第8回　汎用レジスタマシンの構造

POCO の内部構造

POCO のデータパス

前回の命令を実現できる POCO を設計してみよう。演算を行うデータパスは、基本的な構造はアキュムレータマシンと同じだが、以下の違いが生じる。

- アキュムレータの代わりにレジスタファイルを使う。レジスタファイルは図 1 に示す 3 ポート構造を想定する（今の POCO なら実は読み出しの a ポートアドレスと書き込みの c ポートアドレスは共有可能である）。レジスタファイルは一種のメモリであり、aadr から与えられた番号（3 ビット）のレジスタが内容が rf_a に読み出され、badr に与えられた番号（3 ビット）のレジスタの内容が rf_b に読み出される。一方、書き込みについては、cadr に与えた番号のレジスタに対して、rwe が 1 の時のクロックの立ち上がりで、rf_c からのデータが書き込まれる。aadr, badr, cadr は ir 中の命令でレジスタを表すフィールドにより指定する。
- register-register マシンなので、メモリから ALU への直接パスは存在しない。メモリから読み出したデータは、マルチプレクサを介してレジスタファイルの rf_c から書き込まれる。
- プログラムカウンタ相対分岐を使うため、分岐命令実行時には、PC の値に命令のイミーディエイトフィールドの値を符号拡張して加算する必要がある。

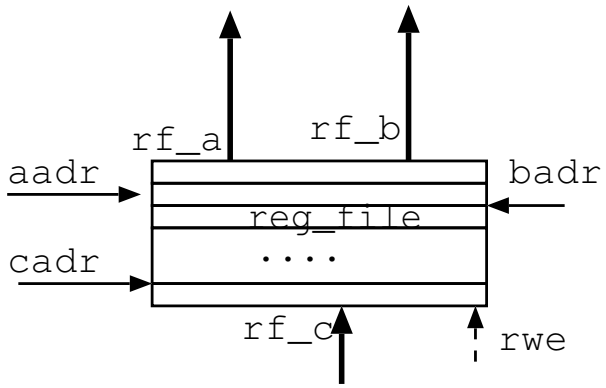

図 1: レジスタファイル

図 2 に、前回の命令を実行する POCO のデータパスを示す。命令が複雑になった分、マルチプレクサを各所に設けて命令に応じてデータの流れを切り替えている。アキュムレータのマルチプレクサは制御線が 0 か 1 かによって二つのデータの流れを切り替えるものだけだったが、今回は 4 つから 1 つを選ぶマルチプレクサも使っている。この場合、制御信号が 00 ならば 00 入力が、01 ならば 01 入力、10 ならば 10 入力が出力される。今回は 11 入力は使っていないので実際は 3 つの入力から一つを選んでいる。

図中の太線は 16 ビットデータ、細い点線は制御信号、それ以外の線は 16 ビットよりも短い 3 ビットや 8 ビットのデータを示す。また、命令フェッチ時に主に動作する部分は、太い点線で表している。ext と書かれた箱は、符号拡張を行うハードウェアモジュールを指す。符号拡張は MSB を複製してくっつければ良いためハードウェアとしては大変簡単である。ext0 は、0 拡張ハードウェアモジュールで、上位に 0 を補うだけの単純な働きをする。＝ 0？と書かれた箱は、読み出したレジスタの値が 0 と等しいかどうかを判断するものでこれも OR 回路のアレイでできた簡単なものである。

