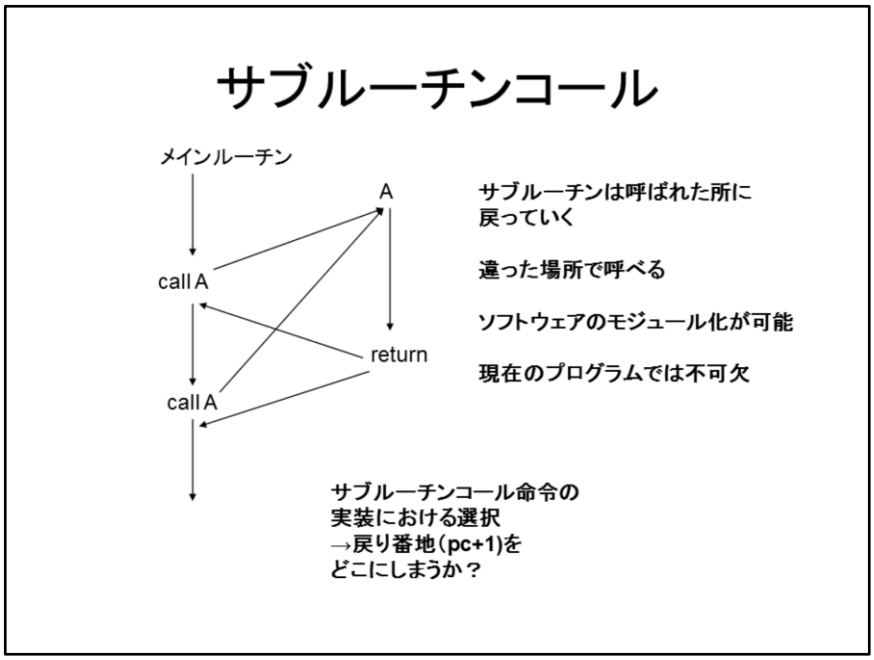
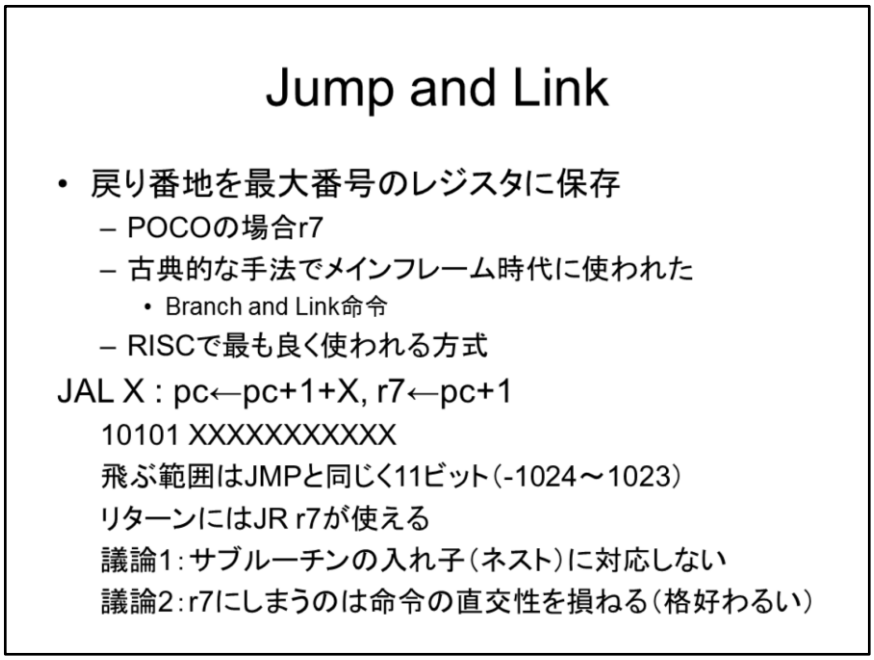
計算機構成　第8回
サブルーチンコールとスタック
テキストp85-90

情報工学科
天野英晴



分岐命令、ジャンプ命令と違ってサブルーチンコール命令は、呼ばれた所（次の命令）に戻ってくる点が特徴です。図の例ではAを呼び出して、リターン命令実行時にコール命令の次に戻ります。Aは色々なところで使えるため、ソフトウェアのモジュール化が可能です。この考え方は現在のプログラムでは不可欠です。問題は、戻り番地（すなわちコール命令実行時のPC+1）をどこにしまっておくか？という点です。



