

コンピュータアーキテクチャ最終レポート

杉山月渚 (03-230426)

July 27, 2025

1 課題 1

「プロセッサを HDL で書いてみよう – シングルサイクル・プロセッサで OK」について以下にまとめる。単一サイクル構成の RISC-V プロセッサの実装における各 Verilog ファイルの役割と、その動作を示す波形画像について解説する。

1.1 回路構成と各モジュールの役割

本プロセッサは以下のコンポーネントから構成されている [2]：

1.1.1 riscv_single_cycle.v

CPU 全体のトップモジュールであり、各コンポーネントを接続する役割を果たす。クロックとりセットを入力とし、プログラムカウンタ、命令メモリ、レジスタファイル、ALU、制御ユニットなどを結合する。

```
1 // riscv_single_cycle.v
2 // シングルサイクル RISC-V プロセッサのトップモジュール
3
4 module riscv_single_cycle (
5     input wire clk,          // クロック
6     input wire reset        // リセット
7 );
8
9     // 内部信号の宣言
10    wire [31:0] pc;           // プログラムカウンタ
11    wire [31:0] instruction;  // 命令
12    wire [31:0] read_data1;   // レジスタファイルからの読み出しデータ1
13    wire [31:0] read_data2;   // レジスタファイルからの読み出しデータ2
14    wire [31:0] extended_immediate; // 符号拡張された即値
15    wire [31:0] alu_result;    // ALUの出力
16    reg [31:0] write_data;     // レジスタファイルへの書き込みデータ <---
17    // CHANGED FROM 'wire' TO 'reg'
18    wire [31:0] mem_read_data; // データメモリからの読み出しデータ wire [4:0]
19    wire [4:0] rs1_addr;       // ソースレジスタ1アドレス
20    wire [4:0] rs2_addr;       // ソースレジスタ2アドレス
21    wire [4:0] rd_addr;        // デスティネーションレジスタアドレス
22    wire reg_write_en;         // レジスタ書き込みイネーブル
23    wire [2:0] alu_op;          // ALU操作コード
24    wire alu_src_b;             // ALUのB入力選択 (0: Reg, 1: Immediate)
25    wire [1:0] mem_to_reg;      // メモリからレジスタへのデータ選択
26    wire mem_read_en;           // データメモリ読み出しイネーブル // NEW: Added
27    // for data memory control
28    wire mem_write_en;          // データメモリ書き込みイネーブル // NEW: Added
29    // for data memory control
30    wire branch_taken;
31    wire [31:0] imm_branch;
32    wire [31:0] next_pc;
```

```

30 wire [31:0] imm_u_type;
31 wire [31:0] imm_j_type;
32
33 // プログラムカウンタ (PC)
34 // リセット時は0、それ以外はPC + 4
35 pc_unit pc_unit_inst (
36     .clk(clk),
37     .reset(reset),
38     .next_pc_in(next_pc), // calculated next PC
39     .pc_out(pc)           // current PC output
40 );
41
42 // 命令メモリ (Instruction Memory)
43 instruction_memory im_inst (
44     .addr(pc),
45     .instruction_out(instruction)
46 );
47
48 always @(posedge clk) begin
49     $display("DEBUG: Time=%0t | PC=%h | imm_branch=%h | branch_taken=%b",
50             $time, pc, imm_branch, branch_taken);
51 end
52
53
54 // 命令デコード
55 // 命令から各フィールドを抽出
56 assign rs1_addr = instruction[19:15];
57 assign rs2_addr = instruction[24:20];
58 assign rd_addr  = instruction[11:7];
59
60 // GTKWAVEチェックのため
61 wire [31:0] x0_debug, x1_debug, x2_debug, x3_debug, x4_debug, x5_debug, x6_debug
62 ;
63
64 // レジスタファイル
65 // rs1_addrとrs2_addrからデータを読み出し、rd_addrにwrite_dataを書き込む
66 register_file rf_inst (
67     .clk(clk),
68     .reg_write_en(reg_write_en),
69     .rs1_addr(rs1_addr),
70     .rs2_addr(rs2_addr),
71     .rd_addr(rd_addr),
72     .write_data(write_data),
73     .read_data1(read_data1),
74     .read_data2(read_data2),
75     .x0_debug(x0_debug),
76     .x1_debug(x1_debug),
77     .x2_debug(x2_debug),
78     .x3_debug(x3_debug),
79     .x4_debug(x4_debug),
80     .x5_debug(x5_debug),
81     .x6_debug(x6_debug)
82 );
83
84 // 即値生成ユニット (Sign Extender)
85 // Iタイプ命令の即値を32ビットに符号拡張
86 // 拡張性のため、他のタイプの即値生成もここに追加
87 sign_extender se_inst (
88     .instruction(instruction),
89     .extended_immediate(extended_immediate),
90     .imm_branch(imm_branch),
91     .imm_u_type(imm_u_type),
92     .imm_j_type(imm_j_type)

