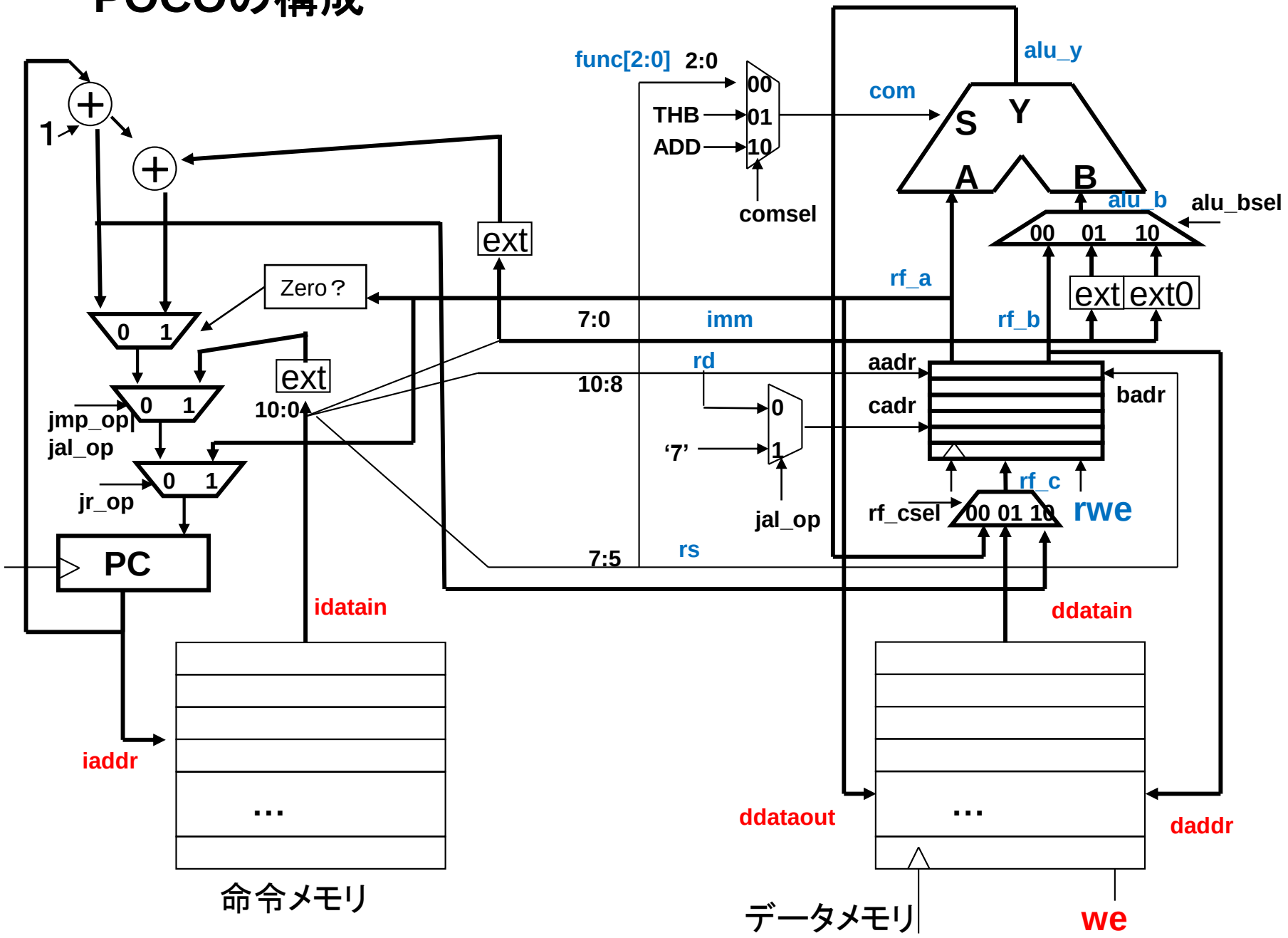
POCOの1サイクルマイクロアーキテクチャ

- POCOは「作りながら学ぶコンピュータアーキテクチャ」（倍風館）で使っている教育用の16ビットRISCである。
- www.am.ics.keio.ac.jp/parthenon/pocobook/も参照のこと

POCOの構成


1
ext
Zero？
0 1
jmp_op
jal_op
jr_op
PC
iaddr
idatain
命令メモリ
func[2:0] 2:0
THB
ADD
00
01
10
comsel
com
alu_y
S
Y
A
B
alu_b
alu_bsel
00 01 10
ext
ext0
rf_a
rf_b
7:0
imm
rd
10:8
10:0
‘7’
jal_op
aadr
cadr
badr
rf_c
rf_csel
00 01 10
rwe
rs
7:5
ddatain
ddataout
daddr
we
データメモリ

