第８回　分岐の拡張とサブルーティンコール

大小比較

前回、演習問題で検討したように BNZ, BEZ だけでは数の大小の比較ができない。そこで、POCO では、以下の命令を設ける。

```
BPL rd, X   if (rd>=0) pc <- pc + X          10010 ddd XXXXXXXX
BMI rd, X   if (rd<0) pc <- pc + X           10011 ddd XXXXXXXX
```

これらの命令はレジスタの内容の正負で分岐するかどうかが決る。これらの命令は、レジスタの符号ビットだけを見れば良いので、実装が楽だという利点がある。一方で、大小比較を行うには、引き算を行った後に、これらの命令を使う必要が出るため、レジスタの内容を破壊してしまう欠点がある。

以下に、0 番地から並んでいる 8 つのデータの中から最大のものを選ぶプログラムを示す。r3 に最大値が格納されるが、一度引いて比較した値を再び足し戻す処理が必要となる。

```
01001_000_00000000          // LDIU r0, #0x00
01001_011_00000000          // LDIU r3, #0x00
01001_100_00001000          // LDIU r4, #8
00000_010_000_01001         // LD r2,(r0)
00000_010_011_00111         // SUB r2, r3
10011_010_00000001          // BMI r2, +1
00000_011_010_00110         // ADD r3, r2
01100_000_00000001          // ADDI r0, #1
01100_100_11111111          // ADDI r4, #-1
10001_100_11111001          // BNZ r4, #-7
10000_100_11111111          // BEZ r4, #-1
```

うんちく：フラグを使う方法

POCO では採用しないが、これらの欠点を嫌ってフラグ (Flag) を使う方法を採用するプロセッサもある。この場合、フラグとは演算の結果の性質を格納する小規模の専用レジスタを指す。良く使われるフラグは、以下の例がある。

- Zero Flag: 演算の結果が 0 になるとセットされる
- Carry Flag: 演算の結果、桁あふれがおきるとセットされる
- Sign Flag: 演算の結果、マイナスになるとセットされる

分岐命令は、このフラグがセットされていれば分岐するという形式を取る。例えば、Branch Zero ならば、Zero Flag がセットされていれば分岐する、という命令に相当する。このような分岐命令で、PICO の命令セットと同様のプログラムが書けるのは容易に理解できるだろう。

フラグは演算の結果セットやリセットされるが、フラグだけを変化させる専用の命令を設ける場合がある。これを比較命令 (Compare) と呼ぶ。例えば、Compare r0,r1 とは、r0 と r1 を比較し、等しければ Zero flag が、r0r1 が成立すれば Sign flag がセットされる。

比較命令を使えば、レジスタを破壊せずに、フラグのみをセットして条件分岐が可能である。このようにフラグを使う方法は、便利で効率が高いが、一面、命令コードの入れ替えが難しい、割り込み発生時にフラグを保存する必要がある等の問題がある。

このため、PICO ではフラグを持っていない。

ジャンプ命令

Branch つまり分岐命令は、判断を伴うものを呼び、単純にプログラムカウンタの値を変更する命令のことをジャンプ命令と呼ぶ。ジャンプ命令は分岐命令と同様、相対指定を行うが、レジスタ指定の必要がない分遠くに飛ぶことができる。

```
JMP X       pc <- pc + X                      10100 XXXXXXXXXXX
```

POCO の場合、飛び先の番地を指定するビットを 11 ビット確保できる。しかし、この方法でも 11 ビットの範囲に入らなければ、飛べないことになり、ときに不便である。そこで、以下の最終手段を用意しておく。

```
JR  rd                pc <- rd                00000 ddd --- 01010
```

この命令は、レジスタ間接ジャンプと呼び、レジスタの値をそのままプログラムカウンタにセットする。一種の絶対ジャンプだが、レジスタの値はプログラムにより制御できるため、結果によってジャンプ先を変えたり、後に述べるようにサブルーチンコールからのリターン命令として利用可能である。

サブルーチンコール

JAL と JALR

ジャンプ命令は一度飛んでいったら、戻って来ない。しかし、プログラムは一定の処理をサブルーチンとして定義し、これを様々な場所から呼び出して利用する命令が必要である。これをサブルーチンコールと呼ぶ。サブルーチンコールは、図 1 に示すように、呼び出して、戻ってくる機能が必要になり、このため、プログラムカウンタをどこかに保存しておく必要がある。保存場所としては特定のレジスタを利用するのが一般的である。多くの場合、最大番号のレジスタを用いるので、POCO では r7 を保存場所として定める。