```

```

92 );
93
94 always @(posedge clk) begin
95     $display("DEBUG: Time=%0t | PC=%h | imm_branch=%h | branch_taken=%b",
96             $time, pc, imm_branch, branch_taken);
97 end
98
99
100 // ALU (Arithmetic Logic Unit)
101 // alu_src_bによってread_data2かextended_immediateを選択し演算
102 alu alu_inst (
103     .src_a(read_data1),
104     .src_b(alu_b_input), // MUX for ALU B input
105     .alu_op(alu_op),
106     .result(alu_result)
107 );
108
109 // B-type
110 wire [6:0] opcode = instruction[6:0];
111 wire [2:0] funct3 = instruction[14:12];
112 localparam OP_BRANCH = 7'b1100011;
113
114 assign branch_taken = (opcode == OP_BRANCH) && (
115     (funct3 == 3'b000 && alu_result == 0) || // beq
116     (funct3 == 3'b001 && alu_result != 0)    // bne
117 );
118
119 // J-type
120 wire jump_en = (opcode == 7'b1101111); // jal
121 assign next_pc =
122     jump_en      ? pc + imm_j_type :
123     branch_taken ? pc + imm_branch :
124     pc + 4;
125
126 always @(posedge clk) begin
127     $display("DEBUG: Time=%0t | PC=%h | imm_branch=%h | branch_taken=%b | next_pc
128             =%h",
129             $time, pc, imm_branch, branch_taken, next_pc);
130 end
131
132 // U-type
133 localparam OP_LUI = 7'b0110111;
134 wire [31:0] alu_b_input = (opcode == OP_LUI) ? imm_u_type :
135     (alu_src_b ? extended_immediate : read_data2);
136
137
138 // データメモリ (Data Memory) - Rタイプでは使用しないが、I/Sタイプで必要
139 // 最初はダミーで置いておく
140 data_memory dm_inst (
141     .clk(clk),
142     .addr(alu_result), // ロード/ストアアドレス
143     .write_data(read_data2), // スタアデータ
144     // .mem_read_en(1'b0), // 後で制御ユニットから接続
145     // .mem_write_en(1'b0), // 後で制御ユニットから接続
146     // .read_data_out() // ロードデータ (未使用だがポートは保持)
147     .mem_read_en(mem_read_en), // 制御ユニットから接続 // NEW: Connected
148     .mem_write_en(mem_write_en), // 制御ユニットから接続 // NEW: Connected
149     .read_data_out(mem_read_data) // ロードデータ // NEW: Connected to new wire
150 );
151
152 // ライトバックステージのMux // MODIFIED: Now a MUX
153 // mem_to_reg 信号に基づいて、ALUの結果またはメモリからの読み出しデータを

```

```

        write_data として選択
154 // mem_to_reg:
155 // 00: ALU結果をレジスタに書き込む (R-type, I-type (addi))
156 // 01: メモリからのデータをレジスタに書き込む (lw)
157 // (他の値は将来の拡張用、例: PC+4 for JAL/JALR)
158 always @* begin
159     case (mem_to_reg)
160         2'b00: write_data = (jump_en ? pc + 4 : alu_result);
161         2'b01: write_data = mem_read_data;
162         default: write_data = 32'b0;
163     endcase
164 end
165 // // alu_resultをwrite_dataとして選択。後でメモリロードデータも選択肢に追加
166 // assign write_data = alu_result;
167
168 // 制御ユニット (Control Unit)
169 // 命令から制御信号を生成
170 control_unit cu_inst (
171     .opcode(instruction[6:0]),
172     .funct3(instruction[14:12]),
173     .funct7(instruction[31:25]),
174     .reg_write_en(reg_write_en),
175     .alu_op(alu_op),
176     .alu_src_b(alu_src_b),
177     .mem_read_en(mem_read_en), // Connected
178     .mem_write_en(mem_write_en), // Connected
179     .mem_to_reg(mem_to_reg) // Connected // 他の制御信号 (Branch, Jump,
        PC_Sourceなど) は必要に応じて追加
180 );
181
182 endmodule

```

Listing 1: CPU トップモジュール

1.1.2 pc_unit.v

プログラムカウンタ (PC) を保持し、分岐やジャンプ命令に応じて次の命令アドレスを供給する。

```

1 // // pc_unit.v
2 // // プログラムカウンタ
3
4 // module pc_unit (
5 //     input wire clk,
6 //     input wire reset,
7
8 //     input wire [31:0] imm_branch, // 分岐・ジャンプのオフセット (符号付き)
9 //     input wire branch_taken, // beq, bne が成功した場合 1
10 //     input wire jump_en, // jal, jalr などジャンプ命令が有効な場合 1
11
12 //     output reg [31:0] pc_out // 現在のPC
13 // );
14
15 //     wire [31:0] pc_plus4 = pc_out + 4;
16 //     wire [31:0] pc_branch_target = pc_out + imm_branch;
17
18 //     wire [31:0] next_pc =
19 //         (branch_taken || jump_en) ? pc_branch_target : pc_plus4;
20
21 //     always @(posedge clk or posedge reset) begin
22 //         if (reset) begin
23 //             pc_out <= 32'h00000000;
24 //         end else begin
25 //             pc_out <= next_pc;

```

```

26 //      end
27 //      end
28
29 // endmodule
30 // pc_unit.v
31
32 module pc_unit (
33     input wire clk,
34     input wire reset,
35     input wire [31:0] next_pc_in,
36     output reg [31:0] pc_out
37 );
38
39     always @(posedge clk or posedge reset) begin
40         if (reset)
41             pc_out <= 32'h00000000;
42         else
43             pc_out <= next_pc_in;
44     end
45
46 endmodule

