,0,1
61  AND 4,0
62  LD 5,1,4
63  ADDI 5,1
64  ST 5,1,4
65  CMP 1,3
66  ADDI 1,1
67  BLT -8
68  LI 0,0
69  SLI 5,1,11
70  LD 6,0,0
71  ADD 6,5
72  ST 6,0,0
73  ADDI 0,1
74  LD 5,0,0
75  ADD 6,5
76  ST 6,0,0
77  CMP 0,2
78  BLT -6
79  SLI 1,1,10
80  SLI 3,1,11
81  SUBI 3,1
82  LD 4,0,1
83  LD 6,0,1
84  AND 4,0
```

```
 85  LD 5,0,4
 86  ST 6,0,5
 87  ADDI 5,1
 88  ST 5,0,4
 89  CMP 1,3
 90  ADDI 1,1
 91  BLT -10
 92  ADDI 2,2
 93  LI 0,0
 94  LI 1,0
 95  ST 0,0,1
 96  ADDI 1,1
 97  CMP 1,2
 98  BLT -4
 99  SUBI 2,2
100  MOV 0,2
101  SLI 1,1,11
102  SLI 3,1,10
103  SUBI 3,1
104  ADD 3,1
105  LD 4,0,1
106  SRL 4,8
107  AND 4,0
108  LD 5,0,4
109  ADDI 5,1
110  ST 5,0,4
111  CMP 1,3
112  ADDI 1,1
113  BLT -9
114  SLI 0,1,7
115  SLI 5,1,10
116  LD 6,0,0
117  ADD 6,5
118  ST 5,0,0
119  ADDI 0,1
120  LD 5,0,0
121  ST 6,0,0
122  ADD 6,5
123  CMP 0,2
124  BLT -6
125  SLI 1,1,7
126  SUBI 1,1
127  LI 0,0
128  LD 5,0,0
129  ST 6,0,0
130  ADDI 0,1
```

```
131  ADD 6,5
132  LD 5,0,0
133  ST 6,0,0
134  CMP 0,1
135  BLT -6
136  SLI 1,1,11
137  SLI 3,1,10
138  SUBI 3,1
139  ADD 3,1
140  LD 4,0,1
141  SRL 4,8
142  LD 6,0,1
143  AND 4,2
144  LD 5,0,4
145  ST 6,0,5
146  ADDI 5,1
147  ST 5,0,4
148  CMP 1,3
149  ADDI 1,1
150  BLT -11
151  HLT
```