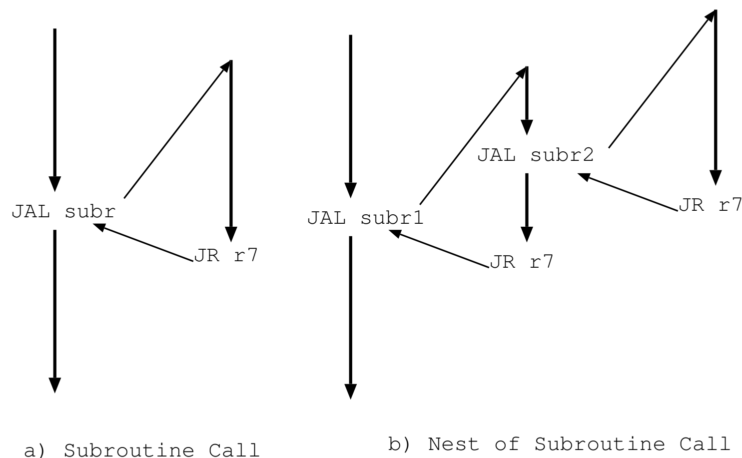


図 1: サブルーチンコール

```
JAL X       r7 <- pc, pc<- pc + X             10101 XXXXXXXXXXX
JALR rd     r7 <- pc,  pc <- rd                00000 ddd --- 01011
```

飛び先は JMP 命令と同様に 11 ビットを指定することができる。11 ビットではアドレス空間全体をカバーできないため、JR と同様にレジスタ間接指定を使えるようにしている。これが JALR である。この命令では飛び先をレジスタの内容で指定することができる。PC の格納場所としては、同様に r7 である。

戻り番地はr7に格納されるため、リターン命令はJR r7となる。下のプログラムは0番地のデータの2乗を計算するものである。掛け算のサブルーチンMultをコールしている。このサブルーチンはr1 X r2の結果がr3に格納される。r1は破壊されないが、r2は破壊されてしまう点に注意。なお、ここではプログラムを見やすくするためにラベルを使っている。通常、ラベルはアセンブラにより、自動的に相対アドレスに変換される。

```
01001_000_00000000          //        LDIU r0, #0x00
00000_001_000_01001          //       LD r1, (r0)
00000_010_001_00001          //       MV r2, r1
10101_00000000010          //         JAL mult
00000_011_000_01000          //       ST r3, (r0)
10100_11111111111          //   end:  JMP end
//Subroutine Mult
01001_011_00000000          //  mult: LDIU r3, #0x00
00000_011_001_00110          // loop: ADD r3, r1
01100_010_11111111          //        ADDI r2, #-1
10001_010_11111101          //        BNZ r2, loop
00000_111_000_01010          //       JR r7
```

スタック

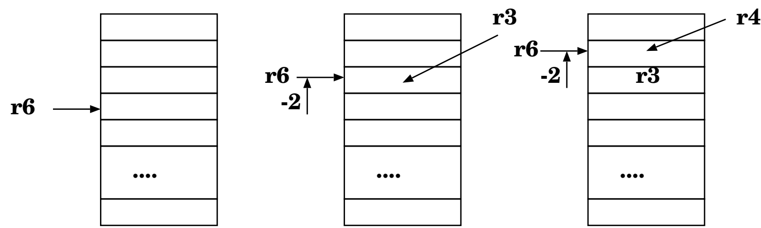
サブルーチンは、プログラムの各所から呼び出せるという利点がある。しかし、ここで問題がある。呼び出すプログラム（メインルーチン）とサブルーチンで利用するレジスタが重なってしまうと、サブルーチンを呼び出してリターンしてきた時に、サブルーチン内でレジスタが破壊されて、プログラム実行上の情報が失われる可能性がある。そこで、サブルーチンは、呼び出された時に、利用するレジスタをどこかにしまって置く必要がある。これを実現するのがスタック(Stack)と呼ぶ。スタックは棚であり、先に積んだものを後で取り出し、後で積んだものを先で取り出すことからLast In First Out (LIFO)またはFirst In Last Out (FILO)とも呼ぶ。スタックにデータを積む操作をpush、取り出す操作をpopと呼ぶ。RISCでは通常スタックは、ソフトウェアで実現する。このためには、スタックとして利用するメモリ領域を指すレジスタが必要になる。これをスタックポインタと呼ぶ。図2の例ではr6がスタックポインタであり、プログラムをスタートする場合これをスタックとして使う領域の最後に設定しておく必要がある。
r3,r4を順にスタックにpushする操作は以下のようになる。

```
ADDI r6,#-1
ST r3,(r6)
ADDI r6,#-1
ST r4,(r6)
```

