第7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。

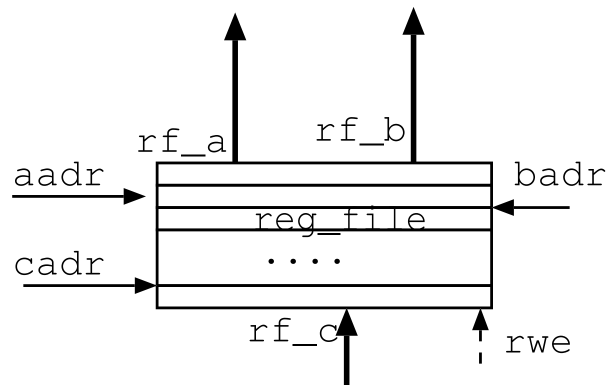

図 1: レジスタファイル

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。

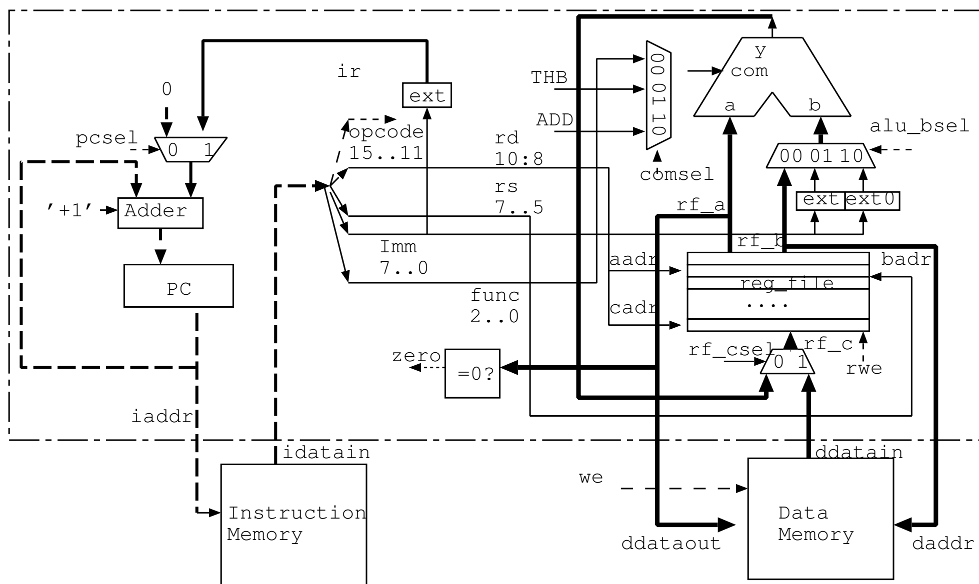

図 2: POCO のデータパス

POCO 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、PC は、命令メモリのアドレスに接続されており、PC 内のアドレスから命令が読み出される。命令を読み出して CPU 内に持ってくる操作を命令フェッチ (Fetch) と呼ぶ。Fetch は行って取ってくるという意味があり、遠くにあるメモリに PC からアドレスを送って、取ってくるという雰囲気を表している。コンピュータの命令は必ずフェッチすると呼び、ゲットするとは言わない。

pc には専用の加算器が付いていて、毎回必ず+1 する。分岐命令が成立した場合は、pcsel を 1 として相対番地を加え、そうでない場合は 0 を加える。この構成は一見奇妙なようだが、実は加算器は一番下の桁上げ入力が空いており、ここに 1 を入れて加算するのはハードウェアが不要なため、時々使われる。

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 から命令は読み出されてくる。

さて、この命令は、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。

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、命令の 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、命令の 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は、準備はしておいて、必要になったら使う、という方針で設計する場合が多い。このようにすると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。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=0 とし、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 3 に示す。