```

Listing 2: PC 管理ユニット

1.1.3 instruction_memory.v

命令ROMとして動作し、プログラムカウンタから指定された命令を返す。mem 配列にテスト用命令が事前に格納されている。

```

1 // instruction_memory.v
2 // 命令メモリ (ROMとして実装)
3
4 module instruction_memory (
5     input wire [31:0] addr,
6     output wire [31:0] instruction_out
7 );
8
9     // 1024ワード (4KB) のメモリを想定
10    reg [31:0] mem [0:1023];
11
12    initial begin
13        // テスト用の命令
14        // I命令: add
15        // 例: addi x1, x0, 1 (0x00100093)
16        //      addi x2, x0, 2 (0x00200113)
17        //      add x3, x1, x2 (0x002081B3)
18        //      sub x4, x3, x1 (0x40118233)
19        // アドレスはワードアドレス (バイトアドレス/4) で指定
20        // mem[0] = 32'h00100093; // addi x1, x0, 1
21        // mem[1] = 32'h00200113; // addi x2, x0, 2
22        // mem[2] = 32'h002081B3; // add x3, x1, x2 (x3 = x1 + x2 = 1 + 2 = 3)
23        // mem[3] = 32'h40118233; // sub x4, x3, x1 (x4 = x3 - x1 = 3 - 1 = 2)
24
25        // テスト用の命令 [funct7][rs2][rs1][funct3][rd][opcode]
26        // I-type
27        mem[0] = 32'b0000000000001_00000_000_00001_0010011; // addi x1, x0, 1 →
28                    x1 = 1
29        mem[1] = 32'b0000000000010_00000_000_00010_0010011; // addi x2, x0, 2 →
29                    x2 = 2
30
31        // R-type
32        mem[2] = 32'b00000000_00010_00001_000_00011_0110011; // add x3, x1, x2 →
33                    x3 = x1 + x2 = 3

```

```

32     mem[3] = 32'b0100000_00001_00011_000_00100_0110011; // sub x4, x3, x1 →
        x4 = x3 - x1 = 2
33     mem[4] = 32'b0000000_00010_00011_100_00101_0110011; // xor x5, x3, x2 →
        x5 = x3 ^ x2 = 1
34     mem[5] = 32'b0000000_00010_00011_110_00110_0110011; // or x6, x3, x2 →
        x6 = x3 | x2 = 3
35     mem[6] = 32'b0000000_00010_00011_111_00111_0110011; // and x7, x3, x2 →
        x7 = x3 & x2 = 2
36     mem[7] = 32'b0000000_00010_00011_001_01000_0110011; // sll x8, x3, x2 →
        x8 = x3 << x2 = 12
37     mem[8] = 32'b0000000_00010_00011_101_01001_0110011; // srl x9, x3, x2 →
        x9 = x3 >> x2 = 0
38
39     // I-type (load, shift)
40     // mem[8] = 32'b0000000000000_00010_111_00101_0000011; // lw x5, 0(x8)
        → x5 = MEM[x8+0]
41     // mem[4] = 32'b0000000000010_00001_001_01000_0010011; // slli x8, x1, 2
        → x8 = x1 << 2 = 4
42     // mem[5] = 32'b010000000010_00001_101_01001_0010011; // srai x9, x1, 2
        → x9 = x1 >>> 2 = 0
43
44     // // S-type
45     // mem[5] = 32'b0000000_00101_00010_010_00100_0100011; // sw x5, 4(x2)
        → MEM[x2+0] = x5
46     // mem[6] = 32'b000000000100_00010_010_00110_0000011; // lw x6, 4(x2)
        → x6 = MEM[x2+4]
47
48     // B-type
49     // mem[5] = 32'b0000000_00011_00010_000_01000_1100011; // beq x2, x3, +8
        → branch not taken
50     // mem[6] = 32'b0000000_00011_00010_001_01000_1100011; // bne x2, x3, +8
        → branch taken
51
52     // // U-type
53     // mem[7] = 32'b0000000000000000000001_00101_0110111; // lui x5, 0x0
        → x5 = 0x00000000
54     // mem[8] = 32'b00010010001101000101_00001_0110111; // lui x1, 0x12345 →
        x13 = 0x12345000
55
56     // // J-type
57     // mem[5] = 32'h002000ef; // jal x1, 4
58 end
59
60     assign instruction_out = mem[addr[31:2]]; // バイトアドレスからワードアドレスに
        変換
61
62 endmodule

```

Listing 3: 命令メモリ

1.1.4 register_file.v

32本のレジスタを持ち、読み出し・書き込み操作を制御信号に基づいて実行する。デバッグ用にx0～x6の出力も提供している。

```

1 // register_file.v
2 // RISC-V レジスタファイル
3
4 module register_file (
5     input wire clk,
6     input wire reg_write_en, // レジスタ書き込みイネーブル
7     input wire [4:0] rs1_addr, // 読み出しアドレス1
8     input wire [4:0] rs2_addr, // 読み出しアドレス2