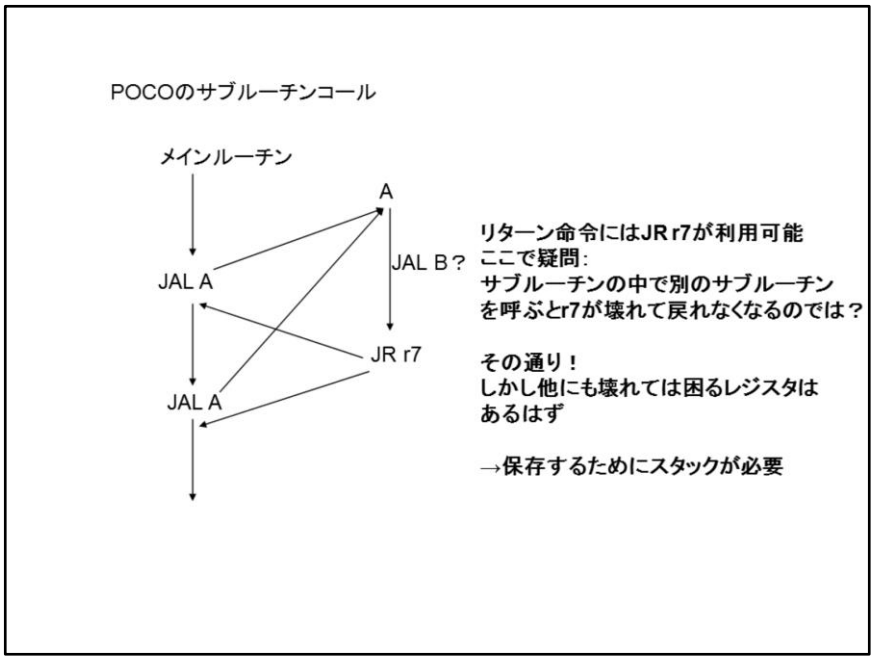
古典的な方法は、戻り番地を最大番号の汎用レジスタに保存しておく方法です。これをJump and Link（Branch and Link)と呼びます。元々メインフレーム時代に開発された由緒ただしい方法で、RISCでもよく使われます。POCOではr7に格納します。飛ぶ範囲は広い方が良いので、JMP命令と同じくJ型とし、オプコード以外は全て飛び先に使い、PC+1を起点として−1024から1023の範囲で飛んでいきます。この方法では戻り番地（PC＋１）はr7に格納されているため、リターン命令はJR　r7となります。さて、この方法には二つ議論すべき点があります。一つは、サブルーチンの中でサブルーチンを呼ぶ（サブルーチンの入れ子と呼びます）と戻り番地のr7が破壊されてしまう点です。もう一つはr7にしまうことにより、r7が他の汎用レジスタと違った役目を持つことになり、命令の直交性（Orthogonality)を悪くしてしまう点です。この点については後で検討します。

2乗を計算する例

```
        LDIU r0,#0
        LD r1,(r0)
        MV r2,r1     r1とr2に同じ値をセットする
        JAL mult
end:    JMP end

// Subroutine Mult  r3 ←  r1 × r2 ここでr2は破壊される
mult:   LDIU r3,#0
loop: ADD r3,r1
        ADDI r2,#-1
        BNZ r2, loop
        JR r7            ←  r7には戻り番地が入っている
```

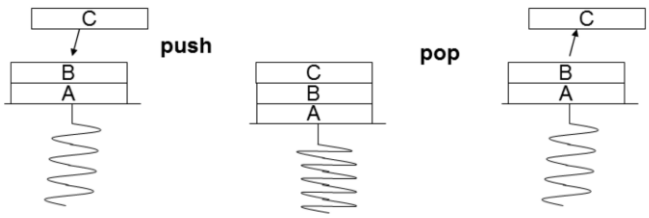
では、2乗を計算する例を紹介しましょう。この例ではサブルーチンとしてmultを定義します。このサブルーチンは、r1の値とr2の値を掛け算して、答えをr3に返します。ここではr2は破壊されてしまいます。まず、メインルーチンではr0を0にセットして、0番地の内容をr1にロードします。r1の内容をr2にコピーしてJAL multを実行するとサブルーチンが実行され、0番地の内容の自乗が計算され、答えがr3に返ります。