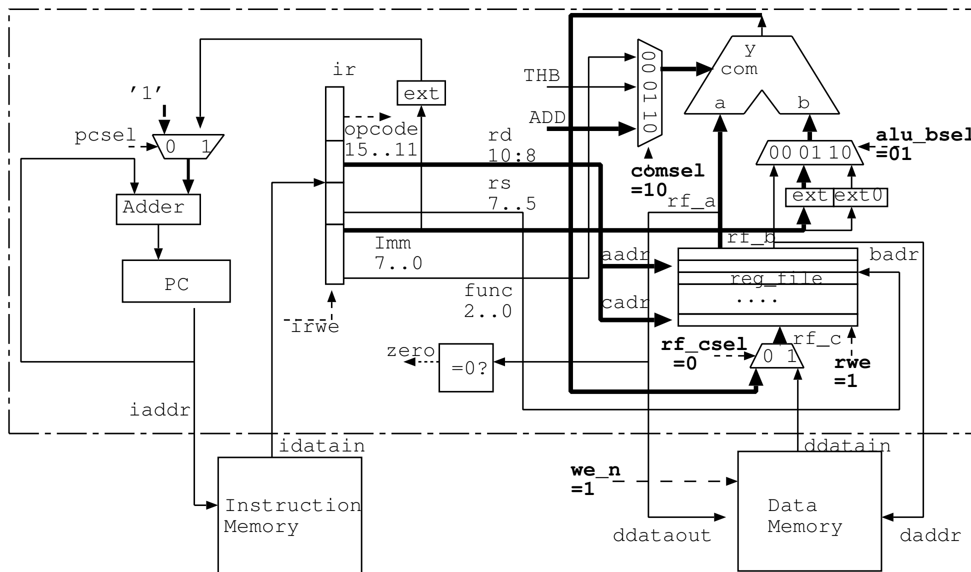

図 3: ADDI 命令実行のデータパス

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。レジスタファイルの b 出力には rs フィールドによって指定されたレジスタが現れる。そこで alu_bsel=00 とすることで、このレジスタファイルの b 出力を、ALU の b 入力に入れてやる。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=0 とし、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。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。

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 5 にこの様子を示す。

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=1 とし、レジスタに書き込んではいけないので、rwe=0 とする。図 6 にこの様子を示す。