R型命令一覧

NOP		**00000------00000**
MV rd,rs	rd← rs	**00000dddsss00001**
AND rd,rs	rd← rd AND rs	**00000dddsss00010**
OR rd,rs	rd← rd OR rs	**00000dddsss00011**
SL rd	rd← rd<<1	**00000ddd---00100**
SR rd	rd← rd>>1	**00000ddd---00101**
ADD rd,rs	rd← rd + rs	**00000dddsss00110**
SUB rd,rs	rd← rd - rs	**00000dddsss00111**
ST rd,(ra)	(ra)← rd	**00000dddaaa01000**
LD rd,(ra)	rd← (ra)	**00000dddaaa01001**
JR rd	pc ← rd	**00000ddd---01010**

I型命令一覧

LDI rd,#X	rd← X(符号拡張)	**01000dddXXXXXXXX**
LDIU rd,rs	rd← X(ゼロ拡張)	**01001dddXXXXXXXX**
ADDI rd,#X	rd←rd+X(符号拡張)	**01100dddXXXXXXXX**
ADDIU rd,#X	rd←rd+X(ゼロ拡張)	**01101dddXXXXXXXX**
LDHI rd,#X	rd←{X,0}	**01010dddXXXXXXXX**
BEZ rd,X	if(rd=0) pc←pc+X+1	**10000dddXXXXXXXX**
BNZ rd,X	if(rd≠0) pc←pc+X+1	**10001dddXXXXXXXX**
BPL rd,X	if(rd>=0) pc←pc+X+1	**10010dddXXXXXXXX**
BMI rd,X	if(rd<0) pc←pc+X+1	**10011dddXXXXXXXX**

J型命令一覧

JMP　#X	pc←pc+X+1	**10100XXXXXXXXXX**
JAL　#X	pc←pc+X+1, r7←pc+1	**10101XXXXXXXXXX**

define文（基本サイズとALUのコマンド）

```
`define DATA_W 16
`define SEL_W 3
`define REG 8
`define REG_W 3
`define OPCODE_W 5
`define IMM_W 8
`define DEPTH 65536

`define ALU_THA `SEL_W'b000
`define ALU_THB `SEL_W'b001
`define ALU_AND `SEL_W'b010
`define ALU_OR `SEL_W'b011
`define ALU_SL `SEL_W'b100
`define ALU_SR `SEL_W'b101
`define ALU_ADD `SEL_W'b110
`define ALU_SUB `SEL_W'b111
```

ALUのコマンドの割り当ては非常に
いい加減に決めてあり、これがALU命令の
機械語のコードに対応している。

define文の続き：命令コードとファンクションコードの定義

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

`define OP_BEZ `OPCODE_W‘b10000        以下はI型命令の定義
`define OP_BNZ `OPCODE_W'b10001
`define OP_BPL `OPCODE_W'b10010
`define OP_BMI `OPCODE_W'b10011
`define OP_JMP `OPCODE_W'b10100
`define OP_JAL `OPCODE_W'b10101
`define OP_LDI `OPCODE_W'b01000
`define OP_LDIU `OPCODE_W'b01001
`define OP_LDHI `OPCODE_W'b01010
`define OP_ADDI `OPCODE_W'b01100
`define OP_ADDIU `OPCODE_W'b01101
`define OP_REG `OPCODE_W'b00000
`define F_ST `OPCODE_W‘b01000          以下はR型命令のファンクションの定義
`define F_LD `OPCODE_W'b01001
`define F_JR `OPCODE_W'b01010
`define F_JALR `OPCODE_W'b11000
```

ALUの記述：これは典型的な条件付選択構文で例題にも使っている。

```
`include "def.h"

module alu (
  input [`DATA_W-1:0] a, b,
  input [`SEL_W-1:0] s,
  output [`DATA_W-1:0] y );
  assign y = s==`ALU_THA ? a:
             s==`ALU_THB ? b:
             s==`ALU_AND ? a & b:
             s==`ALU_OR ? a | b:
             s==`ALU_SL ? a << 1:
             s==`ALU_SR ? a >> 1:
             s==`ALU_ADD ? a + b: a - b ;
endmodule
```

```
`include "def.h"
module rfile (
 input clk,
 input [`REG_W-1:0] aadr, badr, cadr,
 output [`DATA_W-1:0] a, b,
 input [`DATA_W-1:0] c,  input we);
    reg [`DATA_W-1:0] r0, r1, r2, r3, r4, r5, r6, r7;
    assign a = aadr == 0 ? r0:     aadr == 1 ? r1:   aadr == 2 ? r2:
                        aadr == 3 ? r3:     aadr == 4 ? r4:     aadr == 5 ? r5:
                        aadr == 6 ? r6: r7;
    assign b = badr == 0 ? r0:     badr == 1 ? r1:     badr == 2 ? r2:
                        badr == 3 ? r3:     badr == 4 ? r4:     badr == 5 ? r5:
                        badr == 6 ? r6: r7;
    always @(posedge clk) begin
                        if(we)
                        case(cadr)
                                    0: r0 <= c; 1: r1 <= c;  2: r2 <= c;
                                    3: r3 <= c;  4: r4 <= c;  5: r5 <= c;
                                    6: r6 <= c;  default: r7 <= c;
                        endcase
            end
endmodule
```

rfileの記述：レジスタは個別に定義しているが、
これはgtkwaveでデバッグしやすく
するためである

レジスタの読み出しは選択構文で書く

書きこみは
always文中なのでcase文が使える

POCO本体の記述：入出力と信号名定義

