# POCO のマルチサイクル化

## １サイクル CPU の問題点

POCO は、周波数は 125MHz(8nsec) で動作し、コストを大きくして良ければもっと高い周波数でも動作可能であることがわかった。しかし POCO の問題点は、すべての命令を 1 サイクルで動作させることで、このため以下の問題がある。

- 資源の使い回しができない。特に命令メモリとデータメモリを別々に持たなければならない。
- クリティカルパスがもっとも長い命令に支配され、結局かなり長くなること。

### マルチサイクル化によるコストの低減

そこで、他のプロセッサ同様に、POCO をマルチサイクル化、つまり 1 命令を複数クロック掛けて実行するように改造しよう。もっとも容易なのは、以下のように分ける方法である。

- 命令をメモリから取ってくる。(Insruction Fetch:IF) 命令を取ってくることを命令フェッチと呼ぶ。これは昔からの習慣で、「命令ゲット」とは言わない。ここで、取ってきた命令をしまっておくレジスタが必要である。このレジスタを命令レジスタ (Instruction Register: ir) と呼ぶ。
- ir に格納された命令を実行 (Execution:EX) する。

このように分けると、一つのメモリに命令とデータの両方を置くことができるようになる。また、pc のカウントアップ、飛び先の計算も ALU で行うことができる。すなわち、以下のように使う。

- IF: メモリのアドレスに pc を与え、読み出してきた命令を ir に格納する。ALU では pc=pc+1 を計算する。
- EX: メモリのアドレスにはレジスタファイルの b ポートの出力を接続し、読み出してきたデータはレジスタファイルに格納して LD 命令を実行できるようにする。一方、書き込みデータにはレジスタファイルの a ポートの出力を接続して ST 命令を実行する。分岐命令、ジャンプ命令の場合は、ALU 出飛び先アドレスを計算する。

マルチサイクル化したデータパスを図 1 に示す。メモリが単一になっている点に着目されたい。このアドレス入力には pc とレジスタファイルの b 出力がマルチプレクサを介して接続される。メモリから読んできた命令を格納するために ir が設けられている。また、pc に対する演算を行うため、ALU の A 入力、B 入力共にマルチプレクサを拡張する必要がある。

図 1 は、このためのパスを付け加えたものである。PC は A 入力に入れる。B 入力では、IF 状態では、+1 するための 1、分岐命令が成立した場合は下位の 8 ビットを符号拡張し、JMP や JAL ならば 11 ビットを符号拡張して入れる。com にもそれぞれに対応した操作を与える必要がある。

さて、このデータパスを制御するためには、コントローラは今までのような組み合わせ回路ではダメで、「今、命令フェッチをしているのか実行をしているのか」を識別する順序回路が必要である。今回は、単純に IF と EX の 2 つ状態を交互に繰り返す図 2 のような非常に簡単な順序回路を用いる。これは少数のフリップフロップで簡単に実現できる。

各状態で、図中の信号線をどのように制御して良いかを表 1 に示す。単一サイクル制御の場合と違って pc にも、値をセットするかどうかを判断する pcset が必要になる。これが 1 の時だけ入力が設定される。ir も同様である。