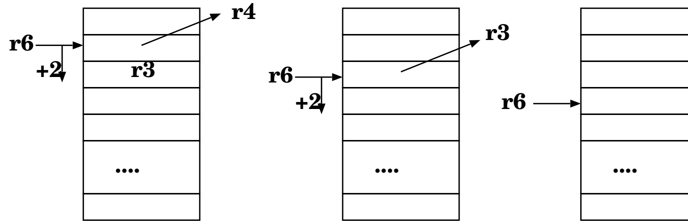
すなわち、あらかじめスタックポインタを減らして領域を確保し、そこにデータを格納する。逆に格納したデータを元に戻すpop操作は以下のようになる。

```
LD r4,(r6)
ADDI r6,#1
LD r3,(r6)
ADDI r6,#1
```

スタック操作の優れた点は、サブルーチンの入れ子（ネスト）に対応可能な点である。サブルーチンの中でサブルーチンを呼んでも、呼ばれたサブルーチンで利用するレジスタが順にスタックに積まれ、逆順で戻るため、スタック領域を使いつくすまでいくらでもサブルーチンを呼ぶことが可能である。実際、再帰呼び出し(recursive call)を行う場



a) Stack push



b) Stack pop

図 2: スタックプッシュとポップ

合、多くのデータがスタックに積まれることになる。ここで、戻り番地は r7 に格納されていることから、忘れずに r7 をスタックに格納し、リターン操作である JR r7 を実行する前に元に戻しておくことが必要である。

JMP, JR, JAL の付加

JMP 命令の付加は前回の演習でやったと思うが、pc の前段に簡単な切り替え回路を付ければ良い。この様子を図 3 に示す。JR は pc に直接 rfile から読み出した rf_a を入れられるようにする。これは加算が必要ないので、加算後に別にマルチプレクサを設けるのが楽である。これも図 3 中に示す。

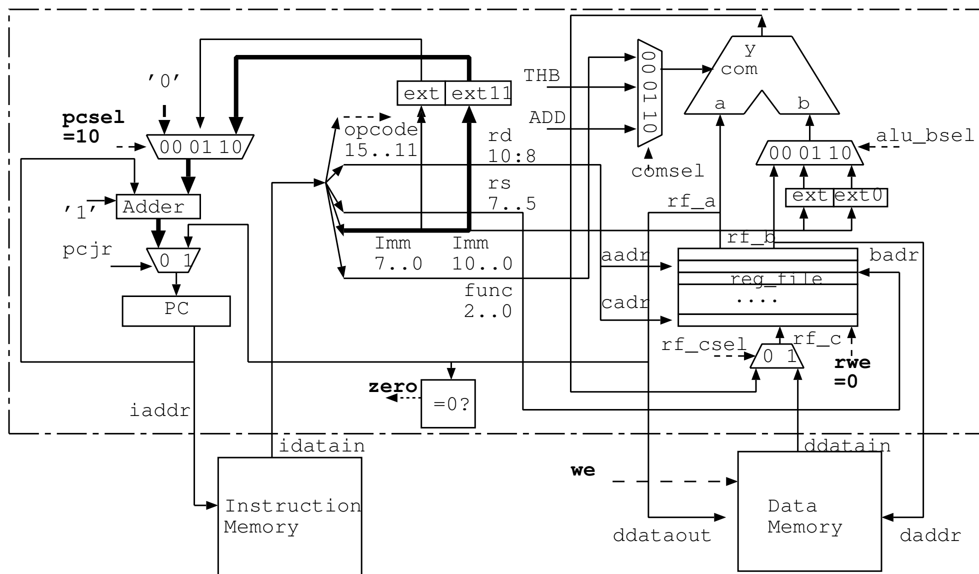


図 3: JMP/JR が実行可能なデータパス

一方、JAL 命令の実現には、r7 に pc を書き込むために、レジスタファイル周辺に付加回路が必要となる。この様子を図 4 に示す。まず、書き込むレジスタである 7 を指定するために、レジスタの c ポートのアドレスにマルチプレクサを設置する。また、pc をレジスタファイルに書き込むために c ポートの入力 rf_c に対するマルチプレクサを拡張

する。さらに、pc に 1 を加えてから格納する必要があるため、加算器がもう一つ必要になる。JAL の戻り番地のためだけに加算器を設けるのはやや無駄だが、これは後にマルチサイクル化する際に減らすことにする。