```
`include "def.h"
module poco(
input clk, rst_n,
input [`DATA_W-1:0] idatain,
input [`DATA_W-1:0] ddatain,
output [`DATA_W-1:0] iaddr, daddr,
output [`DATA_W-1:0] ddataout,
output we);

reg [`DATA_W-1:0] pc;
wire [`DATA_W-1:0] rf_a, rf_b, rf_c;
wire [`DATA_W-1:0] alu_b, alu_y;
wire [`OPCODE_W-1:0] opcode;
wire [`OPCODE_W-1:0] func;
wire [`REG_W-1:0] rs, rd, cadr;
wire [`SEL_W-1:0] com;
wire [`IMM_W-1:0] imm;
wire rwe;
wire st_op, bez_op, bnz_op, bmi_op, bpl_op, addi_op,  alu_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op, jmp_op, jal_op, jr_op, jalr_op;
```

信号名称は2ページの図と一致しているので参照のこと

以下はデコード信号

命令のデコード

```
// Decorder
assign st_op = (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = (opcode == `OP_REG) & (func == `F_LD);
assign jr_op = (opcode == `OP_REG) & (func == `F_JR);
assign jalr_op = (opcode == `OP_REG) & (func == `F_JALR);
assign alu_op = (opcode == `OP_REG) & (func[4:3] == 2'b00);

assign ldi_op = (opcode == `OP_LDI);
assign ldiu_op = (opcode == `OP_LDIU);
assign addi_op = (opcode == `OP_ADDI);
assign addiu_op = (opcode == `OP_ADDIU);
assign ldhi_op = (opcode == `OP_LDHI);
assign bez_op = (opcode == `OP_BEZ);
assign bnz_op = (opcode == `OP_BNZ);
assign bpl_op = (opcode == `OP_BPL);
assign bmi_op = (opcode == `OP_BMI);
assign jmp_op = (opcode == `OP_JMP);
assign jal_op = (opcode == `OP_JAL);
```

R型命令

I型命令

出力信号の接続と、ALU周辺

出力信号の接続

```
assign iaddr = pc;
assign daddr = rf_b;
assign we = st_op
```

読んできた命令の分解

```
assign {opcode, rd, rs, func} = idatain;
assign imm = idatain[`IMM_W-1:0];
```

ALUのB入力の選択

```
assign alu_b = (addi_op | ldi_op) ? {{8{imm[7]}},imm} :
               (addiu_op | ldiu_op) ? {8'b0,imm} :
               (ldhi_op) ? {imm, 8'b0} : rf_b;
```

ALUのcomの選択

```
assign com = (addi_op | addiu_op ) ? `ALU_ADD:
             (ldi_op | ldiu_op | ldhi_op) ? `ALU_THB: func[`SEL_W-1];
```

レジスタファイル周辺、ALUとレジスタファイルの接続

```
assign rf_c = ld_op  ? ddatain : jal_op ? pc+1 : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op
                    | jal_op ;

assign cadr = jal_op ? 3'b111 : rd;

alu alu_1(.a(rf_a), .b(alu_b), .s(com), .y(alu_y));

rfile rfile_1(.clk(clk), .a(rf_a), .aadr(rd), .b(rf_b), .badr(rs),
              .c(rf_c), .cadr(cadr), .we(rwe));
```

レジスタの入力選択

レジスタの書きこみ信号

レジスタCポートの選択

ALUの入出力を接続

レジスタの入出力を選択

PCの制御

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if ((bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0) |
                          (bpl_op & ~rf_a[15]) | (bmi_op & rf_a[15]))
     pc <= pc +{{8{imm[7]}},imm}+1 ;
   else if (jmp_op | jal_op)
    pc <= pc + {{5{idatain[10]}},idatain[10:0]}+1;
   else if(jr_op)
    pc <= rf_a;
   else
     pc <= pc+1;
end

endmodule
```

非同期リセット

BEZ,BNZ,BPL,BMIの条件チェック

8ビットの符号拡張で相対番地計算

11ビットの符号拡張で相対番地計算

JRはレジスタAポートの値をそのまま使う

分岐でない場合はカウントアップ

module poco終了