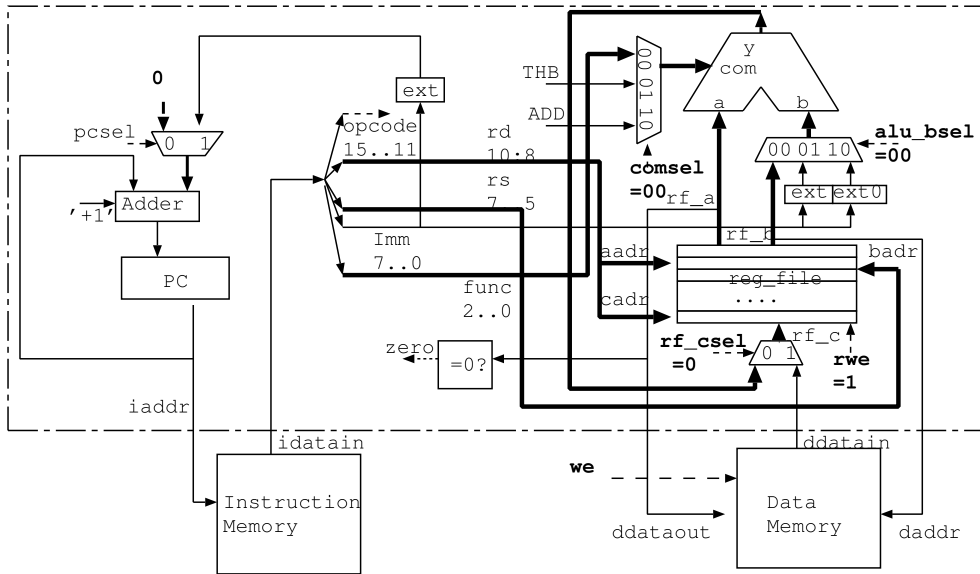

図 4: SUB 命令実行のデータパス

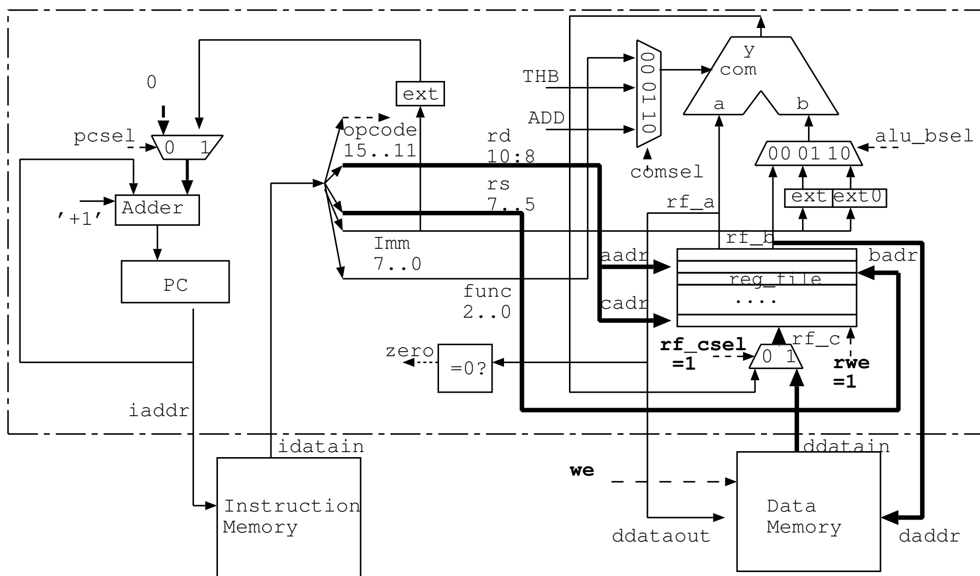

図 5: LD 命令実行のデータパス

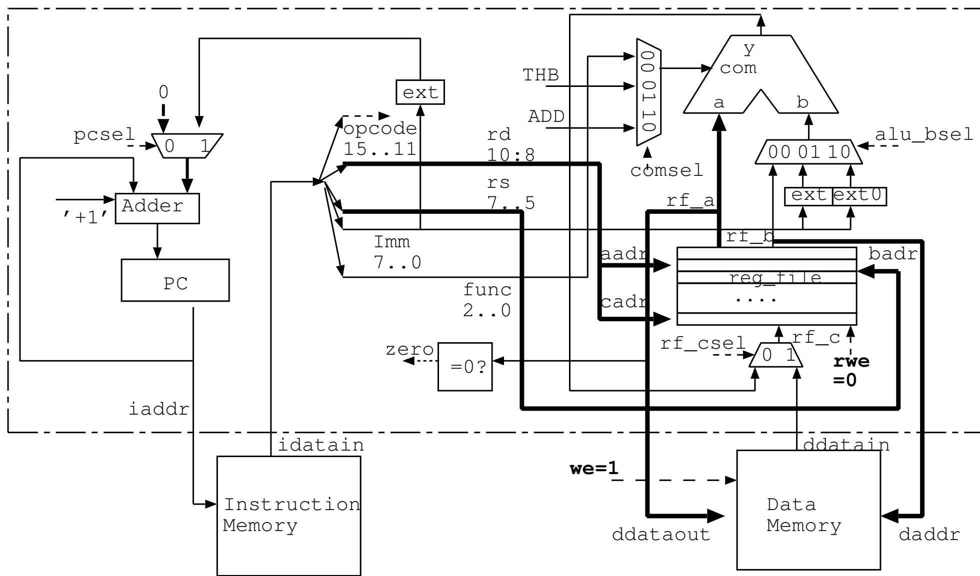

図 6: ST 命令実行のデータパス

BNZ/BEZ 命令の実行

BEZ 命令は、a ポートから読み出された rd が 0 かどうかチェックする。条件が成立、つまりこの場合 0 だった時に pcsel=1 として、Imm フィールドを符号拡張した飛び先を加算器に入れてやる。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 7 にこの様子を示す。