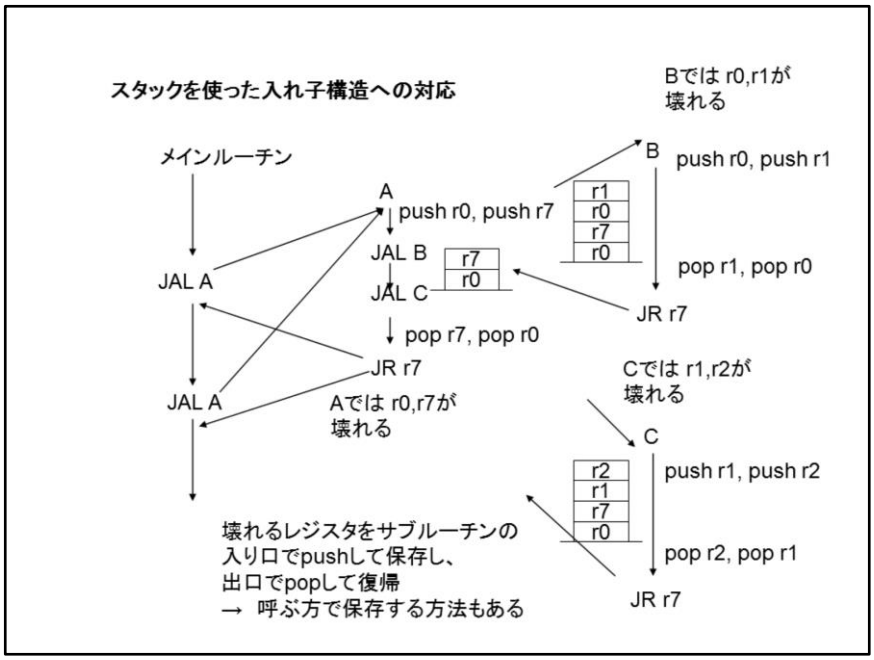
このJALというやり方は、サブルーチンの入れ子に対応しません。サブルーチンAの中で別のサブルーチンBを呼ぶと、r7の内容は破壊されてしまうため、サブルーチンAの最後にJR r7を実行しても、呼ばれた元に戻ることができません。これでは困るではないか、と思うかもしれませんが、実はサブルーチンを呼んだ時のレジスタの破壊は、r7以外にも問題になります。メインルーチンとサブルーチンA,サブルーチンAとサブルーチンBで同じレジスタを使うと、サブルーチンから戻ってきたときに中身が破壊されて、実行が継続できなくなってしまいます。すなわち、r7だけではなく、サブルーチン内でのレジスタの破壊はサブルーチンコール自体の本質的問題なのです。これを解決するにはスタックというデータ構造が必要になります。

スタック

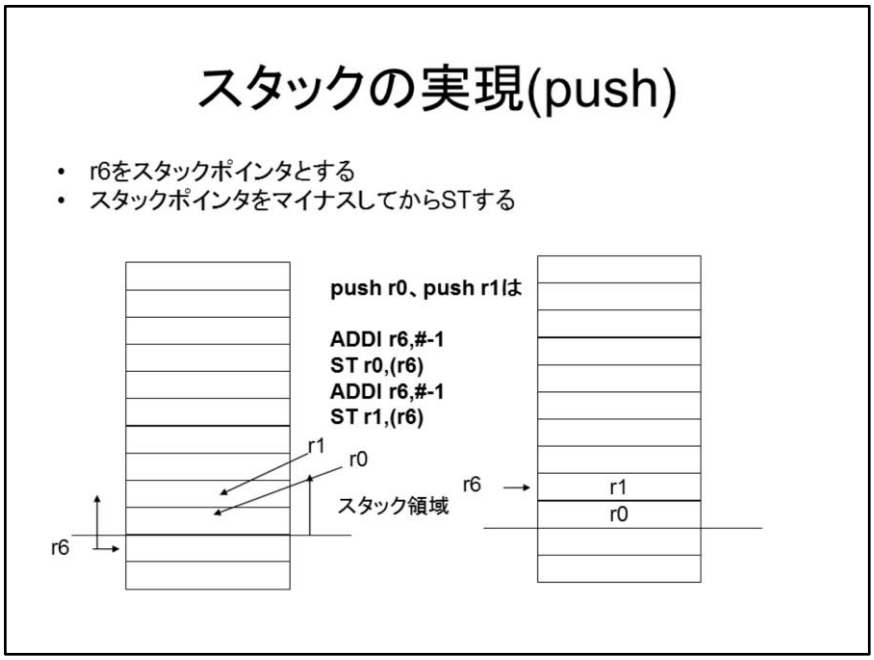
データを積む棚
　push操作でデータを積み
　pop操作で取り出す
- LIFO(Last In First Out)、FILO(First In Last Out)とも呼ばれる
- 演算スタックとは違う（誤解しないで！）
- 主記憶上にスタック領域が確保される



