第7回
MIPSのアセンブラプログラム

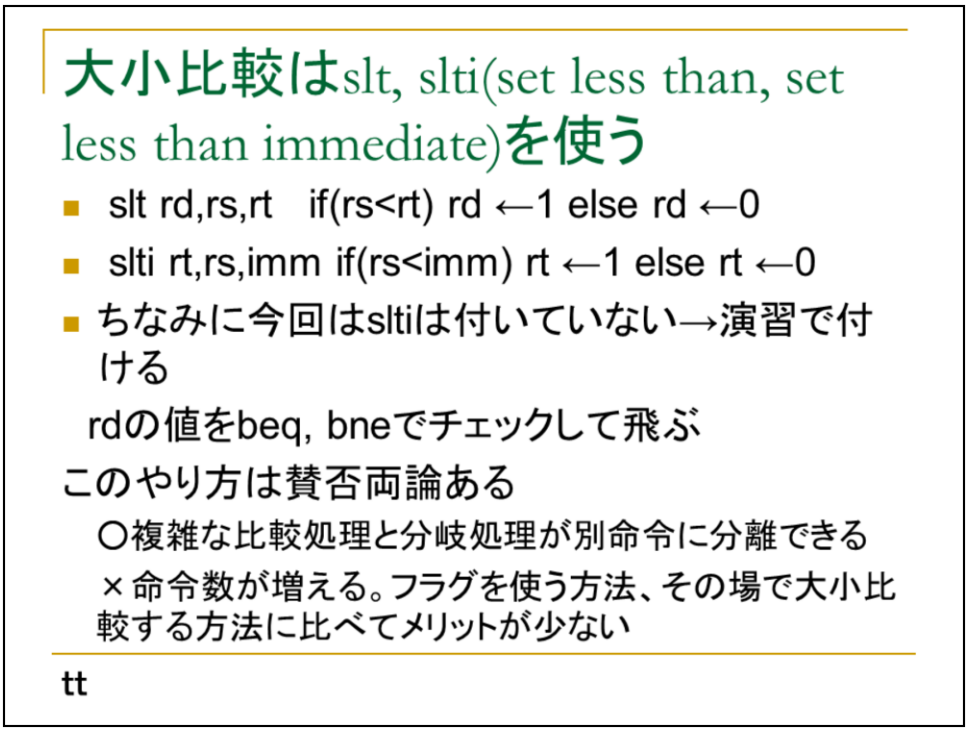
天野　hunga@am.ics.keio.ac.jp

前回までにやった基本的な命令

- メモリアクセス命令
  - lw, sw
- レジスタ間演算命令
  - add, sub, or, sllv, srlv
- イミーディエイト命令
  - addi, ori, lui, sll, srl
- 分岐命令
  - beq, bne

  → 等しいかどうかを判定するだけなので大小比較ができない

前回MIPSの命令を紹介しましたが、制御系の命令は簡単な分岐命令のみにとどめておきました。これだと2つのレジスタを比較して等しいかどうかを判定して分岐するので、大小比較ができません。



