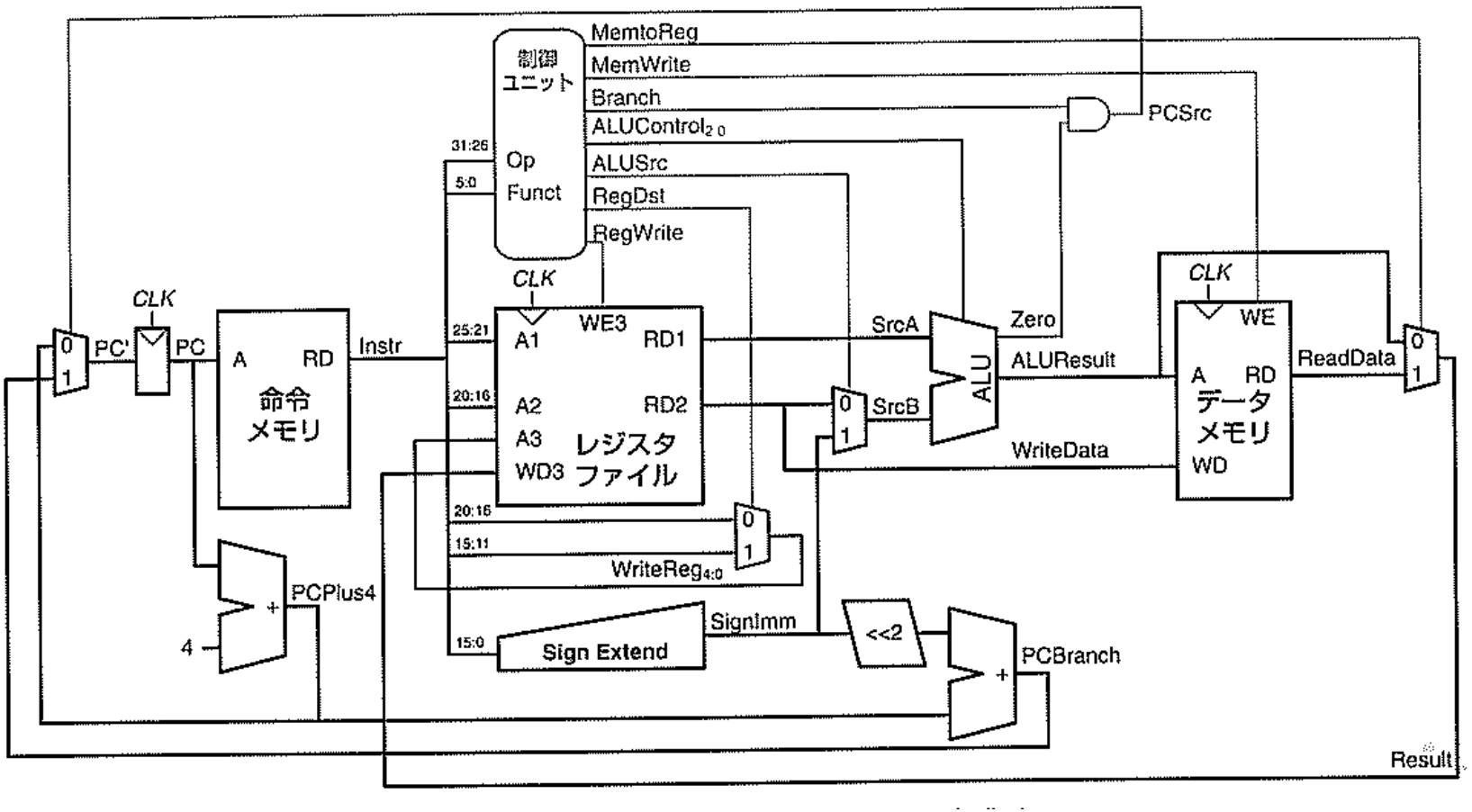
MIPS R3000サブセットの1サイクルマイクロアーキテクチャ

- 命令実装は部分的
- メモリアクセスはLW,SWだけで、メモリはワード単位でしかアクセスができない。（しかしバイトアドレッシングにはなっている）
- 機械語はMIPS R3000準拠
- 信号名はHarris&Harrisのテキストに準拠、ただしVerilog記述上は全て小文字
- Verilog記述は入門スタイルに準拠。Harris&Harrisのとは全然違っている

1サイクルMIPSの構成



Verilogの信号名は全て小文字になっている

define文（基本的なデータ長、ALUの機能）

```
`define DATA_W 32
`define SEL_W 6
`define REG 32
`define REG_W 5
`define OPCODE_W 6
`define SHAMT_W 5
`define LANE_W 4
`define IMM_W 16
`define JIMM_W 26
`define DEPTH 65536

`define ALU_THB `SEL_W'b000001
`define ALU_ADD `SEL_W'b100000
`define ALU_SUB `SEL_W'b100010
`define ALU_AND `SEL_W'b100100
`define ALU_OR `SEL_W'b100101
`define ALU_XOR `SEL_W'b100110
`define ALU_NOR `SEL_W'b100111
```