```

```

9      input wire [4:0] rd_addr,      // 書き込みアドレス
10     input wire [31:0] write_data, // 書き込みデータ
11     output wire [31:0] read_data1,
12     output wire [31:0] read_data2,
13     output wire [31:0] x0_debug, x1_debug, x2_debug, x3_debug, x4_debug, x5_debug,
        x6_debug
14 );
15
16     reg [31:0] registers [0:31]; // 32個の32ビットレジスタ
17
18     // 初期化 (オプション: シミュレーション用に0クリア)
19     integer i;
20     initial begin
21         for (i = 0; i < 32; i = i + 1) begin
22             registers[i] = 32'h00000000;
23         end
24     end
25
26     // 読み出しポート (組み合わせ回路)
27     // x0 レジスタは常に0
28     assign read_data1 = (rs1_addr == 5'h00) ? 32'h00000000 : registers[rs1_addr];
29     assign read_data2 = (rs2_addr == 5'h00) ? 32'h00000000 : registers[rs2_addr];
30
31     // 書き込みポート (同期回路)
32     always @(posedge clk) begin
33         if (reg_write_en) begin
34             if (rd_addr != 5'h00) begin // x0 レジスタには書き込まない
35                 registers[rd_addr] <= write_data;
36             end
37         end
38     end
39
40     assign x0_debug = registers[0];
41     assign x1_debug = registers[1];
42     assign x2_debug = registers[2];
43     assign x3_debug = registers[3];
44     assign x4_debug = registers[4];
45     assign x5_debug = registers[5];
46     assign x6_debug = registers[6];
47
48 endmodule

```

Listing 4: レジスタファイル

1.1.5 alu.v

加算・減算・論理演算などを行う演算器。alu_op 信号により動作内容が決定される。

```

1 // alu.v
2 // 算術論理ユニット (ALU)
3
4 module alu (
5     input wire [31:0] src_a,
6     input wire [31:0] src_b,
7     input wire [2:0] alu_op, // 制御ユニットからのALU操作コード
8     output reg [31:0] result
9     // output wire zero_flag // 将来的にゼロフラグなども追加
10 );
11
12 // ALU操作コードの定義 (例)
13 localparam ALU_ADD = 3'b000; // 加算
14 localparam ALU_SUB = 3'b001; // 減算
15 localparam ALU_AND = 3'b010; // 論理積

```

```

16     localparam ALU_OR    = 3'b011; // 論理和
17     localparam ALU_XOR   = 3'b100; // 排他的論理和
18     localparam ALU_SLL   = 3'b101;
19     localparam ALU_SRL   = 3'b110;
20     localparam ALU_SRA   = 3'b111;
21     // 他の命令 (SLL, SRL, SRA, SLT, SLTUなど) は必要に応じて追加
22
23     always @(*) begin
24         case (alu_op)
25             ALU_ADD: result = src_a + src_b;
26             ALU_SUB: result = src_a - src_b;
27             ALU_AND: result = src_a & src_b;
28             ALU_OR:  result = src_a | src_b;
29             ALU_XOR: result = src_a ^ src_b;
30             ALU_SLL: result = src_a << src_b[4:0];
31             ALU_SRL: result = src_a >> src_b[4:0];
32             ALU_SRA: result = $signed(src_a) >>> src_b[4:0];
33             default: result = 32'hxxxxxxxx; // 未定義の場合
34         endcase
35     end
36
37     // assign zero_flag = (result == 32'h00000000) ? 1'b1 : 1'b0;
38
39 endmodule

```

Listing 5: ALU

1.1.6 control_unit.v

命令の opcode, funct3, funct7 から各種制御信号（レジスタ書き込み許可、ALU 操作、分岐、メモリ読み書きなど）を生成する。

```

1  // control_unit.v
2  // 制御ユニット
3
4  module control_unit (
5      input wire [6:0] opcode,
6      input wire [2:0] funct3,
7      input wire [6:0] funct7,
8      output reg reg_write_en, // レジスタファイル書き込みイネーブル
9      output reg [2:0] alu_op, // ALU操作コード
10     output reg alu_src_b,    // ALUのB入力選択 (0: Reg, 1: Immediate)
11     output reg mem_read_en,  // データメモリ読み出しイネーブル
12     output reg mem_write_en, // データメモリ書き込みイネーブル
13     output reg [1:0] mem_to_reg // メモリからレジスタへのデータ選択
14     // 他の制御信号 (Branch, Jump, PC_Sourceなど) は必要に応じて追加
15 );
16
17 // RISC-V Opcode の定義 (一部)
18 localparam OP_R_TYPE = 7'b0110011; // Rタイプ命令 (add, sub, and, or, xorなど)
19 localparam OP_IMM    = 7'b0010011; // Iタイプ即値命令 (addi, andiなど)
20 localparam OP_LOAD    = 7'b0000011; // ロード命令 (lwなど)
21 localparam OP_STORE   = 7'b0100011; // ストア命令 (swなど)
22 localparam OP_BRANCH  = 7'b1100011; // Bタイプ (beq, bne など)
23 localparam OP_LUI     = 7'b0110111; // Uタイプ (lui)
24 localparam OP_JAL     = 7'b1101111; // Jタイプ (jal)
25
26 // ALU操作コード (ALU.vと同期)
27 localparam ALU_ADD = 3'b000;
28 localparam ALU_SUB = 3'b001;
29 localparam ALU_AND = 3'b010;
30 localparam ALU_OR  = 3'b011;
31 localparam ALU_XOR = 3'b100;