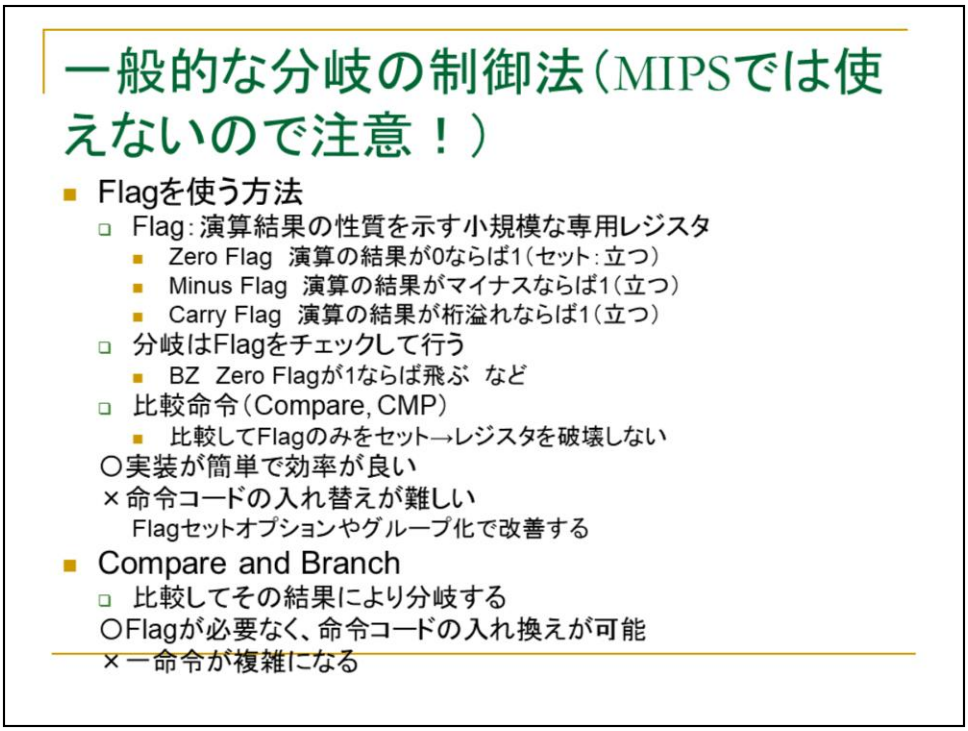
大小比較を行う場合、MIPSではslt (set less than)とslti(set less than immediate)と呼ぶ比較命令を使います。この命令は2つのレジスタ、あるいはレジスタとイミーディエイトを比較して、その結果をレジスタに格納します。slt rd,rs,rtを実行するとrs<rtの場合は、rdに1を、そうでなければ0をセットします。slti rt,rs,immはrs<immの時はrtに1を、そうでなければ0をセットします。（ちなみに今日の設計にはsltiが付いておらず、演習の時につけてもらいます。）

このようにして比較結果が1か0になれば、beq, bneでこれをチェックして分岐することができます。しかし、このやり方は賛否両論があります。大小比較は複雑な処理なので、これを分岐処理と別の命令で実行することで、実装が楽になり動作速度が改善されます。一方で、フラグを使う方法や、その場で大小比較をする方法に比べてそのままではメリットが少ないです。実はこの方法はパイプライン処理のスケジュールがしやすいメリットがあります。これは後程説明します。

最大値を選ぶプログラム例 max.asm

```
        add $1,$0,$0  //      ポインタは$1
        add $3,$0,$0  //      $3は暫定チャンピオン
        addi $2,$0,8  //    調べる数は8つ
loop:   lw $4,0($1)   //      $4は挑戦者
        slt $5,$4,$3  //
        bne $5,$0,skip  //  チャンピオンが勝てばスキップ
        add $3,$0,$4   //    挑戦者をチャンピオンに
skip:   addi $1,$1,4  //      ポインタを進める
        addi $2,$2,-1 //      カウンタを減らす
        bne $2,$0,loop //     8個調べたらおしまい
end:    beq $0,$0, end // Dynamic Stop
```

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。$1はポインタ、＄2はカウンタでそれぞれ0と8を入れておきます。$3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$3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。＄4に＄1をポインタとして値を取って来ます。これが挑戦者です。チャンピオンと挑戦者をsltで比較し、チャンピオンが勝てば、＄5が1になります。これをbneで調べて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。ここで比較の結果が0か正ならば、挑戦者が勝ったことになります。この場合bneは成立せず、次のadd $3,$0,$4が実行され、チャンピオンが交代します。これを繰り返し、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。

