マイクロコンピュータ基礎
マルチサイクルCPU

慶應大学
天野英晴

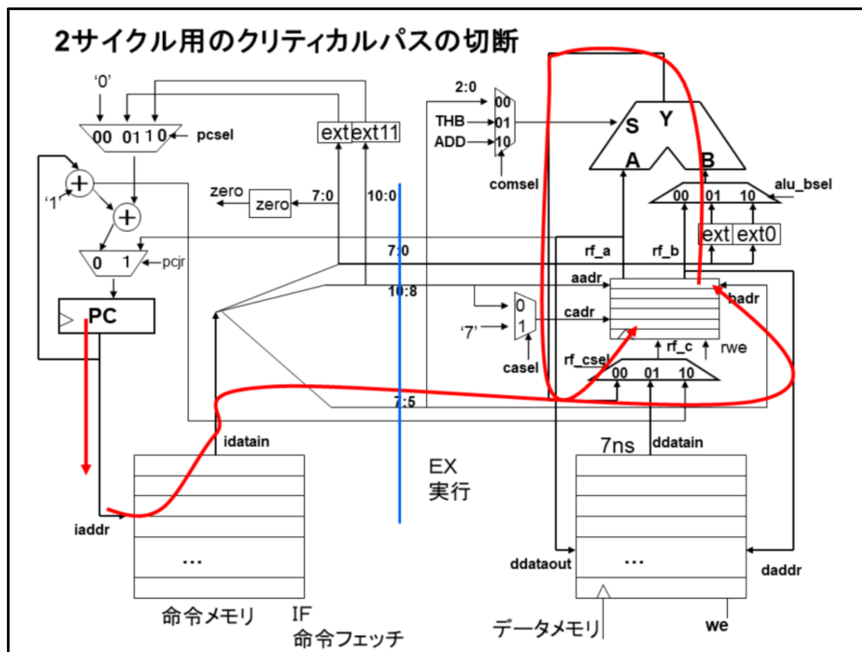
今まで紹介して来たシングルサイクルCPUは、実際のCPUに用いられることはほとんどありません。今回はより現実的にコストが安いマルチサイクルCPUを紹介します。

1サイクルCPUの問題点

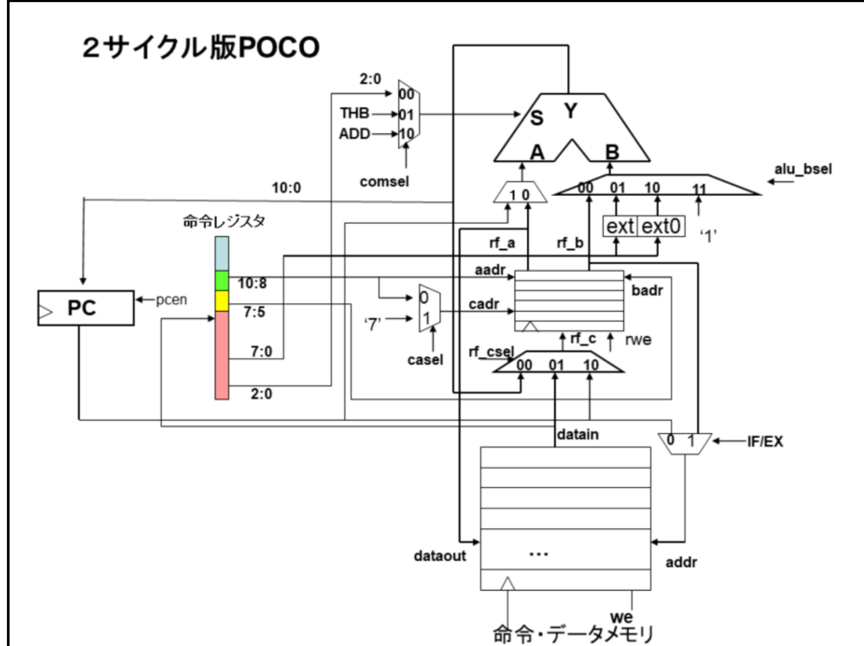
- クリティカルパスが長い
  - 全ての処理を1サイクルで実行
  - 全ての命令が最長の命令遅延に合わせる
    - 動作周波数を上げることが難しい
- 資源の共用ができない
  - ALUの使いまわしができない
  - 命令メモリとデータメモリの共用