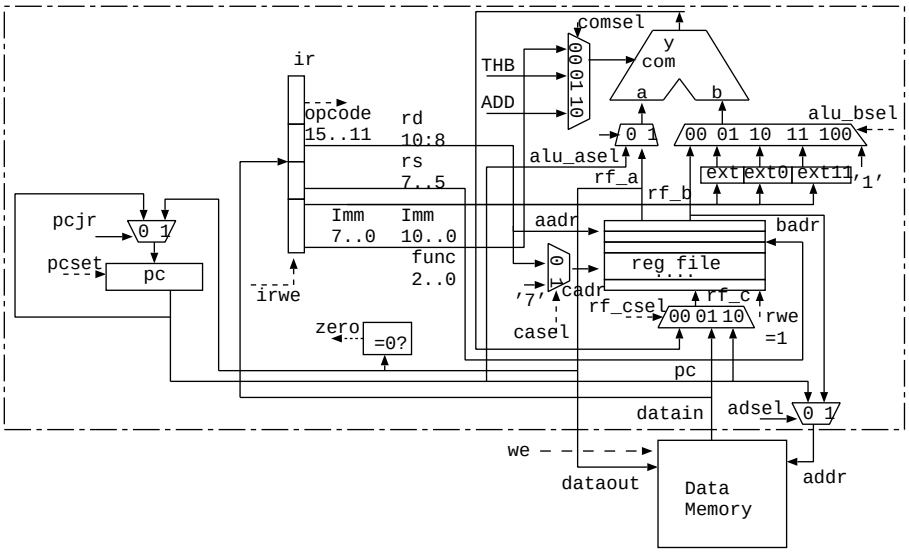


図 1: マルチサイクル化したデータパス

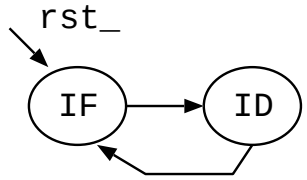


図 2: 状態遷移

表 1: 各命令の制御信号

| | pcset | pcsel | adsel | irwe | comsel | alu_asel | alu_bsel | rf_csel | rwe | we |
|---|---|---|---|---|---|---|---|---|---|---|
| IF | 1 | 0 | 0 | 1 | - | 0 | - | - | 0 | 0 |
| ADDI | 0 | - | 1 | 0 | 10 | 1 | 01 | 0 | 1 | 0 |
| SUB | 0 | - | 1 | 0 | 00 | 1 | 00 | 0 | 1 | 0 |
| LD | 0 | - | 1 | 0 | - | 1 | - | 1 | 1 | 0 |
| ST | 0 | - | 1 | 0 | - | 1 | - | - | 0 | 1 |
| BNZ | 1/0 | 1 | 1 | 0 | - | 0 | - | - | 0 | 0 |

## 2 サイクル poco の Verilog 記述

まず、全体を制御するコントローラの記述を示す。

```
reg [‘STAT_W-1:0] stat;
...
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= ‘STAT_IF;
   else
    case (stat)
      ‘STAT_IF: stat <= ‘STAT_EX;
      ‘STAT_EX: stat <= ‘STAT_IF;
    endcase
end
```

この記述に関連する定義ファイルの内容は以下の通りである。

```
‘define STAT_W 2
‘define STAT_IF ‘STAT_W’b01
‘define IF 1’b0
‘define STAT_EX ‘STAT_W’b10
‘define EX 1’b1
```

ここでは、状態は stat という 2 ビットのレジスタに格納しておく。0 ビット目が 1 ならば IF 状態、1 ビット目が 1 ならば EX 状態である。このため、Verilog 記述上は、

```
stat[‘IF]
```

が真 (1) ならば IF 状態、

```
stat[‘EX]
```

が真 (1) ならば EX 状態であることがわかる。このように状態と同じ数のビットを用いる方法をワンホットカウンタ (one-hot counter) と呼ばれる。したがってメモリのアドレスバスに、IF 状態では pc を繋ぎ、EX 状態ではレジスタファイルのポート B を接続するためには、以下のように書けば良く、簡単である。

```
assign addr = stat[‘IF] ? pc: rf_b;
```

では、最初から記述を追って行こう。

```
module poco(
input clk, rst_n,
input [‘DATA_W-1:0] datain,
output [‘DATA_W-1:0] addr,
output [‘DATA_W-1:0] dataout,
output we);
reg [‘DATA_W-1:0] pc;
reg [‘DATA_W-1:0] ir;
reg [‘STAT_W-1:0] stat;
wire [‘DATA_W-1:0] rf_a, rf_b, rf_c;
```

```
wire [‘DATA_W-1:0] alu_a, 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 [‘JIMM_W-1:0] jimm;
wire pcset, rwe;
wire st_op, bez_op, bnz_op, bmi_op, bpl_op, addi_op, ld_op, alu_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op, jmp_op, jal_op, jr_op, b_op, j_op;
assign dataout = rf_a;
assign addr = stat[‘IF] ? pc: rf_b;
assign {opcode, rd, rs, func} = ir;
assign imm = ir[‘IMM_W-1:0];
assign jimm = ir[‘JIMM_W-1:0];
```