ALUのコードはMIPSの命令コードに合うように決めてある

```
`define ENABLE 1'b1
`define DISABLE 1'b0
`define ENABLE_N 1'b0
`define DISABLE_N 1'b1

`define OP_REG `OPCODE_W'b000000
`define OP_BEQ `OPCODE_W'b000100
`define OP_BNE `OPCODE_W'b000101
`define OP_LB `OPCODE_W'b100000
`define OP_LW `OPCODE_W'b100011
`define OP_SB `OPCODE_W'b101000
`define OP_SW `OPCODE_W'b101011
`define OP_LUI `OPCODE_W'b001111
`define OP_ADDI `OPCODE_W'b001000
`define OP_ORI `OPCODE_W'b001101
`define OP_J `OPCODE_W'b000010
`define OP_JAL `OPCODE_W'b000011
`define OP_SLTI `OPCODE_W'b001010

`define FUNC_JR `OPCODE_W'b001000
`define FUNC_JALR `OPCODE_W'b001001
`define FUNC_SLT `OPCODE_W'b101010
```

define文つづき　命令コード
これもMIPSのコードに準拠して決めている

ALUの記述（機能は一部のみ）

```
`include "def.h"
module alu (
  input [`DATA_W-1:0] a, b,
  input [`SEL_W-1:0] s,
  output [`DATA_W-1:0] y,
  output zero );
  assign y = s==`ALU_ADD ? a+b:
             s==`ALU_SUB ? a-b:
             s==`ALU_AND ? a & b:
             s==`ALU_OR ? a | b:
             s==`ALU_XOR ? a ^ b:
             s==`ALU_NOR ? ~ (a | b): b;
  assign zero = (y == 32'b0);
endmodule
```

条件選択文を直接使っている
zeroは結果がゼロになったときに1となる

レジスタファイル

```
`include "def.h"
module rfile (
 input clk,
 input [`REG_W-1:0] a1, a2, a3,
 output [`DATA_W-1:0] rd1, rd2,
 input [`DATA_W-1:0] wd3,
 input we3);
  reg [`DATA_W-1:0] rf[0:`REG-1];
  assign rd1 = |a1 == 0 ? 0: rf[a1];
  assign rd2 = |a2 == 0 ? 0: rf[a2];
  always @(posedge clk)
            if(we3) rf[a3] <= wd3;
endmodule
```

レジスタファイルrfはメモリの形で宣言

r0は常に0を出力する

初期化は行わない（レジスタファイルをIPで作る場合、通常初期化はできないため）

```
`include "def.h"
module mipse(
input clk, rst_n,
input [`DATA_W-1:0] instr,
input [`DATA_W-1:0] readdata,
output reg [`DATA_W-1:0] pc,
output [`DATA_W-1:0] aluresult,
output [`DATA_W-1:0] writedata,
output memwrite);
```

入出力の定義、信号名は最初の図を参照のこと（Harris&Harrisのテキスト準拠）

```
wire [`DATA_W-1:0] srca, srcb, result;
wire [`OPCODE_W-1:0] opcode;
wire [`SHAMT_W-1:0] shamt;
wire [`OPCODE_W-1:0] func;
wire [`REG_W-1:0] rs, rd, rt, writereg;
wire [`SEL_W-1:0] com;
wire [`DATA_W-1:0] signimm;
wire [`DATA_W-1:0] pcplus4;
wire regwrite;
wire sw_op, beq_op, bne_op, addi_op, lw_op, j_op, jal_op, jr_op, alu_op,slt_op;
wire zero;
```

信号名の定義。最初の図を参照のこと（Harris&Harrisのテキスト準拠）

デコード信号：各命令に対応

```
assign {opcode, rs, rt, rd, shamt, func} = instr;     読み出した命令を分解
assign signimm = {{16{instr[15]}},instr[15:0]};       イミーディエイトを符号拡張

// Decorder
assign sw_op = (opcode == `OP_SW);                    以下はI型命令
assign lw_op = (opcode == `OP_LW);
assign addi_op = (opcode == `OP_ADDI);
assign beq_op = (opcode == `OP_BEQ);
assign bne_op = (opcode == `OP_BNE);
assign j_op = (opcode == `OP_J);
assign jal_op = (opcode == `OP_JAL);

以下はR型命令
assign alu_op = (opcode == `OP_REG) & (func[5:3] == 3‘b100);     ALU演算命令
assign jr_op = (opcode == `OP_REG) & (func == `FUNC_JR);
assign slt_op = (opcode == `OP_REG) & (func == `FUNC_SLT);
```

```
assign memwrite = sw_op;     SW命令ならばメモリ書きこみ
assign srcb = (addi_op | lw_op | sw_op  )？  signimm : writedata;     ALUのB入力
I型ならば符号拡張されたイミーディエイト、そうでなければレジスタファイルからの値
assign com = (addi_op|lw_op|sw_op) ？`ALU_ADD:
                                        (beq_op | bne_op | slt_op  ) ？`ALU_SUB: func;
  ALUのコマンド：命令毎に決める
assign result = slt_op  ？{31'b0,aluresult[31]} :
                              jal_op ？pcplus4: lw_op  ？readdata : aluresult;
  レジスタファイルに何を書き込むか？
assign regwrite = lw_op | alu_op | addi_op | jal_op | slt_op  ;
  レジスタファイルに書き込む信号
assign writereg = jal_op ？5'b11111: alu_op | slt_op ？rd : rt;
  書き込むレジスタ番号、これも命令毎に切り替え
alu alu_1(.a(srca), .b(srcb), .s(com), .y(aluresult), .zero(zero));
  ALUの入出力を接続
rfile rfile_1(.clk(clk), .rd1(srca), .a1(rs), .rd2(writedata), .a2(rt),
              .wd3(result), .a3(writereg), .we3(regwrite));
  レジスタファイルの入出力を接続
```

```
assign pcplus4 = pc+4;                          PC+4を定義しておく
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if (j_op | jal_op)
              pc        <= {pc[31:28],instr[25:0],2‘b0};     J型命令
   else if (jr_op)
              pc        <= srca;                              JRはレジスタの値をセット
   else if ((beq_op & zero) | (bne_op & !zero))
     pc <= pcplus4 +{signimm[29:0],2‘b0} ;       Branch命令
   else
     pc <= pcplus4;                                  それ以外はPC+4
end

endmodule
```