→ マルチサイクル化

シングルサイクルCPUは、簡単で理解しやすいですがいろいろ問題点があります。まず、全ての処理を1サイクルで実行するため、クリティカルパスが長くなります。全ての命令が最長の命令遅延にあわせる必要があるので、動作周波数を上げることが難しいです。もう一つの問題点は、資源の共用ができないことです。1クロックで動作するためには全ての資源を独立に持たなければなりません。このため、ALUの使いまわしもできず、なにより命令メモリとデータメモリの共用ができません。コストという点ではこのメモリが一番大きく効きます。これを避けるための方法がマルチサイクル化です。すなわち、ひとつの命令を何クロックか掛けて実行します。

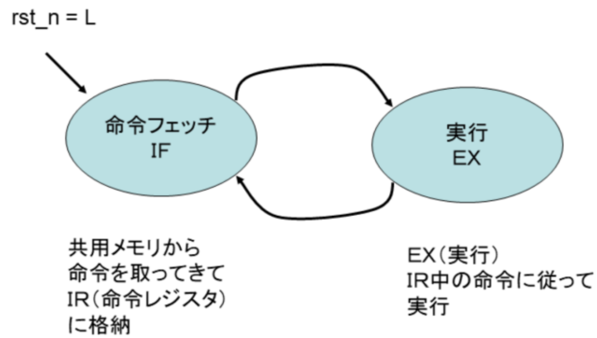


まずは一つの命令を2サイクルで実行することを考えましょう。この場合、現在のシングルサイクルCPUのクリティカルパスをなるべく真ん中で分割するのが有利です。今回、遅延時間が長いのはメモリとALUです。そこで、命令メモリから読み出したところで1サイクル、これを実行するところで1サイクルかけることにします。前者のサイクルを命令フェッチ（Instruction Fetch:IF)、後者のサイクルを実行（Execution)と呼ぶことにします。この分割によりメモリを共有することが可能になります。

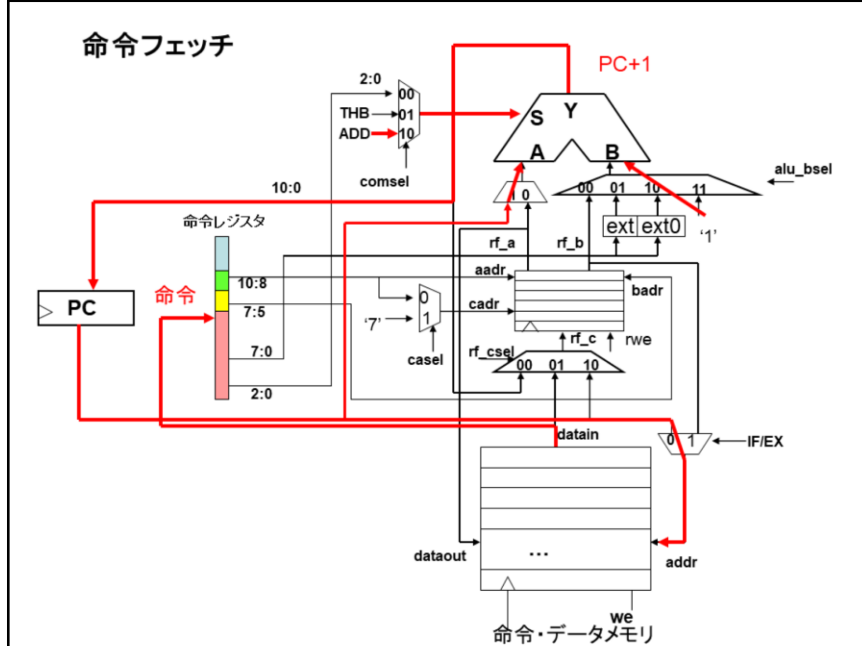


この図が2サイクル版POCOのデータパスです。命令メモリとデータメモリは単一のメモリに統合します。アドレスにマルチプレクサを付け、命令フェッチの時はPCが、実行サイクルではアドレスの入ったレジスタが繋がるようにしています。命令を格納するためのレジスタ、命令レジスタ（Instruction Register:IR)が付加されています。他にもALUの入力のマルチプレクサが拡張され、使いまわすことができるようになっています。PC周辺の加算器がなくなってすっきりしているのが分かります。