この図はデータパスを示すため、制御回路は描かれていない。点線の入出力は制御回路に接続されており、この動作は後に解説する。

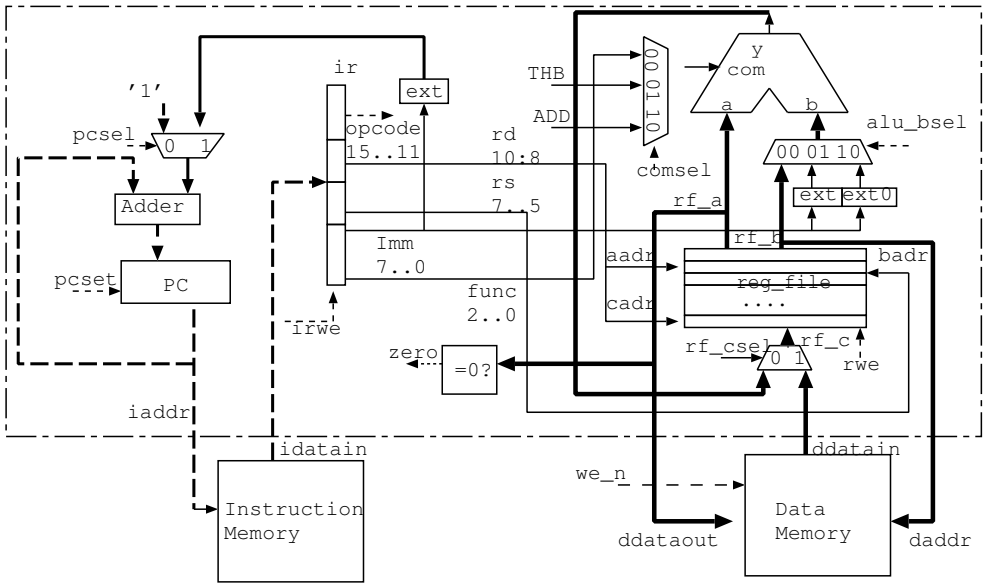

図 2: POCO のデータパス

命令フェッチでの動作

それでは、図のデータパスがどのように動くかを解説しよう。まず、アキュムレータマシンと同じく、命令フェッチと実行の二つの状態間を交代に遷移する方式を考える。

命令フェッチでは、この動作はほとんどアキュムレータマシンと同じである。

pc の出力は命令メモリのアドレス iaddr に常に接続されているので、idatain から読み出された命令を ir に格納すれば良い。さらに、pc を一つカウントアップしてやるので、加算器の入力は'1' を選択する。このためには、irwe=1, pcset=1, pcsel=0 とすれば良い。

命令フェッチ時には、演算用のデータパスは動作していないため、演算部の制御信号はどのようなものでも良い。しかし、レジスタファイルやメモリに書き込みを行ってはまずいので、rwe=0, we_n=1 としておく必要がある。

実行時の動作

実行状態では、ir に命令が格納されており、それは今までの所、図 3 に示す二つのタイプに分類できる。まず、I 型は、5bit の opcode(15bit 目-11bit 目)、3bit のディスティネーションレジスタ rd(11bit 目-8bit 目)、8bit の Imm 部 (7bit 目-0 ビット目) を持つ。LDLI, ADDI などのイミーディエイト命令、BNZ などの分岐命令がこのタイプに相当する。

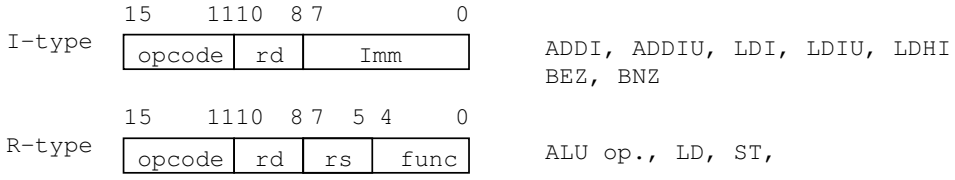

図 3: 命令のフィールド構成

一方、R 型は、5bit の opcode(15bit 目-11bit 目)、3bit のディスティネーションレジスタ rd(11bit 目-8bit 目)、3bit のソースレジスタ rs(7bit 目-5bit 目)、5bit の func(4bit 目-0bit 目) を持つ。