入出力に関しては、メモリを単一にしたため、addr, datain, dataout ともに一系統で済む。後は ir を宣言した点、JMP や JAL 用に jimm を定義して見やすくした点が異なる。

```
// Decorder
assign st_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_LD);
assign jr_op = stat[‘EX] & (opcode == ‘OP_REG) & (func == ‘F_JR);
assign alu_op = stat[‘EX] & (opcode == ‘OP_REG) & (func[4:3] == 2’b00);
assign ldi_op = stat[‘EX] & (opcode == ‘OP_LDI);
assign ldiu_op = stat[‘EX] & (opcode == ‘OP_LDIU);
assign addi_op = stat[‘EX] & (opcode == ‘OP_ADDI);
assign addiu_op = stat[‘EX] & (opcode == ‘OP_ADDIU);
assign ldhi_op = stat[‘EX] & (opcode == ‘OP_LDHI);
assign bez_op = stat[‘EX] & (opcode == ‘OP_BEZ);
assign bnz_op = stat[‘EX] & (opcode == ‘OP_BNZ);
assign bpl_op = stat[‘EX] & (opcode == ‘OP_BPL);
assign bmi_op = stat[‘EX] & (opcode == ‘OP_BMI);
assign jmp_op = stat[‘EX] & (opcode == ‘OP_JMP);
assign jal_op = stat[‘EX] & (opcode == ‘OP_JAL);

assign we = st_op;
assign b_op = bez_op | bnz_op | bpl_op | bmi_op;
assign j_op = jmp_op | jal_op ;
```

デコーダは、各演算が有効になるのは EX 状態のみである（IF 状態ではその前の命令が入っている）ので、EX 状態に居るということを条件に付け加えている。それ以外は同じである。また、相対アドレスの分岐命令であることを示す b_op、相対アドレスのジャンプ命令であることを示す j_op を付け加えた。これは記述を見やすくするためである。

```
assign alu_a = (stat[‘IF] | b_op | j_op | jr_op ) ? pc : rf_a;

assign alu_b = stat[‘IF] ? 16’b1:
```

```
                (addi_op | ldi_op | b_op) ? {{8{imm[7]}},imm} :
                (addiu_op | ldiu_op ) ? {8'b0,imm} :
                (j_op) ? {{5{jimm[10]}},jimm} :
                (ldhi_op) ? {imm, 8'b0} : rf_b;

assign com = (stat[`IF] | addi_op | addiu_op | b_op | j_op ) ? `ALU_ADD:
                (ldi_op | ldiu_op | ldhi_op) ? `ALU_THB:
                (jr_op) ? `ALU_THA :  func[`SEL_W-1:0];

assign rf_c = ld_op  ? datain : jal_op ? pc : 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(alu_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));
```

次に ALU の A 入力 (alu_a) に pc を入れられるようにする。ちなみに IF 状態では必ず+1 する点に注意されたい。これに対応して、B 入力 (alu_b) には IF 状態では 1 を入れる。また、JMP, JAL 命令用に ir の下位を 11 ビットを符号拡張したものも入れられるようにする。結果としてこのマルチプレクサはかなり大きくなり、先に解説したバスの形に近くなる。ALU のコマンドにも、IF 状態では必ず加算が行われるように設定し、EX 状態では分岐、ジャンプ命令にも対応できるようにする。

### 2 サイクル POCO の合成

さて、この 2 サイクル POCO を合成してみよう。1 命令を 2 クロック掛けて実行することを考え、動作周波数を倍すなわち、周期を半分の 4nsec(250MHz) に設定してみる。web 上の poco_2c.tcl を用いて合成をしてみよう。結果として、動作周波数は達成できるが、面積は以下のようになる。

```
Combinational area:       38234.000000
Noncombinational area:    18272.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          56506.000000
Total area:                 undefined
```

結果として、1 サイクル POCO に比べて面積は増えてしまっている。pc 周辺の加算器がなくなり、面積が減るはずだったのに、なぜだろう？これには二つの原因が考えられる。

- ir やメモリのアドレスや ALU の入力にマルチプレクサを加えたことによりハードウェア量が増えた。
- 2 倍の動作周波数を実現するため、ALU に高速な演算回路が必要となりコストが増えた。

しかし、メモリは一つで済むので、全体としてコストは削減される。性能面では、CPI が倍になるが、周期が半分となり、結果として同じ性能が実現される。すなわち、MIPS 値は 250/2=125MIPS となる。

## 3 サイクル POCO

マルチサイクル化を推し進めることで、性能の向上を狙ってみよう。まず、動作周波数を上げるために、2 サイクル POCO のクリティカルパスを見てみよう。レジスタファイルを読み出し、演算を行い、その結果を格納するというパスがもっとも長いことがわかる。そこで、この遅延を切ってみることにする。すなわち、データをレジスタファイルで読み出してマルチプレクサを通した所で格納する。次のクロックで ALU で演算してレジスタファイルに格納する。このために現在の状態の他に ID(Instruction Decode) を設ける。