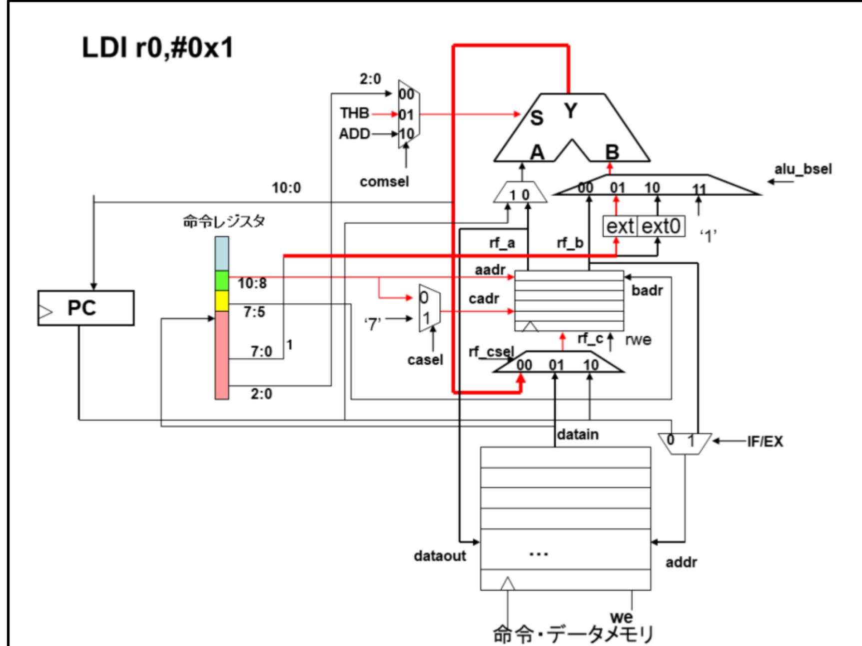
コントローラの状態遷移



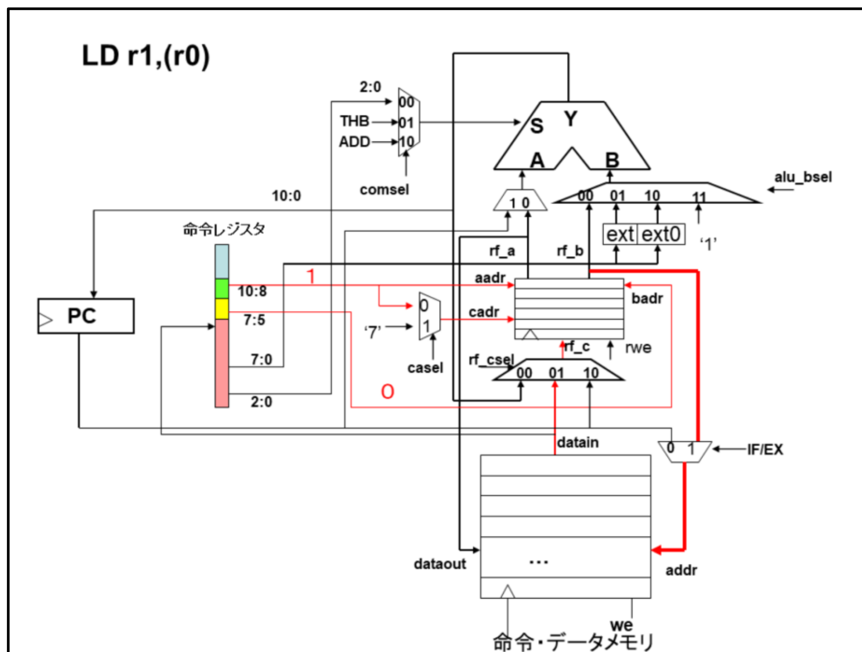
コントローラの状態遷移は大変簡単です。命令フェッチIFと実行EXが順に遷移するだけです。IFで命令をフェッチして命令レジスタIRに格納し、EXでIR中の命令に従って実行します。