スタックとは、データを積む棚です。この棚にデータを積む操作をpush、棚から取り出す操作をpopと呼びます。先に積んだものが後から取り出されることからLIFO (Last In First Out)と呼びます。逆に考えると、後に積んだものが先に取り出されるのでFILO（First In Last Out)と呼ぶ場合もあります。この積んだ逆順に取り出すことのできる性質からサブルーチンコール時にレジスタを退避するのに適しています。
以前紹介したスタックマシンで利用した演算スタックは、演算用の特殊なメモリですが、サブルーチンコールのレジスタの退避用のスタックは主記憶上に確保するのが普通です。スタックは棚ですが、ばねがついているイメージがあります。データを積むときは押し込むイメージからpushと呼び、取り出すときは、飛び出すイメージからpopと呼ばれます。



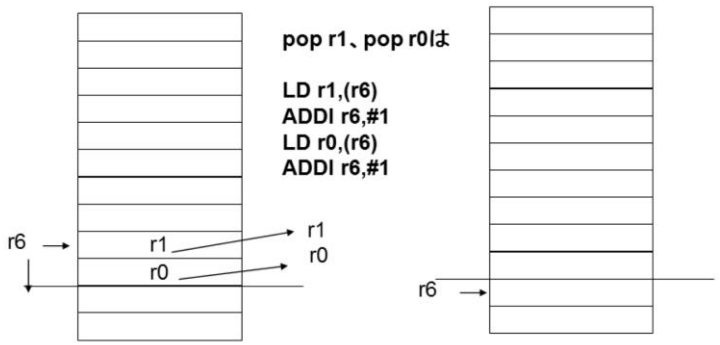
スタックを使ってレジスタを退避する様子を示します。この例では、サブルーチンの入り口で、中で使って壊れるレジスタを退避し、リターンする直前に復帰する方法を示します。これはコーリーサーブと呼びます。逆に呼ぶ側で、壊れて困るレジスタをスタックに積んでからサブルーチンを呼び出す方法（コーラーセーブ）もあります。サブルーチンAではr0を使います。中で別のサブルーチンを呼ぶのでr7も退避します。サブルーチンBを呼んだ際にr0、r1を退避します。この2つのレジスタはサブルーチンAを呼んだ際のr0、r7の上に積まれます。サブルーチンBの中でさらに別のサブルーチンを呼ぶ場合、さらにこの上に積み重なります。サブルーチンからリターンする直前に、pushしたのと逆順にpopします。そのようにすると、スタックの内容は呼ばれた時と同じになります。さらに別のサブルーチンCを呼んだ場合、サブルーチンの入れ子になった場合も同様に対処できます。再帰呼び出し（リカーシブコール）を行った場合も、スタックの容量が許す限り、スタックにレジスタを積み続けることができます。（再帰呼び出しのプログラムにバグがあるとセグメンテーションフォルトになるのは、スタックが溢れてしまうためです）



スタックをメモリ上に実現するには、スタックポインタを使います。ここではr6をスタックポインタの役割に使います。スタックポインタは、スタック領域の一番上の番地＋１の所に初期化します。push操作は、まずスタックポインタを減らし、空いた領域にレジスタを書き込みます。スタック領域はメモリ上の番地が減る方向に伸びていきます。この図はpush r0, push r1を順に実行した様子を示します。

スタックの実現(pop)

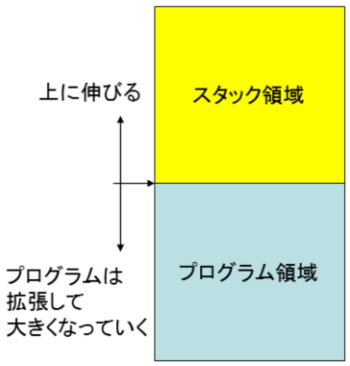
- スタック領域はメモリの番地の小さい方に伸びる→昔からの習慣
- pop操作は、LDしてからスタックポインタを＋する。