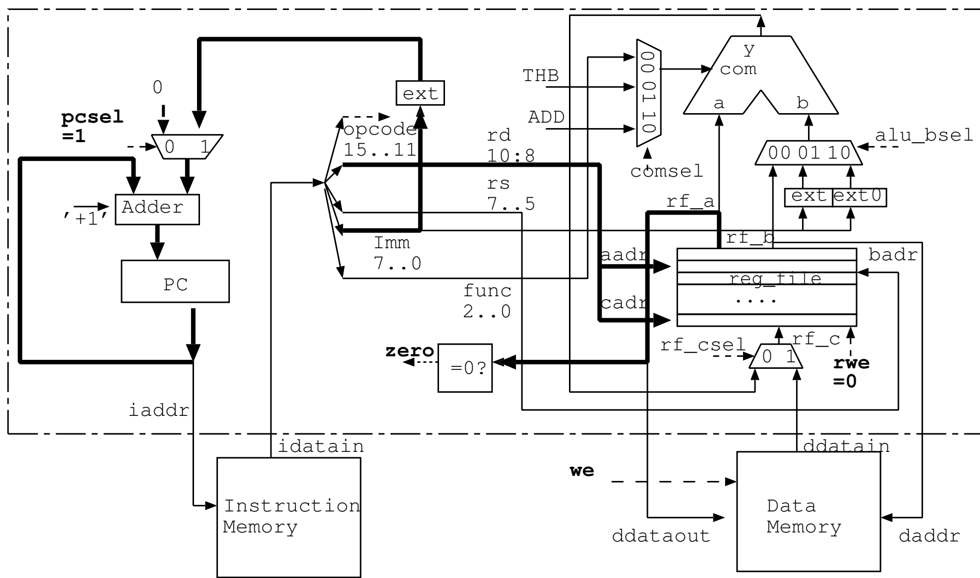

図 7: BEZ 命令実行のデータパス

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組み合わせ回路である。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。

例題 3: LDI、LDIU、ADDIU を表に付け加えよ。

表 1: 各命令の制御信号

	pcsel	comsel	alu_bsel	rf_csel	rwe	we
ADDI	0	10	01	0	1	0
SUB	0	00	00	0	1	0
LD	0	-	-	1	1	0
ST	0	-	-	-	0	1
BNZ	1	-	-	-	0	0

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] 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
```

この書き方はレジスタを個別に宣言していて、あまり賢いやり方ではない。

```
reg [`DATA_W-1:0] r[0:7];
```

とメモリの形で宣言すれば、もっとずっと楽である。なぜわざわざ個別に宣言したかというと、メモリの形で宣言すると gtkwave でシミュレーションをする際に中身を見ることができなくなってしまうためである。通常、メモリは膨大なので、波形シミュレータでその中身をいちいち表示することはしない。そうは言っても演習時にレジスタが見えないと困るので、今回はこのようにした。あくまで演習用と考えて欲しい。

ここで always 文の中で if 文と case 文を使っている。we が 1 である時に、書き込みを行う cadr の値に応じて入力 c がそれぞれのレジスタに書き込まれる。もちろん書き込みが行われるのはクロックの立ち上がりに同期している。case 文は便利だが、always 文と function 文の中にしか書けないため、読み出しの部分では、? : のマルチプレクサ構文を使っている。

さて、これを使って、図 2 を比較的そのまま Verilog で記述したものを示す。