ここで、ir の rd 部 (11bit 目-8bit 目) は、レジスタファイルの a ポートアドレス (aaddr) と c ポートアドレス (caddr) に、rs 部 (7bit 目-5bit 目) は、レジスタファイルの b ポートアドレス (baddr) に接続しておく。R 型の命令がフェッチ

されれば、a ポート、b ポートの両方からレジスタが読み出されて演算に使われ、結果は rd の示すレジスタに c ポートから書き込まれる。一方、I 型の命令がフェッチされた場合は、b ポートから読み出されたデータは意味がないことになるが、この場合は使わなければ良いので害はない。

同様に、ir の imm 部 (7bit 目-0bit 目) は、符号拡張ユニット exp,0 拡張ユニット exp0 に常に送られるが、これも必要でないときは使わなければ良いので害になることはない。このようにハードウェアの設計は、準備はしておいて、必要になったら使う、という方針で行う場合が多い。これは余分なマルチプレクサが不要となり、動作速度も向上するからである。しかし、一方で、利用しない回路が動作するため消費電力の点では不利となる。

ADDI 命令の実行

では、まず、ADDI 命令を実行してみよう。この場合は、I 型であり、符号拡張を行うため、alu_bsel=01 として、符号拡張された Imm を ALU の b 入力に入れる。ALU の a 入力には rd フィールドで rd が読み出される。ALU の操作は加算なので、コマンドを選択する comsel=10 として ADD 命令のコード 110 を入れてやる。答えはレジスタファイルに書き込むので、rf_csel=0 として ALU の出力を通してやり、rwe=1 として書き込みを行う。we_n=1, pcset=0, irset=0 など、使わないレジスタやメモリへの書き込みは行わないようにしてやらなければならない。この様子を図 4 にこの様子を示す。