```



```

32  localparam ALU_SLL = 3'b101;
33  localparam ALU_SRL = 3'b110;
34  localparam ALU_SRA = 3'b111;
35
36  // mem_to_reg の定義
37  localparam MEM2REG_ALU_RESULT = 2'b00;
38  localparam MEM2REG_MEM_DATA = 2'b01;
39
40
41  always @(*) begin
42      // デフォルト値 (安全のため)
43      reg_write_en = 1'b0;
44      alu_op = ALU_ADD; // デフォルトは加算としておく
45      alu_src_b = 1'b0; // デフォルトはレジスタ2
46      mem_read_en = 1'b0;
47      mem_write_en = 1'b0;
48      mem_to_reg = MEM2REG_ALU_RESULT;
49
50      case (opcode)
51          OP_R_TYPE: begin // Rタイプ命令 (add, sub, and, or, xorなど)
52              reg_write_en = 1'b1;
53              alu_src_b = 1'b0; // ALUのB入力 はレジスタ2
54              mem_read_en = 1'b0;
55              mem_write_en = 1'b0;
56              mem_to_reg = MEM2REG_ALU_RESULT;
57              case ({funct7, funct3}) // funct7 と funct3 でALU操作を決定
58                  // add (funct7=0x00, funct3=0x0)
59                  {7'h00, 3'b000}: alu_op = ALU_ADD;
60                  // sub (funct7=0x20, funct3=0x0)
61                  {7'h20, 3'b000}: alu_op = ALU_SUB;
62                  // and (funct7=0x00, funct3=0x7)
63                  {7'h00, 3'b111}: alu_op = ALU_AND;
64                  // or (funct7=0x00, funct3=0x6)
65                  {7'h00, 3'b110}: alu_op = ALU_OR;
66                  // xor (funct7=0x00, funct3=0x4)
67                  {7'h00, 3'b100}: alu_op = ALU_XOR;
68                  // sll (funct7=0x00, funct3=0x1)
69                  {7'h00, 3'b001}: alu_op = ALU_SLL;
70                  // srl (funct7=0x00, funct3=0x5)
71                  {7'h00, 3'b101}: alu_op = ALU_SRL;
72                  // sra (funct7=0x20, funct3=0x5)
73                  {7'h20, 3'b101}: alu_op = ALU_SRA;
74                  default: begin
75                      // 未サポートのRタイプ命令
76                      alu_op = ALU_ADD; // デフォルト
77                  end
78              endcase
79          end
80          OP_IMM: begin // addi, andi など
81              reg_write_en = 1'b1;
82              alu_src_b = 1'b1; // ALUのB入力 は即値
83              mem_read_en = 1'b0;
84              mem_write_en = 1'b0;
85              mem_to_reg = MEM2REG_ALU_RESULT;
86              case (funct3)
87                  3'b000: alu_op = ALU_ADD; // addi
88                  3'b111: alu_op = ALU_AND; // andi
89                  3'b001: alu_op = ALU_SLL; // slli
90                  3'b101: begin // srli or srai
91                      // Check funct7[5] (which corresponds to instruction bit 30)
92                      if (funct7[5] == 1'b1) begin
93                          alu_op = ALU_SRA; // srai
94                      end else begin

```

```

95         alu_op = ALU_SRL; // srl
96     end
97 end// 他のIタイプ命令
98     default: alu_op = ALU_ADD;
99 endcase
100 end
101 OP_LOAD: begin // lw など
102     reg_write_en = 1'b1;
103     alu_src_b     = 1'b1; // アドレス計算に即値を使用
104     mem_read_en   = 1'b1;
105     mem_write_en  = 1'b0;
106     mem_to_reg    = MEM2REG_MEM_DATA;
107     alu_op        = ALU_ADD; // ベースアドレス + オフセット
108 end
109 OP_STORE: begin // sw など
110     reg_write_en = 1'b0;
111     alu_src_b     = 1'b1; // アドレス計算に即値を使用
112     mem_read_en   = 1'b0;
113     mem_write_en  = 1'b1;
114     mem_to_reg    = MEM2REG_ALU_RESULT; // 関係ないがデフォルト
115     alu_op        = ALU_ADD; // ベースアドレス + オフセット
116 end
117 OP_BRANCH: begin
118     reg_write_en = 1'b0;
119     alu_src_b     = 1'b0; // 比較は2つのレジスタ間
120     mem_read_en   = 1'b0;
121     mem_write_en  = 1'b0;
122     mem_to_reg    = MEM2REG_ALU_RESULT;
123     case (funct3)
124         3'b000: alu_op = ALU_SUB; // beq:  $x == y \rightarrow x - y == 0$ 
125         3'b001: alu_op = ALU_SUB; // bne:  $x != y \rightarrow x - y != 0$ 
126         default: alu_op = ALU_SUB; // 他の比較命令（未実装なら一旦SUB）
127     endcase
128     // 分岐制御信号（別途 branch_taken などの生成が必要）
129 end
130 OP_LUI: begin
131     reg_write_en = 1'b1;
132     alu_src_b     = 1'b1; // 無視されるが一応設定
133     mem_read_en   = 1'b0;
134     mem_write_en  = 1'b0;
135     mem_to_reg    = MEM2REG_ALU_RESULT;
136     alu_op        = ALU_ADD; // ALUは不要だが、0 + immの形にしてもよい
137 end
138 OP_JAL: begin
139     reg_write_en = 1'b1;
140     alu_src_b     = 1'b0; // PC + 4 → rd
141     mem_read_en   = 1'b0;
142     mem_write_en  = 1'b0;
143     mem_to_reg    = MEM2REG_ALU_RESULT;
144     alu_op        = ALU_ADD; // PC + 4（ただし別ロジックで）
145 end
146 default: begin
147     // 未サポートの命令
148 end
149 endcase
150 end
151
152 endmodule