逆にpop操作はまずスタックポインタの指し示す番地から取り出して、スタックポインタを増やします。図はpop r1, pop r0を順に実行した様子を示します。スタックポインタは最初の位置に戻ります。

スタックが上に伸びる理由

- 昔からの習慣
- プログラムがアドレスの上に向かって伸びるのとぶつからないようにするため



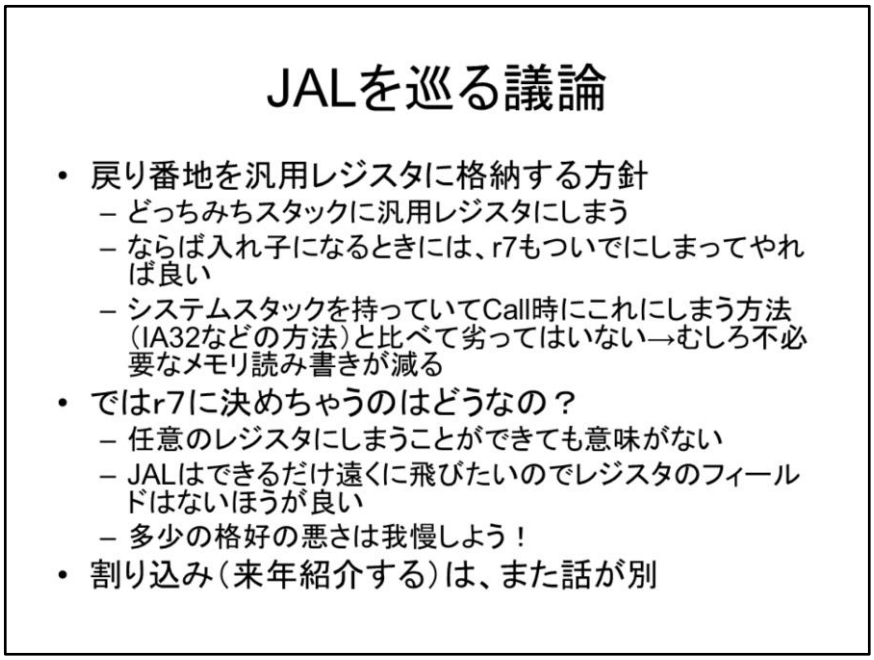
スタックは番地の小さくなる方向に（この図では上に向かって）伸びるのが普通ですが、これは昔からの習慣に基づいています。昔はある場所に境界を引いて、それより小さい番地はスタック領域とし、大きな番地はプログラム領域としました。これはプログラムというのは開発を進めると大きくなるので、番地が増える方向に伸びます。スタックを番地の増える方向に成長させると方向が同じになり、成長したスタックがプログラム領域を食い荒らす可能性があります。そこで、スタックは番地が小さい方向に伸ばす習慣ができました。もちろん領域を食いつぶしてしまえば、どちらの方向でもトラブルになります。

掛け算用サブルーチン

r2を破壊しないサブルーチンコール
r6はメインルーチンで初期化する必要がある

```
// Subroutine Mult  r3 ← r1 × r2
mult:   ADDI r6,#-1  この2行でr2をスタックにpush
        ST r2,(r6)     r2の値がスタックに退避される
        LDIU r3,#0
loop: ADD r3,r1
        ADDI r2,#-1
        BNZ r2, loop
        LD r2,(r6)    この2行でr2をスタックからpop
        ADDI r6,#1  r2の値がスタックから復帰する
        JR r7          それからリターン
```

ではどのレジスタを保存すればよいでしょうか？もちろん答えを返すレジスタであるr3は保存しません。r2は入力の値を渡すレジスタですが、これをスタックに退避することで、r1同様、メインルーチンで値を使うことができます。