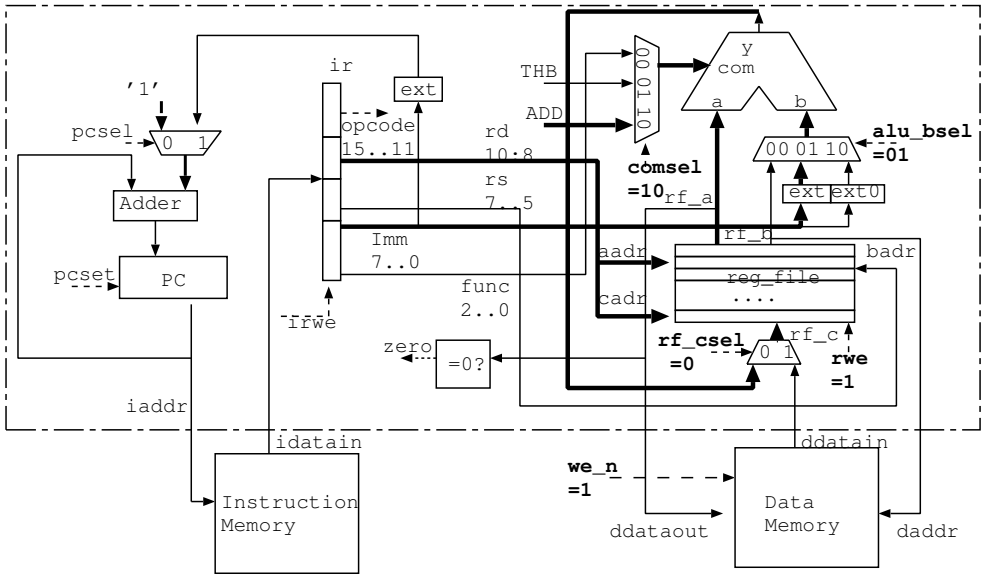

図 4: ADDI 命令実行のデータパス

例題 1: LDI、LDIU、ADDIU は comsel、alu_bsel を変えることで同様に実現できる。どのようにすればよいか？

例題 2: このデータパスでは LDHI 命令は実行できない。実行できるように拡張せよ。

SUB 命令の実行

次は R 型の SUB 命令を実行してみよう。この場合は、R 型なので、レジスタ同士の演算となる。そこで alu_bsel=00 として、レジスタファイルの b 出力から rs フィールドで指定されたレジスタを入れてやる。ALU の操作は減算で、これはコード中の func フィールドの下 3 ビットを入れてやる。答えはレジスタファイルに書き込むので、rf_csel=0 として ALU の出力を通してやり、rwe=1 として書き込みを行う。we_n=1, pcset=0, irset=0 など、使わないレジスタやメモリへの書き込みは行わないようにしてやらなければならない。この様子を図 5 にこの様子を示す。この場合、func フィールドでコマンドが決るので、全く同じデータパスで、レジスタ同士の演算命令がすべて実現できる。

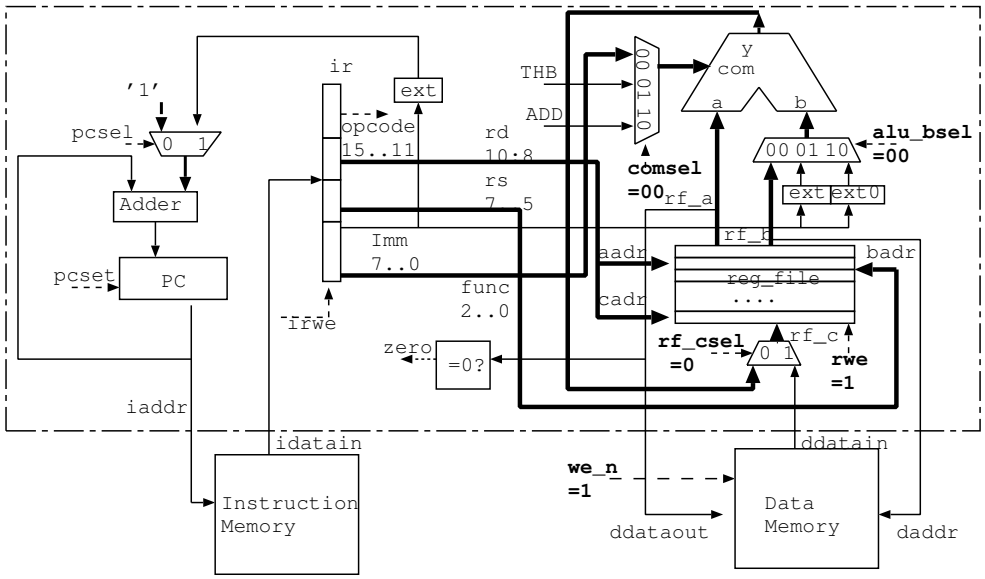

図 5: SUB 命令実行のデータパス

LD/ST 命令の実行

LD 命令も R 型なので、rd フィールド、rs フィールドの両方を用いるが、a ポートから読み出された rd は使われない。b ポートからの rs はデータメモリのアドレスに接続されているので、そのアドレスからデータを読み出してレジスタファイルに書き込む。このため、rf_csel=1 としてメモリからのデータを通してやり、rwe=1 とする。図 6 にこの様子を示す。

