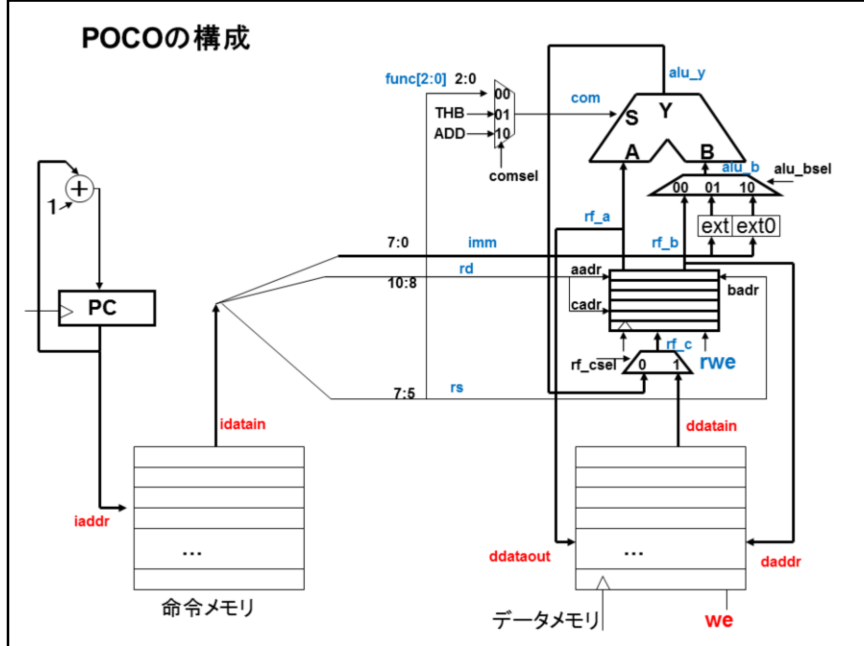
マイクロプロセッサ特論　第7回
分岐命令とプログラムの実行
テキスト第5章

情報工学科
天野英晴



前回、POCOのVerilog記述を紹介しました。ただし、この構成ではPCは毎クロック1増えるだけなので、プログラムは順番にしか進みません。アキュムレータマシン同様、分岐命令を付けてやりましょう。

POCOの条件分岐命令

BEZ rx,X： if(rx=0) PC←PC+1+X

10000 ddd XXXXXXXX

BNZ rx,X： if(rx≠0) PC←PC+1+X

10001 ddd XXXXXXXX

- PC相対指定
  - 命令の位置＋1を起点としてX分命令を飛び越す
  - 8ビットのフィールド→ -128から127まで飛べる
  - 局所性（Locality)があるので多くの場合大丈夫
  - プログラムが再配置可能（Relocatable)になる

アキュムレータマシンと同様、POCOにもレジスタの中身をチェックして0ならば飛ぶBEZ命令と0でなければ飛ぶBNZ命令を付けましょう。汎用レジスタマシンはレジスタが複数あるので、レジスタを指定して判定するようにします。もう一つ、重要な違いがあります。POCOの条件分岐命令はPC相対指定なのです。これは命令の位置＋1を起点としてX命令飛び越します。POCOは8ビット分フィールドがあるので、－128から127まで飛べます。このようにすると絶対指定で必要な16ビットをまるまる指定しなくて済みます。それでもプログラムには局所性があるので、多くの場合は問題が生じません。また、相対指定を使えば、プログラムが再配置可能（Relocatable)になります。すなわち、プログラムをどの番地から配置しても動作します。

PC相対指定

その前の命令
前の命令
BEZ r0, -3　　　10000_000_11111101
次の命令を起点にして3つ分飛ぶ
BEZ r0,-1は自己ループになる
– 16bitのアドレスが命令コードに入らないので8bitに縮めたのがPC相対という解釈もあるが、、
– 16bit全て指定するのは無駄！という考え方もある
– 事実上全てのプロセッサはPC相対指定を利用
– 遠くに飛ぶ命令を別に用意すれば良い

PC相対指定は、その分岐命令ではなく、その次の命令を起点とします。すなわち、BEZ r0,-3の場合は、次の命令を起点にして3つ分飛ぶので、前の前の命令に戻ることになります。すなわち、BEZ r0,-1は自分自身に飛ぶループになり、BEZ　r0、0は、成立してもしなくても次の命令を実行するので、意味のない命令になります。その分岐命令を起点としても命令セットアーキテクチャとしては特に不都合はないのですが、実装上便利なため、全てのコンピュータがこの方法を採用しています。
さて、POCOでは飛び先のアドレスは8ビットにしていますが、アドレス空間は16ビットです。これは16ビットに入らないから8ビットに縮めたと解釈もできますが、むしろ16ビット全部持つのは無駄という考え方もあります。局所性の原則により、近場に飛ぶことが多いので、全アドレス空間を持つのは確かに無駄です。例外的に飛べない場合は、遠くに飛べる命令を別に用意しておけばよいのです。このため、この方法は事実上全てのプロセッサで使われています。