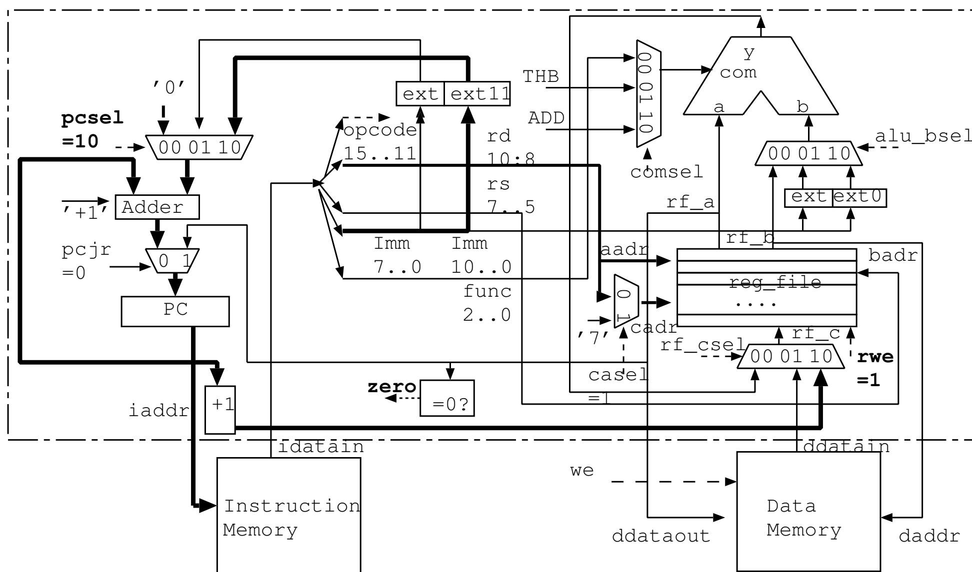


図 4: JAL が実行可能なデータパス

これで、POCO のデータパスの拡張はひとまず終わりである。見ての通りかなりごちゃごちゃしてしまった。これを整理する方法は次回検討することにしよう。では、各マルチプレクサの制御を表にまとめてみよう。今回はほとんど拡張した部分が中心である。

表 1: 各命令の制御信号

	pcsel	pcjr	comsel	alu_bsel	casel	rf_csel	rwe	we
BNZ	01/00	0	-	-	-	-	0	1
JMP	10	0	-	-	-	-	0	1
JR	-	1	-	-	-	-	0	1
JAL	10	0	-	-	1	10	1	1

例題

上の表に JALR を付け加えよ。

Verilog の変更

図 4 に示すハードウェアの変更はほとんどそのまま Verilog 記述に反映することができる。ここでは変更箇所のみ示す。まず wire 文で、各命令に対応する信号を宣言してから、デコーダに以下の部分を付け加える。これで、それぞれの命令に対応した信号が利用可能となる。

```
assign jr_op = (opcode == ‘OP_REG) & (func == ‘F_JR);
assign bpl_op = (opcode == ‘OP_BPL);
assign bmi_op = (opcode == ‘OP_BMI);
assign jmp_op = (opcode == ‘OP_JMP);
assign jal_op = (opcode == ‘OP_JAL);
```

まず、JAL 命令用にレジスタファイルの c ポートの入力とアドレスの拡張が必要となる。そこで、以下のようにする。

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

すなわち、JAL 命令の際は、c ポートのアドレス cadr が 7 となり、入力には pc+1 が設定される。次に、pc については以下のように修正する。

```
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) |
                (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
```

ここで、JMP と JAL では ir の下位 11 ビットを符号拡張している点に注意されたい。また、JR では直接レジスタファイルの a ポートから読み出されたレジスタが設定されている。always 文中であるため、if 文が使えるため、条件は見やすいと思う。

POCO の基本命令

今回で、POCO の命令上の拡張は打ち止めである。最後にまとめておく。

```
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          10000 ddd XXXXXXXX
BNZ rd, X   if (rd!=0) pc <- pc + X          10001 ddd XXXXXXXX
BPL rd, X   if (rd>=0) pc <- pc + X          10010 ddd XXXXXXXX
BMI rd, X   if (rd<0) pc <- pc + X           10011 ddd XXXXXXXX
JMP X       pc <- PC + X                     10100 XXXXXXXXXXX
JAL X       r7 <- pc, pc<- pc + X            10101 XXXXXXXXXXX
JR  rd                pc <- rd               00000 ddd --- 01010
JALR rd     r7 <- pc,  c <- rd               00000 ddd --- 11000
```

本日の課題

その 1

例題の乗算ルーティンを使って、0 番地に格納されているデータの階乗を計算するプログラムを書け。(まずは値を小さくしてテストすることをお勧めする。アセンブラ shapa を利用せよ。)

その 2

JALR を poco1.v に付け加え、その機能をテストせよ。