```

Listing 6: 制御ユニット

1.1.7 sign_extender.v

I/B/U/J 型命令の即値を符号拡張して 32 ビットに変換するユニット。

```
1 // sign_extender.v
2 // 即値生成・符号拡張ユニット
3 // 将来的には他の命令フォーマットにも対応させる
4
5 module sign_extender (
6     input wire [31:0] instruction,
7     output wire [31:0] extended_immediate,
8     output wire [31:0] imm_branch,
9     output wire [31:0] imm_u_type,
10    output wire [31:0] imm_j_type
11 );
12    // B-type
13    assign imm_branch = {{19{instruction[31]}}, instruction[31], instruction[7],
14                        instruction[30:25], instruction[11:8], 1'b0};
15
16    // Iタイプ命令の即値 (instruction[31:20]) を符号拡張
17    // RISC-Vは下位12ビットの即値の場合、上位20ビットを最上位ビットで埋める
18    assign extended_immediate = {{20{instruction[31]}}, instruction[31:20]};
19
20    // U type
21    assign imm_u_type = {instruction[31:12], 12'b0};
22
23    // J type
24    assign imm_j_type = {{12{instruction[31]}}, instruction[19:12], instruction[20],
25                        instruction[30:21], 1'b0};
26
27 endmodule
```

Listing 7: 即値生成ユニット

1.1.8 data_memory.v

ロード(lw)・ストア(sw)命令に使用されるデータメモリ。読み書きはmem_read_enとmem_write_enにより制御される。

```
1 // data_memory.v
2 module data_memory (
3     input wire clk,
4     input wire [31:0] addr,
5     input wire [31:0] write_data,
6     input wire mem_read_en,
7     input wire mem_write_en,
8     output reg [31:0] read_data_out
9 );
10
11    reg [31:0] mem [0:1023];
12    integer i;
13
14    initial begin
15        for (i = 0; i < 1024; i = i + 1)
16            mem[i] = 32'hDEAD_BEEF;
17        mem[2] = 32'hDEAD_BEEF;
18    end
19
20    // 読み出し
21    always @(*) begin
22        if (mem_read_en) begin
23            read_data_out = mem[addr[31:2]];
24        `ifdef DEBUG_MEM
```

```

25         $display("[%0t] DMEM_READ: addr=%h idx=%0d data=%h",
26                 $time, addr, addr[31:2], read_data_out);
27     `endif
28     end else begin
29         read_data_out = 32'h0000_0000;
30     end
31 end
32
33 // 書き込み（同期）
34 always @(posedge clk) begin
35     if (mem_write_en) begin
36         mem[addr[31:2]] <= write_data;
37     `ifdef DEBUG_MEM
38         $display("[%0t] DMEM_WRITE: addr=%h idx=%0d data=%h",
39                 $time, addr, addr[31:2], write_data);
40     `endif
41     end
42 end
43
44 endmodule

```

Listing 8: データメモリ

1.2 動作波形と確認

以下に、ALU 演算やジャンプ命令の動作確認を行ったときの波形を示す。

1.2.1 add, sub, xor 命令

```

1 // I-type
2 mem[0] = 32'b0000000000001_00000_000_00001_0010011; // addi x1, x0, 1    → x1
   = 1
3 mem[1] = 32'b0000000000010_00000_000_00010_0010011; // addi x2, x0, 2    → x2
   = 2
4
5 // R-type
6 mem[2] = 32'b00000000_00010_00001_000_00011_0110011; // add x3, x1, x2    → x3
   = x1 + x2 = 3
7 mem[3] = 32'b0100000_00001_00011_000_00100_0110011; // sub x4, x3, x1    → x4
   = x3 - x1 = 2

```



Figure 1: I, R-type 命令の動作波形

1.2.2 or, and, sll, srl 命令

```

1 // I-type
2 mem[0] = 32'b0000000000001_00000_000_00001_0010011; // addi x1, x0, 1    → x1
   = 1
3 mem[1] = 32'b0000000000010_00000_000_00010_0010011; // addi x2, x0, 2    → x2
   = 2
4
5 // R-type
6 mem[2] = 32'b00000000_00010_00001_000_00011_0110011; // add x3, x1, x2    → x3
   = x1 + x2 = 3

```

```

7  mem[3] = 32'b0100000_00001_00011_000_00100_0110011; // sub x4, x3, x1 → x4
   = x3 - x1 = 2
8  mem[4] = 32'b0000000_00010_00011_100_00101_0110011; // xor x5, x3, x2 → x5
   = x3 ^ x2 = 1
9  mem[5] = 32'b0000000_00010_00011_110_00110_0110011; // or x6, x3, x2 → x6
   = x3 | x2 = 3
10 mem[6] = 32'b0000000_00010_00011_111_00111_0110011; // and x7, x3, x2 → x7
   = x3 & x2 = 2
11 mem[7] = 32'b0000000_00010_00011_001_01000_0110011; // sll x8, x3, x2 → x8
   = x3 << x2 = 12
12 mem[8] = 32'b0000000_00010_00011_101_01001_0110011; // srl x9, x3, x2 → x9
   = x3 >> x2 = 0