JALは戻り番地を汎用レジスタr7にしまうため、サブルーチンが入れ子になると壊れてしまいます。しかい、どっちみち他の汎用レジスタだってメインルーチンとサブルーチンで両方使う場合は、壊れるのでスタックにしまう必要があります。サブルーチンの入れ子になる場合はこれと一緒にr7もしまっていまえばいい、という考え方です。これはリーゾナブルだと思います。Intelのx８６（IA32)などでは、システムスタックというシステムで管理するスタックを持っていてサブルーチンコール時に戻り番地をこれに自動的にpushする方法を取ります。これはサブルーチンの入れ子に対応可能ですが、入れ子でない場合も強制的にスタックに積んでしまうので、無駄なメモリアクセスが増えると言えます。
次にr7に決めてしまっていますが、これはどうでしょう？JALは出来る限り遠くに飛びたいです。これは、サブルーチンは、ライブラリとして、ユーザープログラムとは別の番地に置かれる場合が多いためです。戻り番地をしまうレジスタはどっちみちどこかに決めてしまいます。ならば、これをr7に決めてしまい、残りの全てのビットを飛び先を決めるアドレスに充てた方が良いです。このことにより命令の直交性が低下します。直交性とは、ある操作をレジスタ番号や命令の種類に関係なしに施すことができるかどうかを示す性質です。直交性の高い命令は美しい命令になります。r7のみ戻り番地をしまえることにより直交性は低下しますが、この場合実害はないので、多くのRISCはこの方法を使っています。

JAL命令のVerilog記述

- まず例によってJAL命令をデコードする

```
assign jal_op = (opcode == `JAL_OP);
```

- pcをr7に保存する部分

```
assign rf_c = ld_op ? ddatin: jal_op ? pc+1: alu_y;   //
        JAL命令ならばPC+1をレジスタファイルの入力に与える
assign rwe = ld_op | alu_op…| jal_op;        //
        レジスタファイルに書き込むためにrweを1にする
```

では、JAL命令を付け加えてみましょう。JAL命令は他の命令とやることがかなり違うので、改造箇所が多いです。まず他の命令と同じく命令デコード信号を作ります。これがjal_opです。いうまでもなく、def.hにJAL_OPを定義（10101)しておく必要があります。次にpcをr7に保存しなければいけないので、レジスタファイルの入力rf_cを変更する必要があります。jal_opの時はpc+1を入れるようにし、rweも1になるようにします。

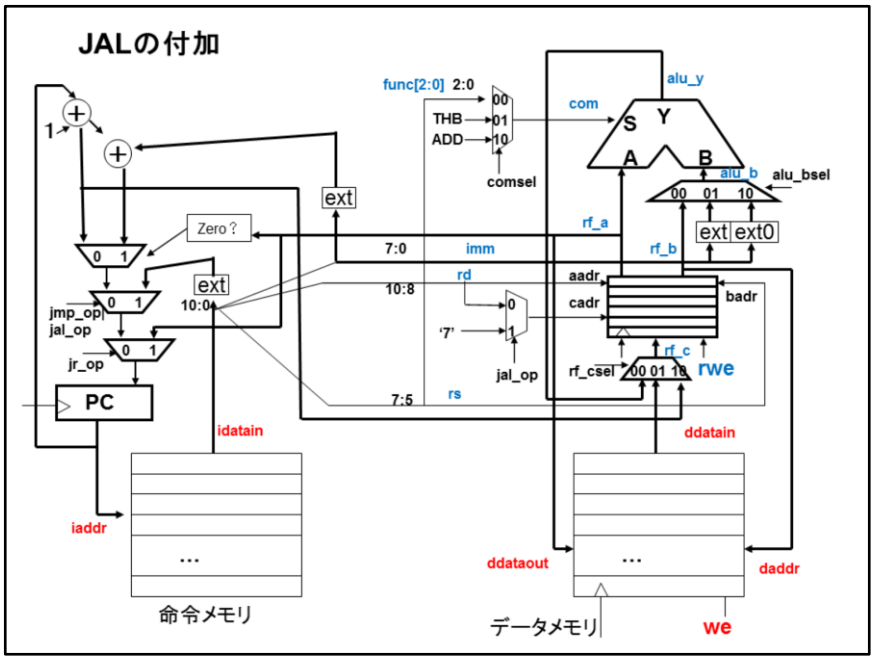
r7へのpc+1の書きこみ

今まで、レジスタファイルのAポートとCポートは同じくディスティネーションレジスタの番号(rd)を入れていた。しかしJAL命令では番号7を強制的に入れる必要がある。そこで、新しくcadrという信号名を設ける。

```
wire [`REG_W-1:0] cadr;
assign cadr = jal_op? 3'b111 : rd; //
        JALのときだけ7それ以外は今まで通りrd

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

次にレジスタファイルのCポートのアドレスについて変更が必要です。これまではレジスタファイルのAポートとCポートは同じディスティネーションレジスタ番号rdを入れていました。これはPOCOの演算のrdがソースとディスティネーションを兼ねるためです。ところがJAL命令に関してはr7にpc+1を書き込むという特殊な操作が必要になります。そこで、Cポートのアドレスcadrという信号名を定義し、jal_opのときのみr7を示すために‘7’を入れてやります。それ以外の命令では今まで通りrdを入れます。このcadrをレジスタファイルのCポートアドレスに接続します。



この変更によって、POCOの構造は図のように変わります。Cアドレスのところに新しいマルチプレクサが必要です。レジスタファイルのCポート入力のマルチプレクサも拡張しPC+1を引っ張ってきて入れてやります。

PC周辺

飛び方はJMPと同じ