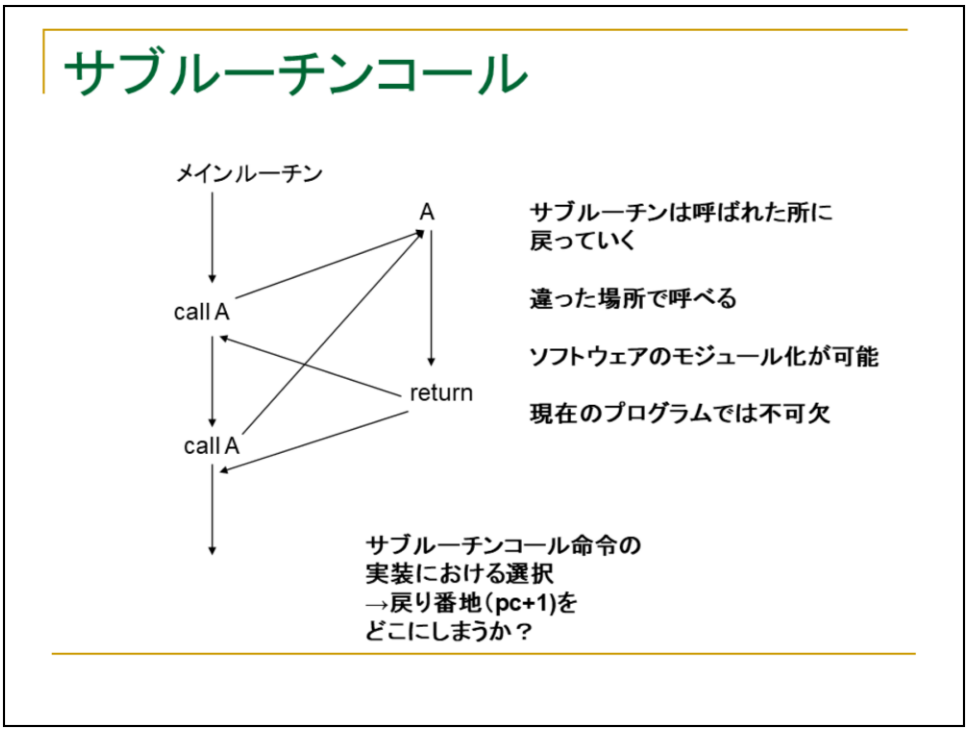


slt命令はたった1ビットのためにレジスタ全体を使ってしまいます。これを防ぐために、Flagという方法がよく使われます。Flagは演算結果の性質を示す小規模な専用レジスタです。Zero Flag、Minus Flag, Carry Flagなどがあり、それぞれ演算の結果によりセットされたりリセットされたります。分岐命令はこのFlagをチェックして分岐するかどうか判定します。BZ（Branch Zero)はZeroフラグが立っていれば成立します。ここで、比較（Compare）命令を用意します。この命令は、引き算を行うが、結果をレジスタに入れない命令で、フラグだけがセットされます。この命令を使えば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。

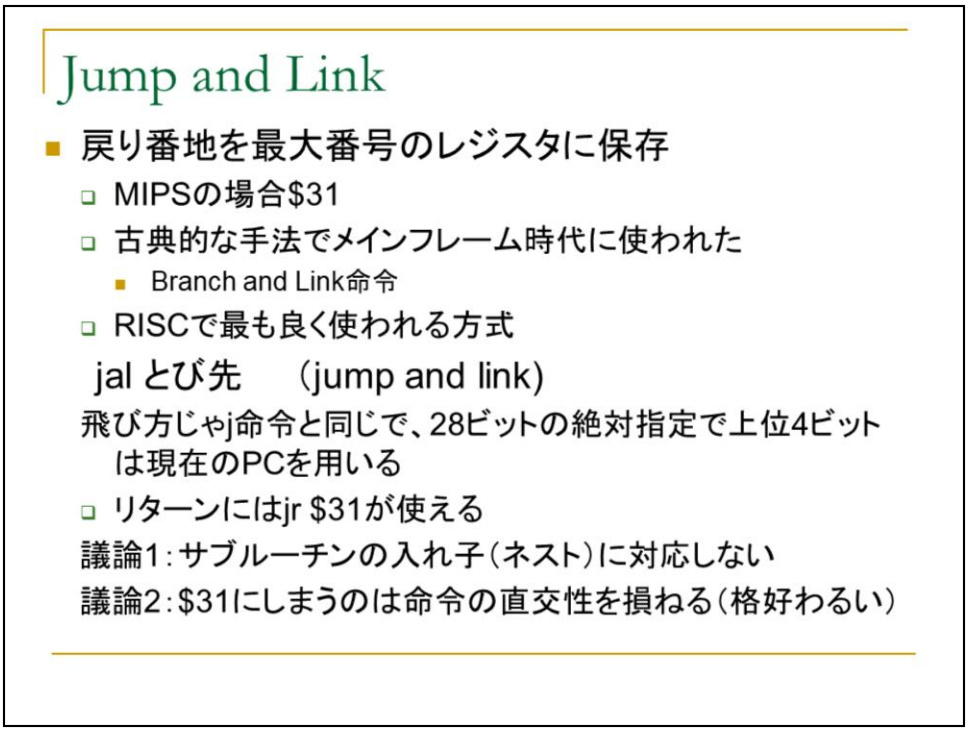
一方、レジスタ同士を比較してその結果により分岐するCompare and Branchという方法もあります。これは比較命令と分岐命令が一体化しています。例えば、blt(branch less than) rt,rsという命令を作ればいいのです。これならフラグの必要がなく、命令コードの入れ替えが可能ですが、一つの命令が複雑になってしまう問題点があります。大小比較は等しいかどうかの比較に比べてずっと遅延時間が掛かります。