PC相対指定による分岐
（掛け算のプログラムの例）

```
LDIU r0,#2
LD r1,(r0)        r1 ←  2番地の内容
LDIU r0,#3
LD r2,(r0)        r2 ←3番地の内容
LDIU r3,#0        r3は答が入るので0に初期化
ADD r3,r1         r3にr1を足しこむ
ADDI r2,#-1       r2から1を引く
BNZ r2, -3        0でなければループ
LDIU r0,#0
ST r3,(r0)
BEZ r2,-1
```

では掛け算のプログラムを実行してみましょう。この例題では、2番地の内容と3番地の内容を掛け算します。2番地の内容はr1に、3番地の内容はr2に入ります。答えを入れるr3は0にしておきます。あとはr3にr1に足しこんで行き、r2から1を引き、0でなければループします。ループは2命令前に戻るので、BNZ r2,-3を使います。ループを抜けたら答えを0番地に入れます。

アセンブラshapaを使おう mult.asmの例

```
       LDIU r0,#2
       LD r1,(r0)       r1 ← 2番地の内容
       LDIU r0,#3
       LD r2,(r0)       r2 ←3番地の内容
       LDIU r3,#0       r3は答が入るので0に初期化
loop:  ADD r3,r1        r3にr1を足しこむ
       ADDI r2,#-1      r2から1を引く
       BNZ r2, loop     0でなければループ
       LHIU r0,#0
       ST r3,(r0)
end:   BEZ r2,end
```

loop, endなどのラベルが利用可能→いくつ飛ばすか計算しなくて良い
自動的に機械語に変換してくれる

ここでは、アセンブラshapaを使ってみます。アセンブラはラベルを使うことができます。この場合loopというラベルを使います。

shapaは、rubyで書かれています。アセンブラ本体と命令の定義部が完全に分かれています。命令の定義部poco.rbを修正すれば様々な命令を付け加えることができます。./shapa mult.asm –o imem.datとやればimem.datに機械語が吐き出されます。shapaを使って掛け算のプログラムをアセンブルして実行してみましょう。

大小比較と分岐

BPL rd, X　if(rd>=0) pc ← pc+1+X

10010 ddd XXXXXXXX

– Branch Plus rdが0以上ならば飛ぶ

BMI rd,X if(rd<0) pc ←　pc+1+X

10011 ddd XXXXXXXX

– Branch Minus rdがマイナスならば飛ぶ

- 実装は簡単だが引き算をしてレジスタを破壊しないと分岐ができない

BEZ、BNZでは単純な0との比較なので、数の大小比較ができません。そこでPOCOではレジスタの値が0か正の数の場合、成立するBPL（Branch Plus)と、負の数の場合に成立するBMI（Branch Minus）を設けます。引き算をした結果に対してこの命令で分岐すれば大小判定が出来ます。この方法は符号ビットを見れば飛ぶかどうかの判定ができるので、判定が簡単かつ高速であるメリットがあります。しかし、引き算をしてレジスタを破壊しないと分岐できないという欠点があります。

最大値を選ぶプログラム例 max.asm

```
      LDIU r0,#0    ポインタは0
      LDIU r3,#0    r3は暫定チャンピオン
      LDIU r4,#8    調べる数は8つ
loop: LD r2,(r0)    r2は挑戦者
      SUB r2,r3     引いてみる
      BMI r2,skip   チャンピオンが勝てばスキップ
      ADD r3,r2     足し戻すことでr3とr2が交換される
skip: ADDI r0,#1    ポインタを進める
      ADDI r4,#-1   カウンタを減らす
      BNZ r4,loop   8個調べたらおしまい
end:  BEZ r4,end
```