```

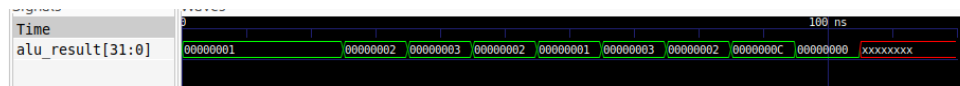


Figure 2: or, and, sll, srl 命令の動作波形

1.2.3 lw, sw 命令

```

1  // I-type
2  mem[0] = 32'b0000000000001_00000_000_00001_0010011; // addi x1, x0, 1 → x1
   = 1
3  mem[1] = 32'b0000000000010_00000_000_00010_0010011; // addi x2, x0, 2 → x2
   = 2
4
5  // R-type
6  mem[2] = 32'b0000000_00010_00001_000_00011_0110011; // add x3, x1, x2 → x3
   = x1 + x2 = 3
7  mem[3] = 32'b0100000_00001_00011_000_00100_0110011; // sub x4, x3, x1 → x4
   = x3 - x1 = 2
8  mem[4] = 32'b0000000_00010_00011_100_00101_0110011; // xor x5, x3, x2 → x5
   = x3 ^ x2 = 1
9  mem[5] = 32'b0000000_00010_00011_110_00110_0110011; // or x6, x3, x2 → x6
   = x3 | x2 = 3
10 mem[6] = 32'b0000000_00010_00011_111_00111_0110011; // and x7, x3, x2 → x7
   = x3 & x2 = 2
11 mem[7] = 32'b0000000_00010_00011_001_01000_0110011; // sll x8, x3, x2 → x8
   = x3 << x2 = 12
12 mem[8] = 32'b0000000000000_00010_111_00101_0000011; // lw x5, 0(x8) →
   x5 = MEM[x8+0]

```

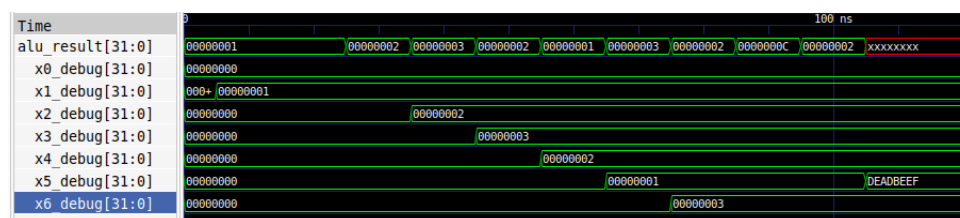


Figure 3: or, and, sll, srl 命令の動作波形

1.2.4 beq, bne 命令

```

1  // I-type
2  mem[0] = 32'b0000000000001_00000_000_00001_0010011; // addi x1, x0, 1 → x1
   = 1

```

```

3  mem[1] = 32'b000000000010_00000_000_00010_0010011; // addi x2, x0, 2    → x2
   = 2
4
5  // R-type
6  mem[2] = 32'b00000000_00010_00001_000_00011_0110011; // add x3, x1, x2    → x3
   = x1 + x2 = 3
7  mem[3] = 32'b0100000_00001_00011_000_00100_0110011; // sub x4, x3, x1    → x4
   = x3 - x1 = 2
8  mem[4] = 32'b00000000_00010_00011_100_00101_0110011; // xor x5, x3, x2    → x5
   = x3 ^ x2 = 1
9
10 // B-type
11 mem[5] = 32'b00000000_00011_00010_000_01000_1100011; // beq x2, x3, +8    →
   branch not taken
12 mem[6] = 32'b00000000_00011_00010_001_01000_1100011; // bne x2, x3, +8    →
   branch taken
13
14 // U-type
15 mem[7] = 32'b0000000000000000000000001_00101_0110111; // lui x5, 0x0    → x5 =
   0x00000000
16 mem[8] = 32'b00010010001101000101_00001_0110111; // lui x1, 0x12345    → x13 =
   0x12345000

```

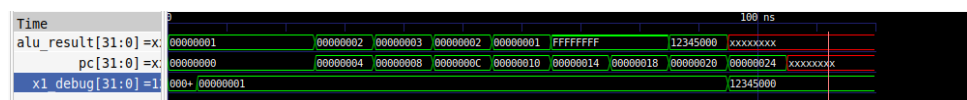


Figure 4: B-type, U-type 命令の動作波形

1.2.5 jal 命令

```

1  // I-type
2  mem[0] = 32'b0000000000001_00000_000_00001_0010011; // addi x1, x0, 1    → x1
   = 1
3  mem[1] = 32'b000000000010_00000_000_00010_0010011; // addi x2, x0, 2    → x2
   = 2
4
5  // R-type
6  mem[2] = 32'b00000000_00010_00001_000_00011_0110011; // add x3, x1, x2    → x3
   = x1 + x2 = 3
7  mem[3] = 32'b0100000_00001_00011_000_00100_0110011; // sub x4, x3, x1    → x4
   = x3 - x1 = 2
8  mem[4] = 32'b00000000_00010_00011_100_00101_0110011; // xor x5, x3, x2    → x5
   = x3 ^ x2 = 1
9
10 mem[5] = 32'h002000ef; // jal x1, 4