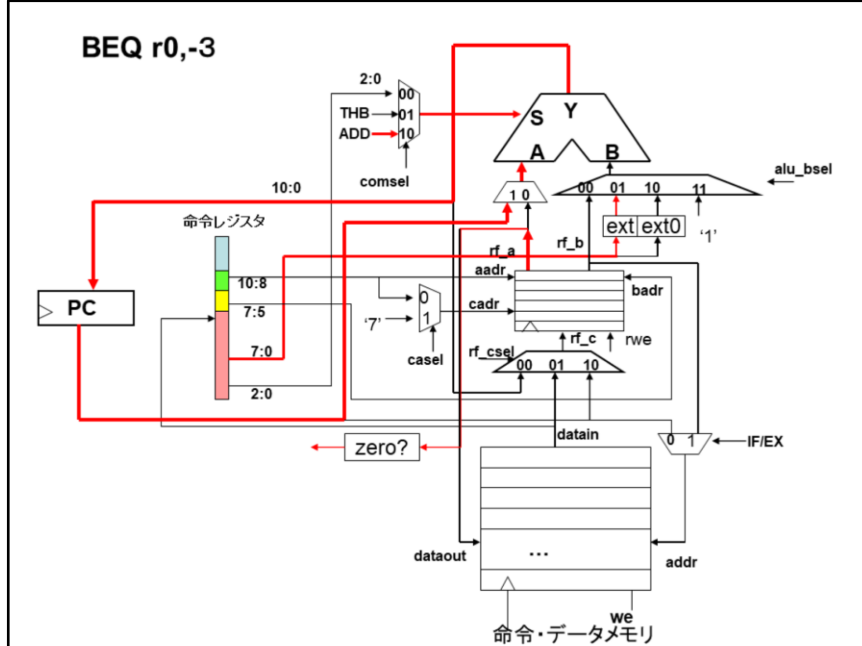
では、このデータパスがどのように働くか見てみましょう。まず命令・データメモリのアドレスにはIF/EXのマルチプレクサを制御してPCをつなぎます。読み出して来た命令は命令メモリに格納します。この間PCをALUのA入力に入れ、1をB入力に入れます。コマンドにはADDを入れてやります。ALUの出力からはPC+1が出力されるので、これをPCに取り込みます。PCはイネーブル付きのレジスタにしておき、必要なときだけデータをセットするようにしておきます。これで実行時はPCはPC+1になっています。



取ってきて命令レジスタに入った命令がLDI命令であったとすると、次のEX状態では以下のように動きます。命令の下8ビットが符号拡張されて、ALUのB入力に入ります。コマンドはTHBが入るのでA入力は関係なく、出力には符号拡張されたイミーディエイト値、すなわち1が入ります。レジスタファイルのマルチプレクサをALUからの出力を通してやり、レジスタファイルのrdつまりここではr0に1を書き込みます。

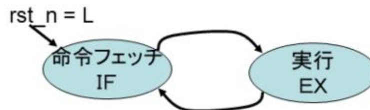


次にフェッチされた命令がLD命令ではどのように働くでしょうか？この場合rd,rsフィールドから1，0がレジスタファイルのアドレスに入り、r0がrf_bから読み出されます。マルチプレクサは今度はPCではなく、rf_bをaddrに与えます。読み出したデータはdatainからマルチプレクサを経由してrf_cに入り、cadr=1すなわちr1に書き込まれます。ちなみにST命令もほぼ同様なステップで実行され、rf_aからのデータがwe=1にすることによりメモリに書き込まれます。



では、分岐命令はどのように働くでしょうか？IFでPCに1加えたのと同様、PCをALUのA入力に入れます。命令レジスタの下8ビットを符号拡張してB入力に入れます。コマンドにはADDを入れれば飛び先が計算されます。IFが終わった段階でPCは1増えているので、PC+1に命令コードの下8ビットの符号拡張が加えられることになります。実は、これがPC+1が飛び先の起点となっている一つの理由です。