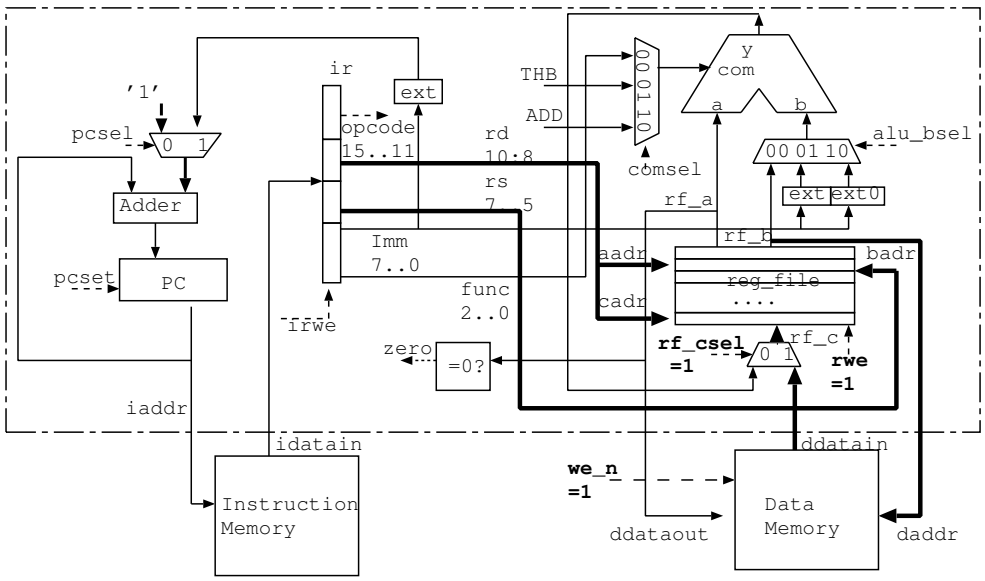

図 6: LD 命令実行のデータパス

ST 命令は、a ポートから読み出された rd をデータメモリの入力に送ってやり、これを書き込む。このため、データメモリの書き込み制御線 we_n=0 とし、レジスタに書き込んではいけないので、rwe=0 とする。データメモリの書き込み信号はアクティブ L と仮定した点に注意されたい。図 7 にこの様子を示す。

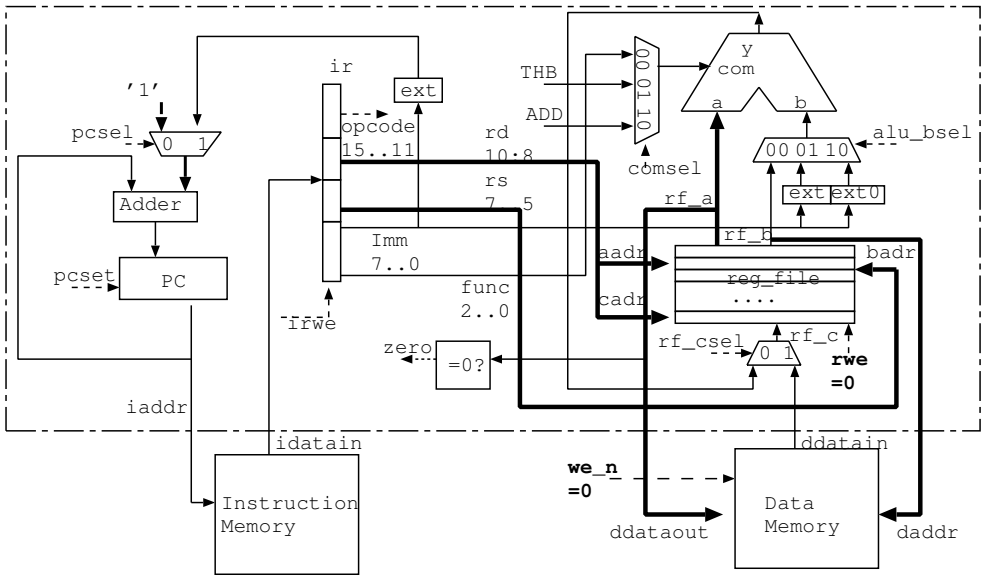

図 7: ST 命令実行のデータパス

BNZ/BEZ 命令の実行

BEZ 命令は、a ポートから読み出された rd を 0 かどうかチェックする。このデータパスでは ALU を用いず Adder を用いる。Adder は命令フェッチで PC をインクリメントするのに使うが、実行状態では暇なので、これを利用するのが自然である。もっとも全てを ALU で実行するように改造すればこの Adder 自体が不要になるのだがこれは後の演習に譲ることにする。ここでは Imm フィールドを符号拡張し、これを pcsel=1 として、これを pc に設定し、条件が成立すれば pcset=1 として pc に飛び先を設定する。条件が成立するかどうかは制御回路で簡単に判断できる。図 8 にこの様子を示す。