```

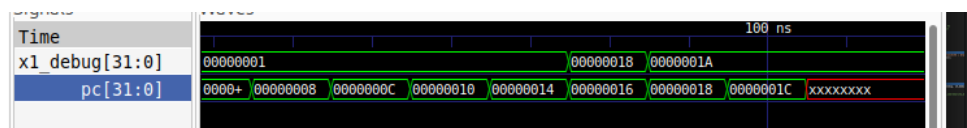


Figure 5: jal 命令の動作波形 (x1 に PC+4 が保存される)

2 課題 2

2023 年以降に主要なコンピュータアーキテクチャ会議で発表された論文の中から、「FlashLLM: A Chiplet-Based In-Flash Computing Architecture to Enable On-Device Inference of 70B LLM」(MICRO 2024) [1] を選定し、その概要と将来のアーキテクチャ課題への貢献について詳述する。この論文は、大規模言語モデル (LLM) をスマートフォンや車両などのエッジデバイスに展開する際の、メモリ帯域幅の制約と電力効率の課題に正面から取り組んでいる。FlashLLM は、チップレット技術を介して NPU (ニューラルプロセッシングユニット) に直接接続された専用のフラッシュチップを特徴とする、革新的なハイブリッドアーキテクチャを提案している。この設計は、モデルの重みを行動中にフラッシュメモリ内で処理する「インフラッシュコンピューティング」の概念を導入することで、データ転送量を劇的に削減し、結果として推論速度とエネルギー効率を大幅に向上させている。

2.1 モチベーション: エッジデバイスにおける LLM 展開の課題

大規模言語モデル (LLM) は、そのテキスト生成能力により、現代の自然言語処理の中核を担っている。ChatGPT の急速な普及が示すように、LLM はすでにテクノロジー業界の製品に深く組み込まれている。しかし、LLM は数十億から数兆に及ぶ膨大な数のパラメータを必要とし、スマートフォンや車両などのリソースが限られたエッジデバイスへの展開において、重大な課題を提起している。既存のフラッシュオフロード技術では、モデルの重みをフラッシュベースのストレージにオフロードする研究が盛んに行われているが、フラッシュメモリの限られた帯域幅が推論速度を著しく阻害するという問題がある。特に、エッジ推論シナリオではバッチサイズが 1 であり、演算強度が最小限であるため、この帯域幅の制約がボトルネックとなる。これは、プロセッサとメモリ間のデータ転送速度の不均衡を示す「メモリウォール」問題が、LLM の文脈でさらに悪化していることを意味する。

2.2 アイデア: FlashLLM のチップレットベースハイブリッドアーキテクチャ

課題に対処するため、FlashLLM は 70B LLM のオンデバイス推論のために特別に設計された、チップレットベースのハイブリッドアーキテクチャを導入している。このアーキテクチャは、以下の主要コンポーネントで構成される:

- 専用フラッシュチップ: チップレット技術を介してニューラルプロセッシングユニット (NPU) に直接接続されている。主な目的は、モデルの重み行列を保存することと、オンダイ処理能力を活用してデータ転送を削減することである
- ニューラルプロセッシングユニット (NPU): フラッシュチップと連携して行列演算を実行する。さらに、NPU は DRAM 内のキーバリュー (KV) キャッシュを管理し、フラッシュのオンダイ処理能力を超える特殊な機能計算を担当する。

2.3 効果

Table 1: LLM モデルにおける推論速度と高速化率 [1]

LLM モデルサイズ	推論速度 (トークン/秒)	既存のフラッシュオフロード技術に対する速度向上
70B	3.44	22 倍以上
7B	36.34	45 倍以上

Table 1 が示すように、FlashLLM は、70B LLM で 3.44 トークン/秒、7B LLM で 36.34 トークン/秒の推論速度を達成し、既存のフラッシュオフロード技術と比較して 22 倍から 45 倍の速度向上を実現している。この性能向上は、メモリ帯域幅、容量、電力という複数のボトルネックに同時に取り組む、多層的な解決策の有効性を示している。

2.4 将来のアーキテクチャ課題への貢献

FlashLLM が示す方向性は、将来的な「オンデバイス LLM」における以下の課題に答えている：

- 帯域制約と電力消費の克服：データを「動かす」より「現地で計算する」アプローチ
- チップレット技術の実用化：SoC 外部の非 DRAM 記憶装置との協調演算という新しいデザイン空間
- 誤り訂正との統合アーキテクチャ：誤りに脆弱なフラッシュでも AI 推論が実行可能になる設計の第一歩

References

- [1] Zhongkai Yu, Shengwen Liang, Tianyun Ma, Yunke Cai, Ziyuan Nan, Di Huang, Xinkai Song, Yifan Hao, Jie Zhang, Tian Zhi, Yongwei Zhao, Zidong Du, Xing Hu, Qi Guo, and Tianshi Chen. Cambricon-llm: A chiplet-based hybrid architecture for on-device inference of 70b llm, 2024.
- [2] 修一 坂井. コンピュータアーキテクチャ. 電子情報通信レクチャーシリーズ C-9. コロナ社, 2004.