状態遷移の
Verilog記述




```
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
```

では、2サイクル版POCOのVerilog HDL記述を解説します。まず状態遷移のコントローラを記述します。今、状態を入れておくレジスタとしてstatというレジスタを定義します。IF状態はSTAT_IF、EX状態はSTAT_EXとします。状態遷移は、クロックの立ち上がりで遷移し、通常のレジスタの記述通りのalways文を使います。今回の状態遷移は簡単で、現在の状態がIFならば次の状態はEX、現在がEXならば次はIFとなるだけです。もっと複雑な状態遷移も同様のcase文で書くことができます。

One-Hot Counter

- 状態数だけビットを使う

○どの状態にあるかを検出するのが容易

stat[`IF] が1ならばIF状態、stat[`EX]が1ならばEX状態

○割り当てが容易

×ビット数が多い

△状態遷移が2ビット変化

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

今回、状態の割り当てには、One-Hotカウンタという方法を使っています。この方法では状態の数だけビット数を設け、ある状態に居るときに、これに対応するビットを1にします。今回2状態なのでstatを2ビットのレジスタとし、01をIF、10をEXとして割り当てます。すなわち、statの0ビット目が1ならばIF、1ビット目が1ならばEXであることが一発でわかります。ここでは、それぞれの状態の名前をSTAT_IF,STAT_EXとし、それぞれ01、10を割り当てます。0ビット目の0をIF、1ビット目の1をEXと定義すれば、stat[`IF]が1ならばIF、stat[`EX]が1ならばEX状態であることがわかります。この表記で、それぞれの状態になっていることを表します。皆さんは順序回路設計の時間に2進数を状態に割り当てる方法（2進カウンタ）を習ったと思いますが、One-Hotカウンタはこれに比べてビット数が多い一方、割り当てが簡単で、状態の判断が高速に可能です。なにしろ対応するビットが1かどうか見ればよいので。違った状態への遷移には常に2ビットの変化を伴います。これは、1ビット変化ですべての状態を遷移させることができるジョンソンカウンタよりは不利ですが、多くのビット数が一度に変化する可能性がある2進カウンタよりは優れていると言えます。

mem.datの実行シミュレーション

- mem.datは、0x10番地の内容と0x11番地の内容の掛け算を行うプログラム
- 命令とデータが混在しているところに注意！

make poco_2c.out

./poco_2c.out | moreでシミュレーション可能

- 状態が表示されるので注目
- 2クロックで1命令なのに注意
- memの値は0x10番地以降を表示してある

では、2サイクル版のPOCOをシミュレーションしましょう。今まで命令メモリ用の初期設定ファイルをimem.dat,データメモリ用をdmem.datとしてきました。今回は統合メモリですので、mem.datという名前で呼ぶようにしました。シミュレーションを行うファイルはmake poco_2c.outと打ち込めば生成されます。./poco_2c.out | moreでシミュレーションをすることが出来ます。今回状態が表示されるのに注目してください。01がIF、10がEXです。2クロックで1命令なので、実行には時間が掛かるような印象があると思いますが、本当はクロック周波数が上がっているので、一概に遅くなったとはいえません。memの値は0x10番地以降を表示してあります。

2サイクル版POCOのVerilog記述

- 状態に応じて制御を切り替える
  - デコード信号はEX状態でのみ有効にする
- やや見難くなってしまうが、stat[`IF]と stat[`EX]に注目！

では、2サイクル版のVerilog記述を紹介します。今回の記述は状態に応じてやることを変えています。stat[`IF]が1の時はIF状態、stat[`EX]が1の時はEX状態でのみ有効な信号です。たとえば、デコード信号はEX状態のみで有効になるようになっています。

```
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];
```


では実際の記述を見ましょう。命令メモリ、データメモリが共有されているので、入出力は減っています。datain, dataoutがメモリのデータ入出力、addrはメモリのアドレスです。新たに命令レジスタirと状態statを定義します。メモリに対するアドレスには、状態がIFならばPCが、そうでなければrf_bが出力されます。今回は、命令の下位11ビット分もjimmという名前にしています。