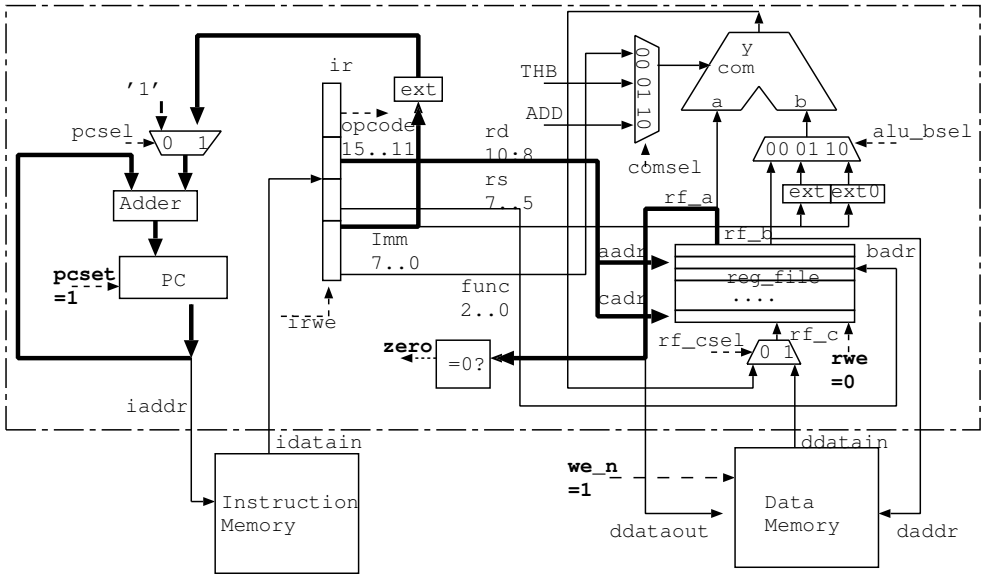

図 8: BEZ 命令実行のデータパス

制御回路の構成

制御回路は、状態遷移をコントロールする部分と、状態に合わせて制御信号を発生する部分で構成する。状態は二つしかないので、順序回路は非常に単純で済む。制御信号は、今までそれぞれの実行データパスで調べて来た値を発生するように組み合わせ回路を設計してやる。組み合わせ回路は一種の表なので、以下の表をハードウェア化してやれば良い。

表 1: 各命令の制御信号

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

例題 3: LDI、LDIU、ADDIU を表に付け加えよ。

Verilog での記述

最初に図 1 の Verilog 記述は以下のようになる。

```
‘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] rf[0:‘REG-1];
assign a = rf[aadr];
assign b = rf[badr];
always @(posedge clk)
if(we) rf[cadr] <= c;
endmodule
```

これは、説明の必要はあまりないと思う。さて、これを使って、図 2 を比較的そのまま Verilog で記述したものである。

```
‘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_n);

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_b, alu_y;
wire [`OPCODE_W-1:0] opcode;
wire [`OPCODE_W-1:0] func;
wire [`REG_W-1:0] rs, rd;
wire [`SEL_W-1:0] com;
wire [`IMM_W-1:0] imm;
wire pcsel, rwe;
wire st_op, bez_op, bnz_op, addi_op, ld_op, alu_op;
wire ldi_op, ldiu_op, ldhi_op, addiu_op;

assign ddataout = rf_a;
assign iaddr = pc;
assign daddr = rf_b;

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

// Decorder
assign st_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_LD);
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 we_n = ~st_op;
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;

assign com = (addi_op | addiu_op ) ? `ALU_ADD:
(ldi_op | ldiu_op | ldhi_op) ? `ALU_THB: func[`SEL_W-1:0];
assign rf_c = ld_op  ? ddatain : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op ;
assign pcsel = (bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0);
```

```
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(rd), .we(rwe));

always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= pc +{{8{imm[7]}},imm} ;
   else if(stat[`IF])
     pc <= pc+1;
end

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

endmodule
```