j(jump)は無条件で飛ぶ命令です。この命令は、普通の分岐命令が16ビットの範囲であるのに対して、26ビットの範囲で飛び越す命令数を指定できます。すなわち、メモリの番地では28ビットの範囲で飛んでいけます。全体のメモリ空間が32ビットなので、かなり遠くに飛んでいけると言えます。しかし、この命令はPC相対指定ではなく、絶対指定で、28ビットで指定できない上位4ビットは、現在のPCの値が入ります。これは、このジャンプ命令が全体のアドレス空間を16に区切ってその一つから外には出られないことを意味します。これはちょっと変なのではないかと思うのですが、事実上これで十分ということでこのような仕様になっているのだと思います。この特徴を嫌ってこの命令を使わない人もいます。16ビットの範囲で良ければ、beq $0,$0,飛び先を、常にジャンプする命令として使えばよいからです。

jr(jump register)はレジスタ間接指定で、32ビットのレジスタの中身の番地のどこにでも飛べます。後に紹介するようにサブルーチンコールのリターンに使います。ただし、下位2ビットは00で4の倍数にしておかなければならないです。

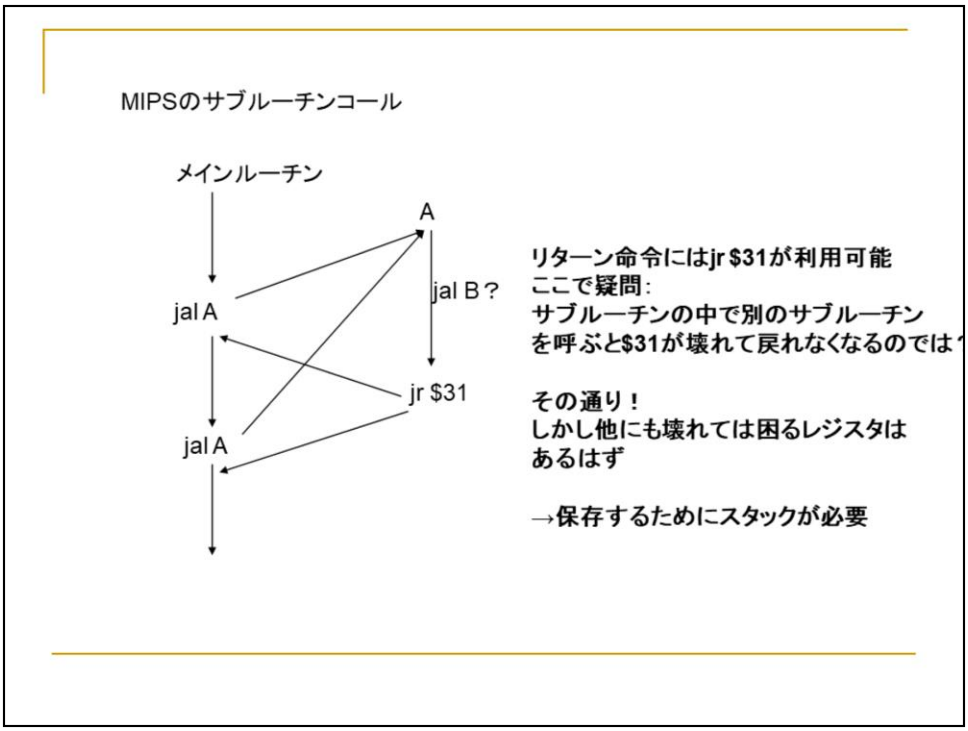


分岐命令、ジャンプ命令と違ってサブルーチンコール命令は、呼ばれた所（次の命令）に戻ってくる点が特徴です。図の例ではAを呼び出して、リターン命令実行時にコール命令の次に戻ります。Aは色々なところで使えるため、ソフトウェアのモジュール化が可能です。この考え方は現在のプログラムでは不可欠です。問題は、戻り番地（すなわちコール命令実行時のPC+4）をどこにしまっておくか？という点です。



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。MIPSでは戻り番地は$31に格納します。飛ぶ方は、j命令と同じく28bitの範囲の絶対番地で、上位4ビットは現在のPCの値を使います。この方法では戻り番地(PC＋4)は$31に格納されているため、リターン命令はjr $31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$31が破壊されてしまう点です。もう一つは$31にしまうことにより、$31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。



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、$31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 $31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、$31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、$31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。