実行状態でのみ
デコード結果は有効

```
// 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 ;
```

デコード信号は1サイクルpoco同様、I型命令のopcode、R型命令のfuncを見て判断します。1サイクル型との相違は、stat[`EX]とANDすることで、これが有効になるのがEX状態だけにしています。ちなみに、分岐命令全体をまとめてb_op、ジャンプとJALをまとめてj_opとして定義しています。

```
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));
```


赤い枠がALU周辺、青い枠がレジスタファイル周辺です。次のページの図と対応しましょう。A入力は、IF状態と、分岐、ジャンプ系の命令ではPCが入り、その他の命令ではレジスタファイルのAポート（rf_a）が入ります。B入力は結構大変です。ADDI、LDI、分岐命令の時は命令の下位8ビットが符号拡張、ADDIU、LDIUでは下位8ビットがゼロ拡張、ジャンプ系ならば下位11ビットが符号拡張されて入ります。LDHIは、上位8ビットにイミーディエイト値を入れます。どれでもなければレジスタファイルのBポート（rf_b)が入ります。コマンドはIF、ADDI、ADDIU、分岐、ジャンプ系の場合はADDコマンドを、LDI、LHIU、LDHIではBをスルーするTHB、jr命令は、rf_aの値をスルーするためTHAを入れます。そうでなければファンクションコードの下位3ビットを入れます。
青い枠はレジスタファイルです。レジスタファイルのC入力の記述、rwe、Cポートのアドレスなど、レジスタファイル周辺の記述はシングルサイクル版と同じです。
最後はALUとレジスタファイルの実体化と入出力の接続です。これもシングルサイクル版と同じです。