大小比較の例として最大値を選ぶプログラム例を示します。この例では0番地から並んだ8個の数の最大値を選びます。r0はポインタ、r4はカウンタでそれぞれ0と8を入れておきます。r3を最大値すなわちチャンピオンが入るレジスタとします。最初はr3は0、すなわち最弱のチャンピオンを入れておきます。ループ内は以下のように動きます。r2にr0をポインタとして値を取って来ます。これが挑戦者です。挑戦者からチャンピオンを引き算します。マイナスになるとチャンピオンが勝ったので、防衛成功で次の命令をスキップして何もしないで、ポインタ進めてカウンタを減らして0になってなければループします。ここで引き算の結果が0か正ならば、挑戦者が勝ったことになります。この場合BMIは成立せず、次のADD r3,r2が実行されます。r2にはr2ーr3が入っているので、これをr3に足せば、r3←r2になります。すなわち足し算を行うことでr3とr2が交換されます。（これはアセンブラの昔からあるテクニックです）
ループを抜け出たときのr3が8個のデータのうちの最大値です。

最大値を求めるプログラムは、引いた結果を足し戻すことで、レジスタの交換を行うため、引いたことによりレジスタが破壊される欠点が表面化しません。しかし、これは一般的にはうまく行かず、判断のために引き算を行うとレジスタが無駄になります。これを防ぐために、Flagという方法がよく使われます。Flagは演算結果の性質を示す小規模な専用レジスタです。Zero Flag、Minus Flag, Carry Flagなどがあり、それぞれ演算の結果によりセットされたりリセットされたります。分岐命令はこのFlagをチェックして分岐するかどうか判定します。BZ（Branch Zero)はZeroフラグが立っていれば成立します。ここで、比較（Compare）命令を用意します。この命令は、引き算を行うが、結果をレジスタに入れない命令で、フラグだけがセットされます。この命令を使えばレジスタを破壊せずに分岐ができます。フラグを使う方法は実装が簡単で効率が良いので様々なプロセッサで使われています。（死亡フラグというのは用法がこれと同じです。フラグが立っても死ぬとは限りません。フラグが立っても対応する分岐命令がなければ飛ばないのです）一方で、Flagを使うと命令コードの順番を入れ替えるのが難しくなります。この欠点はフラグセットオプションやグループ化である程度の改善が可能です。
一方、レジスタ同士を比較してその結果により分岐するCompare and Branchという方法もあります。これは比較命令と分岐命令が一体化しています。フラグの必要がなく、命令コードの入れ替えが可能ですが、一つの命令が複雑になってしまう問題点があります。

無条件に飛ぶ命令：ジャンプ

JMP X　pc ← pc+1+X　Jump

10100 XXXXXXXXXXX

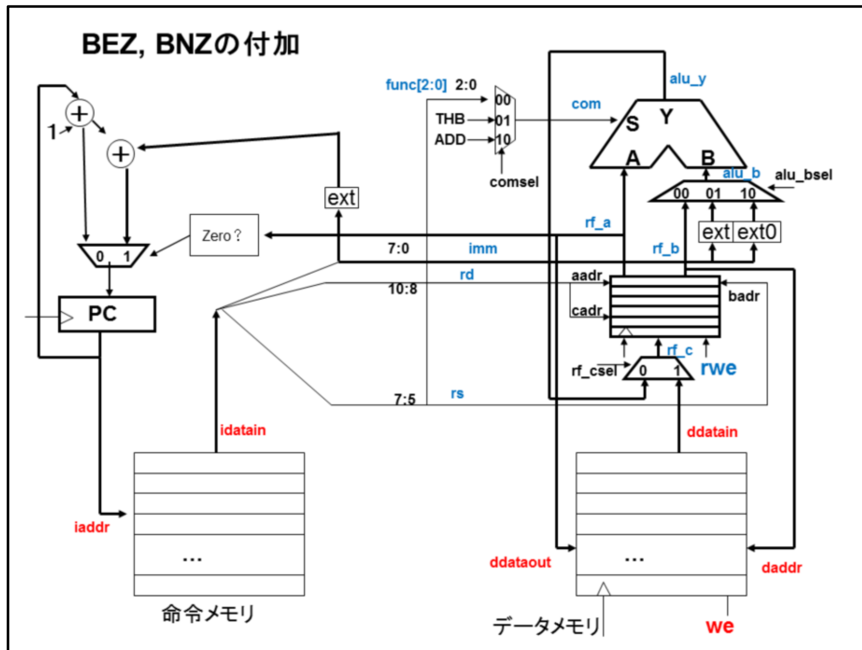
- 無条件に相対指定でX分飛ぶ
- 普通、無条件に飛ぶ命令をジャンプ、判断を含む命令をブランチと呼ぶ
- レジスタ指定がない分11bit分指定でき、遠くに飛べる
  - -1024～+1023
  - J型を使う