例題3

- 掛け算のサブルーチン（＄1と＄2を掛けて＄3に入れる）を用いて自乗を計算するプログラムを実行せよ
- jijo.asmを実行して結果を確認

では例題を見てみましょう。これはサブルーチンコールの例題です。

2乗を計算する例

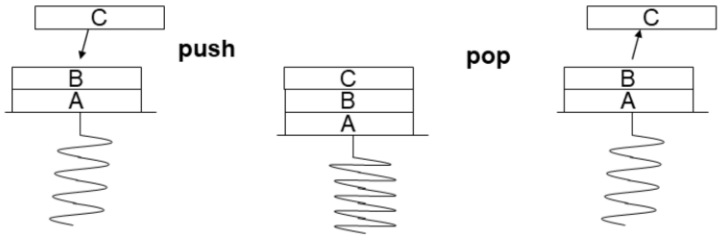
```
        lw $1, 0($0)
        lw $2,0($0)
        jal mult
end:    j end

// Subroutine Mult  $3 ←  $1×$2 ここで$1は破壊される
mult:   add $3,$0,$0
loop:   add $3,$3,$2
        addi $1,$1,-1
        bne $1, $0,loop
        jr $31          ←  $31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、$2の値と$3の値を掛け算して、答えをr3に返します。ここでは$3は破壊されてしまいます。まず、メインルーチンでは$1を0にセットして、0番地の内容を$2にロードします。$2の内容を$3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。