```
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+{{8imm[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
```

PC周辺も若干変更します。JALはJMPと同じくJ型の命令なので、オプコード以外の11ビットを全て飛び先指定に使います。したがってJMP命令と同じ符号拡張の記述を使います。

R型命令一覧

NOP		00000------00000
MV rd,rs	rd← rs	00000dddsss00001
AND rd,rs	rd← rd AND rs	00000dddsss00010
OR rd,rs	rd← rd OR rs	00000dddsss00011
SL rd	rd← rd<<1	00000ddd---00100
SR rd	rd← rd>>1	00000ddd---00101
ADD rd,rs	rd← rd + rs	00000dddsss00110
SUB rd,rs	rd← rd - rs	00000dddsss00111
ST rd,(ra)	(ra)← rd	00000dddaaa01000
LD rd,(ra)	rd← (ra)	00000dddaaa01001
JR rd	pc ← rd	00000ddd---01010

今までに出てきたR型の命令です。前回と同じです。

I型命令一覧

LDI rd,#X	rd← X(符号拡張)	01000dddXXXXXXXX
LDIU rd,rs	rd← X(ゼロ拡張)	01001dddXXXXXXXX
ADDI rd,#X	rd←rd+X(符号拡張)	01100dddXXXXXXXX
ADDIU rd,#X	rd←rd+X(ゼロ拡張)	01101dddXXXXXXXX
LDHI rd,#X	rd←{X,0}	01010dddXXXXXXXX
BEZ rd,X	if(rd=0) pc←pc+X+1	10000dddXXXXXXXX
BNZ rd,X	if(rd≠0) pc←pc+X+1	10001dddXXXXXXXX
BPL rd,X	if(rd>=0) pc←pc+X+1	10010dddXXXXXXXX
BMI rd,X	if(rd<0) pc←pc+X+1	10011dddXXXXXXXX

今まで出てきたI型の命令です。これも前回と同じです。

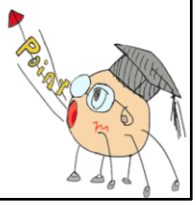
J型命令一覧

JMP #X	pc←pc+X+1	10100XXXXXXXXXX
JAL #X	pc←pc+X+1, r7←pc+1	10101XXXXXXXXXX

今回新しくJAL命令が加わっています。

本日のまとめ

- サブルーチンコール命令JAL 飛び先
  - 飛び方はJMPと同じで11ビットの範囲で飛べる
  - 戻り番地はr7に格納される
- リターン命令はJR r7
- 壊していけないレジスタはスタックに退避
  - r6スタックポインタとする
  - pushはADDI r6,#-1 , ST rx,(r6)
  - popはLD rx,(r6), ADDI r6,#1
  - サブルーチンの入れ子になる場合はr7を退避



インフォ丸が教えてくれる今日のまとめです。

演習

1.multを利用して0番地の数の3乗を計算せよ
提出物はsanjo.asm(jijo.asmまたはjijo2.asm
を利用のこと） 0番地が0x1bになれば正解

2. JALR rdは、JAL同様r7に戻り番地をしまって
JR同様rdの中身の番地にそのまま飛ぶ命令
である。この命令を実装せよ。

JALR rd 00000ddd---11000

./shapa jalrtst.asm –o imem.datでテスト可能

0番地が9になれば正解

提出物はpoco.v

演習１は、2乗の例を拡張し、multというサブルーチンを２回呼んでください。
演習２はJALR命令を付加する課題です。JALは１１ビットの飛び先フィールドを持っていますが、稀にこの範囲に入らないことがあります。この場合、どこでも飛べるように、レジスタ間接指定のサブルーチンコールを用意します。これがJALR命令です。この命令はJAL同様r7に戻り番地をしまいますが、飛び先はrdで示します。JR同様、絶対指定です。jalrtst.asmを使って２乗が計算できるか調べてください。