```
`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;
wire [`SEL_W-1:0] com;
wire [`IMM_W-1:0] imm;
wire 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} = idatain;
assign imm = idatain[`IMM_W-1:0];
```

```
// Decorder
assign st_op = (opcode == `OP_REG) & (func == `F_ST);
assign ld_op = (opcode == `OP_REG) & (func == `F_LD);
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 we = 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 ;

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 ((bez_op & rf_a == 16'b0 ) | (bnz_op & rf_a != 16'b0))
     pc <= pc +{{8{imm[7]}},imm}+1 ;
   else
     pc <= pc+1;
end

endmodule
```

アキュムレータマシンと異なり、メモリは CPU と分離されている。今回は命令、データそれぞれ 64K × 16 ビットのメモリを使うのだが、この程度のメモリでも、チップに搭載する場合は、専用の IP(Intellectual Property) を用い、FPGA に搭載する場合は Block RAM と呼ばれる組み込みメモリを用いる。このため、CPU とは分離して設計する必要があり、CPU は，命令メモリに対するアドレス iaddr, データ入力 idatain、データメモリに対するアドレス daddr, データ入力 ddatain, データ出力 ddataout、書き込み制御信号 we を持つ。ddataout はレジスタファイルの a

出力が、daddr には b 出力が接続され、iaddr には pc が接続される。これらはもう決っているので最初に宣言する。

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

さて、idatain から読んできた命令は、以下のように分離する。

```
assign {opcode, rd, rs, func} = idatain;
assign imm = idatain[‘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 (解読器) と呼ぶ。デコードの結果、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 = (opcode == ‘OP_REG) & (func == ‘F_ST);
assign ld_op = (opcode == ‘OP_REG) & (func == ‘F_LD);
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 we = 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 ;
```

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 pc の記述である。

```
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))
     pc <= pc +{{8{imm[7]}},imm}+1 ;
   else
     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。ここでは+が二ヶ所に表われるが、条件が排他的なので、論理合成時には一つの加算器が割り当てられ、図 2 に相当する回路が生成される。

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 7-1

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 とし、def.h も書き換えよ。これで、web 中のテストプログラムが使える。

演習 7-2

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。ただし、opcode を 10100 とせよ。def.h も書き換えよ。

今回の範囲の POCO の基本命令

BMI と JMP は演習で付けるものなので注意！

```
NOP                                         00000 --- --- 00000
MV  rd,rs    rd <- rs                       00000 ddd sss 00001
AND rd,rs    rd <- rd AND rs                00000 ddd sss 00010
OR  rd,rs    rd <- rd OR rs                 00000 ddd sss 00011
SL  rd       rd <- rd<<1                    00000 ddd --- 00100
SR  rd       rd <- rd>>1                    00000 ddd --- 00101
ADD rd,rs    rd <- rd + rs                  00000 ddd sss 00110
SUB rd,rs    rd <- rd - rs                  00000 ddd sss 00111
ST rs, (ra)     rs -> (ra)                  00000 sss aaa 01000
LD rd, (ra)     rd <- (ra)                  00000 ddd aaa 01001
LDI rd,#X       rd <- X (符号拡張)          01000 ddd XXXXXXXX
LDIU rd,#X      rd <- X (符号拡張なし)      01001 ddd XXXXXXXX
ADDI rd,#X      rd <- rd + X (符号拡張)     01100 ddd XXXXXXXX
ADDIU rd,#X     rd <- rd + X (符号拡張なし) 01101 ddd XXXXXXXX
LDHI rd,#X      rd <- X|0                   01010 ddd XXXXXXXX
BEZ rd, X   if (rd==0) pc <- pc + X+1       10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X+1       10001 ddd XXXXXXXX
```

```
BMI rd, X   if (rd< 0) pc <- pc + X+1        10011 ddd XXXXXXXX
JMP  X                 pc <- pc + X+1        10100 XXXXXXXXXXX
```