アキュムレータマシンと同様に。状態は、レジスタ stat で記憶し、01 で命令フェッチ、10 で実行を表す。このため、IF を 0,EX を 1 とすると、stat[`IF] は命令フェッチに居ること、stat[`EX] は実行状態に居ることを示す。

まず、デコーダの部分を説明しよう。命令レジスタの各フィールドを以下のように分離する。

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

次に opcode で命令を判別する。これが以下の部分で、これをデコーダと呼ぶ。このデコーダの結果、命令の種類が分かるので、表 1 の信号を簡単に発生できる。

```
// Decorder
assign st_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = stat[`EX] & (opcode == `OP_REG) & (func == `F_LD);
assign 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 we_n = ~st_op;
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;

assign com = (addi_op | addiu_op ) ? ‘ALU_ADD:
(ldi_op | ldiu_op | ldhi_op) ? ‘ALU_THB: func[‘SEL_W-1:0];
assign rf_c = ld_op  ? ddatain : alu_y;
assign rwe = ld_op  | alu_op | ldi_op | ldiu_op | addi_op | addiu_op | ldhi_op ;
assign pcsel = (bez_op & rf_a == 16’b0 ) | (bnz_op & rf_a != 16’b0);
```

Verilog はマルチプレクサを ? : で表す点に注意されたい。表 1 の制御信号はデコーダの命令により、?の前の条件として表されている。3 入力以上のマルチプレクサは?を 2 回使って条件を順番にチェックしている。

次に以下の記述は、ALU とレジスタファイルの周辺の接続である。

```
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(rd), .we(rwe));
```

最後の部分はレジスタの記述である。まずは pc である。

```
always @(posedge clk or negedge rst_n)
begin
   if(!rst_n) pc <= 0;
   else if(pcsel)
     pc <= pc +{{8{imm[7]}},imm} ;
   else if(stat[‘IF])
     pc <= pc+1;
end
```

always 文中では、if 文が使えるので、pc の前段のマルチプレクサは if 文で表現されている。また、ここでは+が二ヶ所に表われるが、条件が排他的なので、通常は合成時に一つの加算器が使い回され、図 2 の回路が生成される。

次は、ir だが、これは命令フェッチでは常にセットされるので簡単である。アキュムレータマシンと同じで、irset 信号は明示的に表われない点に注意されたい。

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

最後に状態遷移である。これもアキュムレータマシンと同じである。

```
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
```

さて、この記述では、マルチプレクサが always 文の中では if 文で、組み合わせ回路中では? :で表わされ、統一が取れていない。記述は確かに図 2 に対応しているが、図がなければ必ずしも分かりやすいものではない。この辺、Verilog にはさまざまな記述法があり、この点は後の回で解説する。

演習 8-1

BMI 命令は BEZ 命令と同じフォーマットで、レジスタの内容がマイナス（つまり最上位ビットが 1）の時分岐が成立する。この命令を Verilog 記述に付け加えよ。ただし、opcode を 10011 とせよ。これで、web 中のテストプログラムが使える。

演習 8-2

JMP 命令は、レジスタの内容をチェックしないで、opcode 以外の 11 ビットを相対的な飛び先としてジャンプする命令である。この命令を Verilog 記述に付け加えよ。また、合成してゲート数を求めてみよ。

2 つの演習は全て、12/23 までにやれば良い。poco.v のみを別々のメールで提出すること。次回 12 ／ 9 はまたしても 4 限休講で、5 限はできていない人のための演習とする。