JR rd　pc←rd　Jump Register

00000 ddd --- 01010　R型なので注意！

- レジスタ間接ジャンプ
- 絶対指定
- 16ビットのアドレス空間のどこにでも飛べる最終手段
- サブルーチンコールのリターン（来週）
- テーブルジャンプ

無条件に飛ぶ命令をジャンプと呼びます（判断を含むのを分岐：枝分かれ、ブランチと呼びます。）この命令は無条件で相対指定で飛びます。飛び方は分岐命令と同じですが、レジスタ指定がない分とび先は8ビットから11ビットに広がり、－1024から＋１０２３まで、より広い範囲で飛べます。ジャンプ命令は新たな命令形式が必要になります。この命令形式はJ型と呼び、5ビットのオプコードの残り11ビットは全て飛び先に使われます。ブランチ命令よりも遠くに飛べます。
JR rdはレジスタ間接ジャンプと呼びます。これはR型でレジスタを1個だけ指定します。レジスタの値がそのまま飛び先に使われます。これは絶対指定です。16ビットのアドレス空間のどこでも飛べるので、遠くに飛びたい場合の最終手段と言えます。来週紹介しますが、サブルーチンコールのリターンにも使われます。他にも計算結果やメモリから値を読み込んで飛んでいくテーブルジャンプにも使われます。しかし、最近はあまりこのような使い方は推奨されません。



では、BEZ,BNZを付けてみましょう。レジスタの中身がゼロかどうか比較する操作は、ALUでも可能ですが、簡単なので専用判定器（Zero)に使います。これは、全ビットのORを取ってノNOTを取ればよいです。命令の下位8ビットは符号拡張してPC+1に加算します。分岐する、と判断された場合、PCにこの値を書き込みます。

分岐命令のVerilog記述

```
always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) pc <= 0;  // リセットならばPCは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;
        // 分岐の条件が成立すればPCに8ビットの相対指定を足す
    else if (jmp_op)
           pc <= pc +{{5{idatain[10]}},idatain[10:0]}+1;
        // JMP命令ならばPCに11ビットの相対指定を足す
    else if(jr_op)
           pc <= rf_a;
        // JR命令ならばレジスタの値をそのままPCにセット
    else
           pc<=pc+1;  // どれでもなければPCを先に進める
end
```

PCに対する操作はalwaysの中でif文が使えるので格好良く書ける！

分岐命令、JMP命令、JR命令はPCに対するalways文中に実装します。順番に分岐命令でそれぞれの条件が満足されるかどうかをチェックし、成立すれば8ビットを符号拡張したものをPC+1に足します。ジャンプならば11ビットを符号拡張したものを足します。JRの場合、レジスタファイルからの出力rf_aを書き込みます。どの条件も満足されなければPCに1を加えます。PCに対する操作はalways文中に書けるので格好良く書けます。

BEZ,BNZ,BPL,BMIの条件判定

```
assign bez_op =  (opcode == `OP_BEZ);
assign bnz_op = (opcode == `OP_BNZ);
assign bmi_op = (opcode == `OP_BMI);
assign bpl_op = (opcode == `OP_BPL);
.....
else if ((bez_op & rf_a == 16'b0) | (bnz_op & rf_a != 16'b0) |
        (bpl_op & ~rf_a[15]) | (bmi_op & rf_a[15]))
```

レジスタは、レジスタファイルのAポートから出力されるのでrf_aを調べればよい

rf_a[15]は符号ビット、これが1のときBMIは成立、0のときBPLは成立

BPLはプラスか0で分岐成立することに注意！

分岐命令はオプコードで当該命令がフェッチされたかどうかを判定します。これに条件を組み合わせて、分岐が成立するかどうかを判定します。対象となるレジスタはレジスタファイルのAポートから出てくるのでrf_aを使います。BMI、BPLは符号ビットを見るのでこの点を注意しましょう。

相対番地の計算

- 8ビットの符号拡張 Branch系の命令
  - **pc<=pc+{{8{imm[7]}},imm}+1;**
- 11ビットの符号拡張 JMP命令
  - **pc <= pc +{{5{idatain[10]}},idatain[10:0]}}+1;**
    - **8bitの直値はimmで定義してあるが11bitの直値は定義してない。このため命令コードidatainを直接扱っているのでちょっとみにくいかも、、、、**
- **＋1するのは起点が次の命令だから**
  - なぜこうなったのか？
  - マルチサイクル実装、パイプライン実装に適しているから
- **加算器がたくさん必要な気がするが、これは論理合成系がまとめてくれる**