また、2 サイクル POCO で pc の加算を ALU で行うようにしたが、あまりコストの削減には効果がなかった。そこで、pc のカウントアップや分岐命令の飛び先の計算には、今まで通り専用の加算器を使ってしまうことにする。そして分岐系の命令は 2 クロックで終わるようにしよう。このようにして設計した 3 サイクル POCO の状態遷移を図 3 に示す。

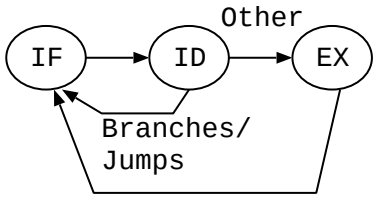


図 3: 3 サイクル POCO の制御回路

### 3 サイクル POCO

Verilog 記述を、以下のように変更して 3 サイクル POCO を実現する。今回は各命令は ID 状態と EX 状態のどちらで使われるか判らないため、以下のようにデコードする。

```
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 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);
```

次に ALU の入力の alu_a, alu_b、com をそれぞれレジスタ (reg_a,reg_b,reg_com) に格納する。

```
always @(posedge clk) begin
   if(stat[‘ID]) begin
     reg_b <= alu_b;
     reg_a <= rf_a;
     reg_com <= com;
```

```
  end
end
```

レジスタファイルへの書き込みは JAL 命令では ID 状態で pc を r7 に格納する。このため、書き込み信号は以下のようになる。

```
assign rf_c = ld_op  ? datain : jal_op ? pc : alu_y;
assign rwe = stat[‘EX] & (ld_op  | alu_op | ldi_op | ldiu_op |
                addi_op | addiu_op | ldhi_op) | (stat[‘ID] & jal_op) ;
assign cadr = jal_op ? 3’b111 : rd;
```

状態遷移は図 3 に示す制御となる。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) stat <= ‘STAT_IF;
   else
    case(stat)
     ‘STAT_IF: stat <= ‘STAT_ID;
     ‘STAT_ID: if(bnz_op | bez_op | bpl_op | bmi_op | jal_op |
                        jmp_op | jr_op ) stat <= ‘STAT_IF;
                else stat <= ‘STAT_EX;
      ‘STAT_EX: stat <= ‘STAT_IF;
    endcase
end
```

もちろん、stat は 3 ビットに拡張する必要がある。

pc は 1 サイクル POCO と同様に、専用の加算器を用いる設計に戻してやる。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(stat[‘IF])
     pc <= pc+1;
   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} ;
   else if (jmp_op | jal_op)
    pc <= pc + {{5{ir[10]}},ir[10:0]};
   else if(jr_op)
    pc <= rf_a;
end
```

残りの部分は 2 サイクル POCO と同じである。

### 3 サイクル POCO の合成

ではこの設計を合成してみよう。クリティカルパスにレジスタを入れて 2 クロック掛けて実行するようにした効果があって、クロックの設定をより厳しくして 3nsec でも slack が MET する。すなわち、この設計は 333MHz で動作

する。では CPI はどうなるだろう？BNE, BEZ, BMI, BPL, JMP, JR, JAL など分岐命令については 2、それ以外の命令では 3 クロックで動作する。したがって、CPI は実行するプログラムに依存する。

例えば分岐命令が全て合わせて 30%実行される場合を考える。この場合 CPI は以下のようになる。

$$CPI = 2 * 0.3 + 3 * 0.7 = 2.7$$

動作周波数 (MHz) を CPI で割ると MIPS 値を求めることができる。この場合、333/2.7= 123.3 となり、残念ながら 1 サイクル、2 サイクルの 125MIPS よりも遅くなる。分岐命令の確率が 35%の時は、125.7MIPS となり、少しだけ勝つことができる。

一方、面積を見てみよう。

```
Combinational area:       47794.000000
Noncombinational area:    21808.000000
Net Interconnect area:      undefined  (No wire load specified)
Total cell area:          69602.000000
Total area:                 undefined
```

となってかなり増えてしまう。これは、動作周波数を上げたため、高速で高コストな ALU を使う必要が生じたためである。

性能とコストを考えると、どうもこのままでは、この改造はあまりうまく行っていないようだ。しかし、この方法は、来年習得するパイプライン処理のベースとして重要である。

### 本日の課題

2 サイクル POCO に JALR 命令を取り付け、合成せよ。周波数を変えて最大動作遅延×面積ができるだけ小さくなるように工夫せよ。提出物は、Verilog 記述と合成した場合の最大動作遅延×面積。