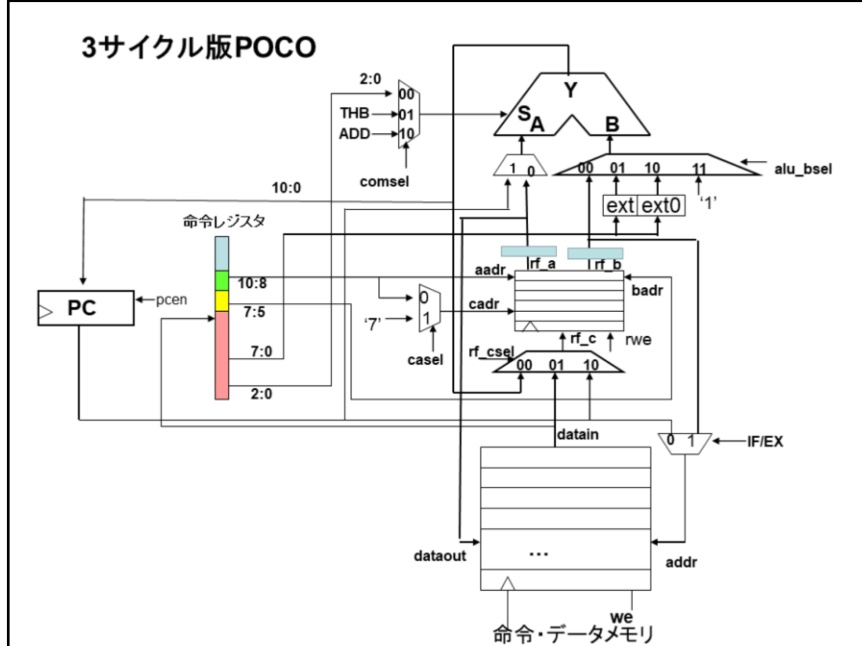
２サイクル版POCO

前のページの記述と対応させてみてください。

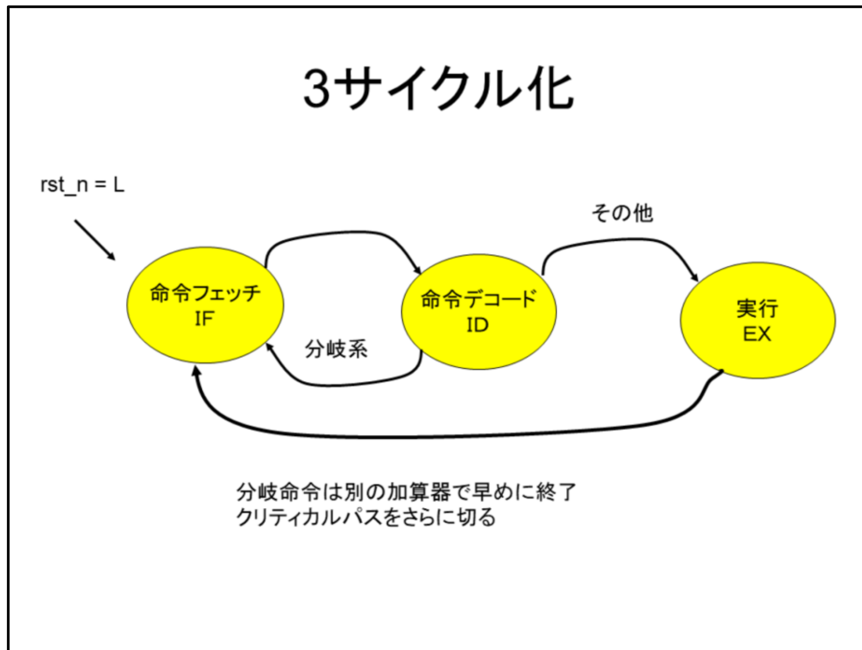
実行状態でのみ
デコード結果は有効

```
assign pcset = stat[`IF] |   (bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0) |
          (bmi_op & rf_a[15] == 1 ) | (bpl_op & rf_a[15] == 0) | j_op | jr_op ;

always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if(pcset) pc <= alu_y;
end

always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) ir <= 0;
  else if(stat[`IF])
    ir <= datain;
end
```

最後はPC周辺です。今回PCの入力はALUの出力のみに整理しています。このためにJR命令もTHAを使ってレジスタファイルのAポートからの出力をALUの出力に送っています。pcsetという信号を定義して、これが１の時にはaly_yがPCに入るようにします。したがって、always文内の記述は大変簡単です。pcsetは分岐する命令のみ１になるようにそれぞれの場合をORしてやります。
次は、命令レジスタirの記述ですが、これも簡単です。リセット時には0にし、IF状態の時にだけ読んで来たdatainをirにセットします。実際のpoco.vは、この後に状態遷移の記述がありますが、これは以前に紹介しましたので省略します。POCOの2サイクル版は比較的シンプルな構造になっているのが分かります。



さらに、この考え方を発展させると、3サイクル版もできます。2サイクル版のPOCOでクリティカルパスが最も長いのは、レジスタファイルからデータを読んで来てALUで計算してその答えをレジスタファイルにしまうまでのパスです。もうひとつはレジスタファイルからデータを読んで来て、これをデータメモリのアドレスに与え、これによりデータを読み出し、レジスタファイルに書き込むまでのパスです。これは両方共、レジスタパスの出力にレジスタを入れることで分割することができます。レジスタファイルに読み出すまでのサイクルをレジスタフェッチ（RF)あるいは命令デコード（ID)と呼びます。これはレジスタを
読み出す間に命令の判別を行うためです。ALUで計算を行うかメモリからデータを読み出すサイクルをEXと呼ぶことにします。



3サイクルPOCOの状態遷移はこの図のようになります。IDを付け加えることにはもう一つメリットがあります。それは分岐系の命令は、レジスタファイルから読み出すデータに基づく分岐判定をうまくやると、2サイクルで終わります。このため、命令によって実行時間が変わります。

状態遷移の記述

```
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
```

2サイクル版の状態遷移は簡単すぎたので、3サイクル版の記述を示します。IF状態の次はID状態になり、ここで、分岐系の命令ならばIF状態に戻り、そうでなければEX状態に進みます。EX状態の次はIF状態に戻ります。図の通りに記述されているのが分かります。

演習

2. ２サイクル版のPOCOを前回と同様の環境で論理合成して、動作周波数、Utilization、電力を評価せよ。

mvivado.tarを利用せよ