相対番地の計算は分岐命令では8ビットを符号拡張し、ジャンプ命令では11ビットを符号拡張します。ジャンプ命令は、11ビットの直値を定義していないので、命令コードidatainの下位11ビットを直接拡張しています。このため少し見難いかもしれません。jimmなどの信号線名をつければよかったかもしれません。
ちなみに、＋1にして次の命令を起点としますが、これはマルチサイクル実装、パイプライン実装に適しているためです。このPC周辺では＋を多用するため加算器がたくさん必要になってしまいますが、これは論理合成系がまとめてくれるのでさほど気にしないでも大丈夫です。

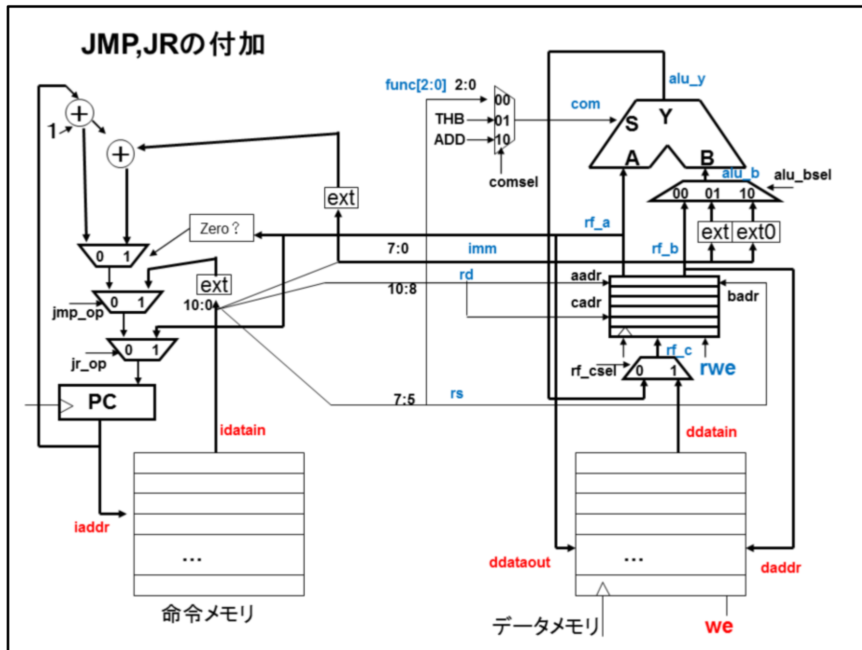
JMPとJR

```
assign jmp_op = (opcode ==`OP_JMP);
assign jr_op = (opcode ==`OP_REG) & (func == `F_JR);
…
else if (jmp_op)
        pc <= pc +{{5{idatain[10]}},idatain[10:0]}}+1;
      // JMP命令ならばPCに11ビットの相対指定を足す
  else if(jr_op)
        pc <= rf_a;
      // JR命令ならばレジスタの値をそのままPCにセット
```

jr_opはR型なのでデコードの方法が違うことに注意！
funcによって命令を識別する

後は判断がないので簡単！

JMPとJRは判断がないので、簡単です。JRはR型なのでfuncフィールドで識別する点に注意してください。rf_a、すなわちレジスタの値を直接PCに入れます。



JMP、JRを付加した全体の構造を示します。PC周辺はマルチプレクサは増えて、命令に応じて8ビット符号拡張、11ビット符号拡張、レジスタの値そのまま、PC+1のうち一つが選ばれてPCにセットされます。

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型命令の一覧を示します。JRが付け加わっています。

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

分岐命令は全てI型になります。

J型命令一覧

JMP #X	pc←pc+X+1	10100XXXXXXXXXXX

新たにJ型としてJMP命令が加わっています。

本日のまとめ

- POCOの分岐命令(BEZ,BNZ)はレジスタの内容で判定
- 飛び先はプログラムカウンタ相対指定
  - 起点は分岐命令の次の命令
- POCOでは大小比較はBPL、BMIで行う
  - 引き算を伴うのでレジスタが破壊される
  - 他のプロセッサではFlagやCompare and Branchを使う
- JMP命令は判断がないが11ビットの範囲で遠くに飛べる
- JR命令はレジスタの内容で絶対ジャン

アセンブラshapaは

./shapa XXX.asm –o imem.dat

ラベルを使おう

インフォ丸が教えてくれる今日のまとめです。

演習

1．1番地に入っているXについて
　1+2+3+…＋Xを計算するプログラムを書け
　mult.asmを参考に
　提出物xsum.asm

2．0から並んでいる8個の数字の総和を求めて
　8番地に書き込むプログラムを書け
　8回足してはならない。ループを作る事
　答えは35（16進で23）になるはず
　提出物sum8.asm

今回はプログラムを重点的にやってみましょう。