スタック

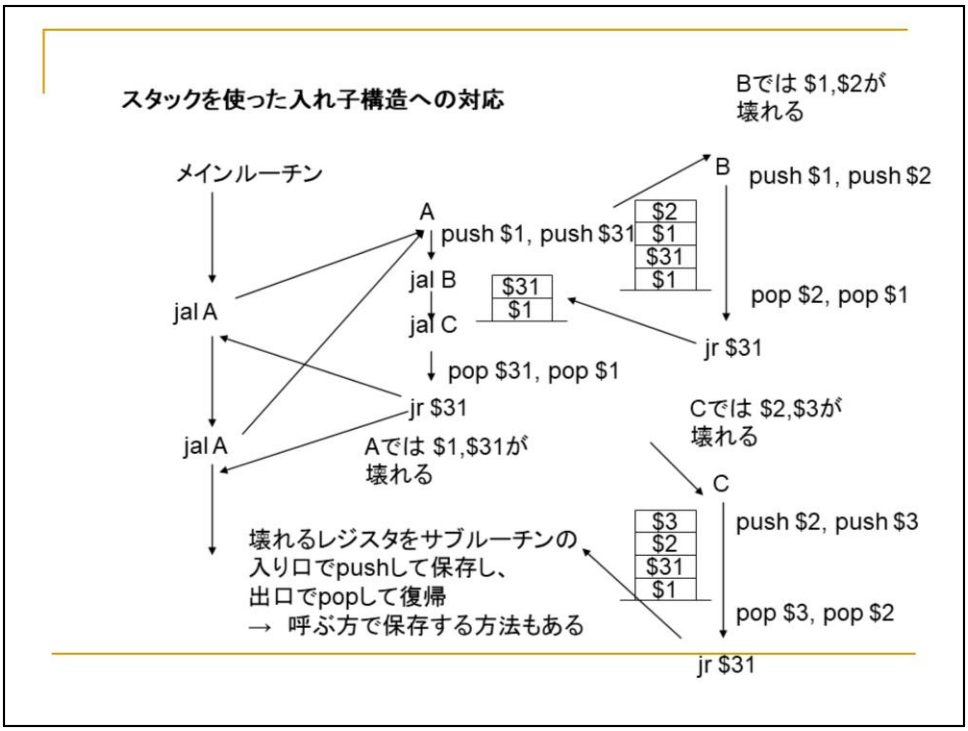
データを積む棚
push操作でデータを積み
pop操作で取り出す

- 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

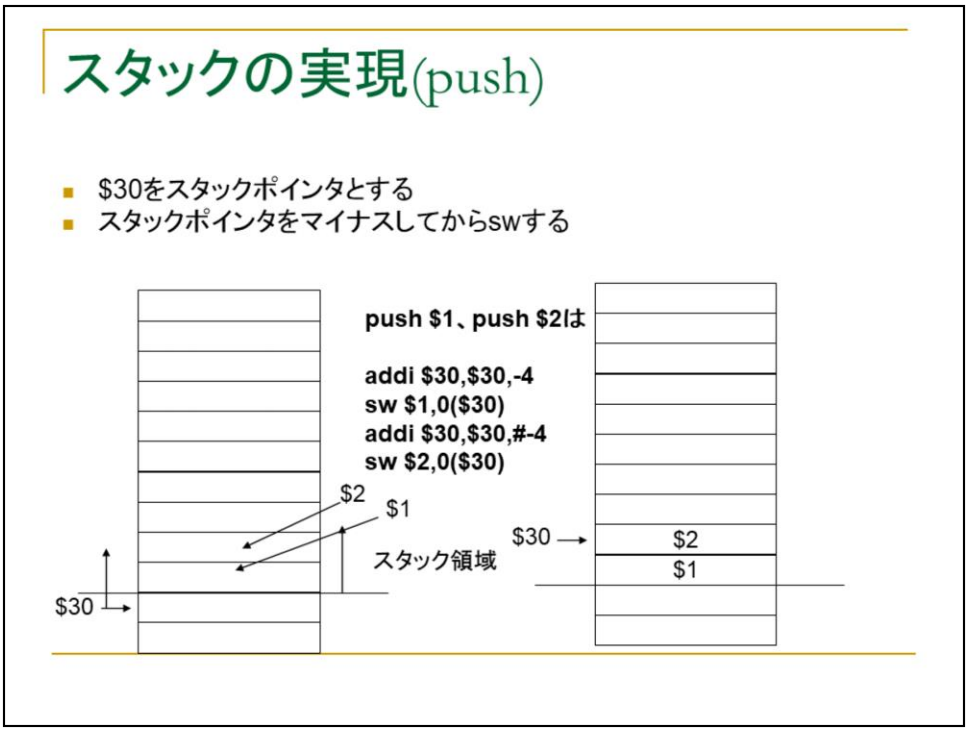


スタックとは、データを積む棚です。この棚にデータを積む操作をpush、棚から取り出す操作をpopと呼びます。先に積んだものが後から取り出されることからLIFO (Last In First Out)と呼びます。逆に考えると、後に積んだものが先に取り出されるのでFILO（First In Last Out)と呼ぶ場合もあります。この積んだ逆順に取り出すことのできる性質からサブルーチンコール時にレジスタを退避するのに適しています。

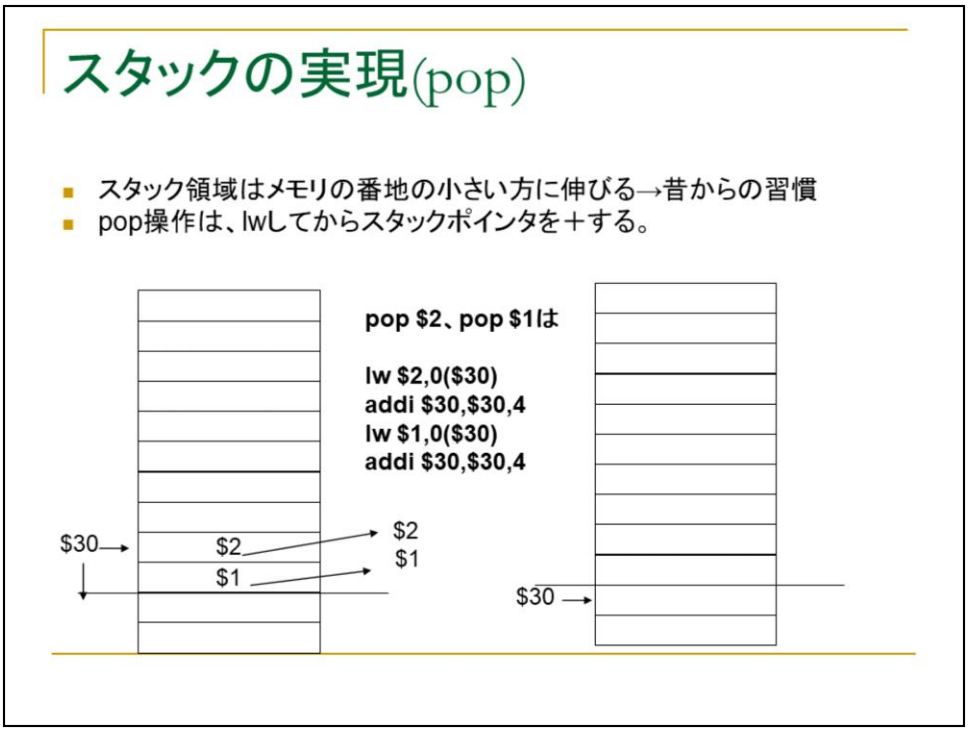
以前紹介したスタックマシンで利用した演算スタックは、演算用の特殊なメモリですが、サブルーチンコールのレジスタの退避用のスタックは主記憶上に確保するのが普通です。スタックは棚ですが、ばねがついているイメージがあります。データを積むときは押し込むイメージからpushと呼び、取り出すときは、飛び出すイメージからpopと呼ばれます。



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$1を使います。中で別のサブルーチンを呼ぶので$31も退避します。サブルーチンBを呼んだ際に$1、$2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$1、$31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)



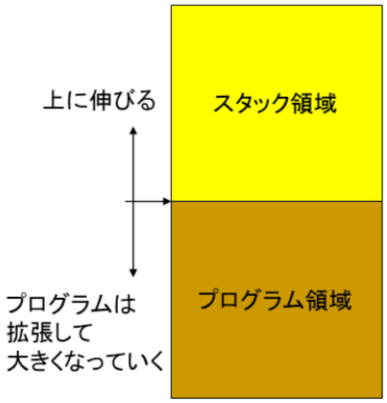
スタックをメモリ上に実現するには、スタックポインタを使います。ここでは$30をスタックポインタの役割に使います。スタックポインタは、スタック領域の一番上の番地＋1の所に初期化します。push操作は、まずスタックポインタを減らし、空いた領域にレジスタを書き込みます。スタック領域はメモリ上の番地が減る方向に伸びていきます。この図はpush $1, push $2を順に実行した様子を示します。



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 $2, pop $1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

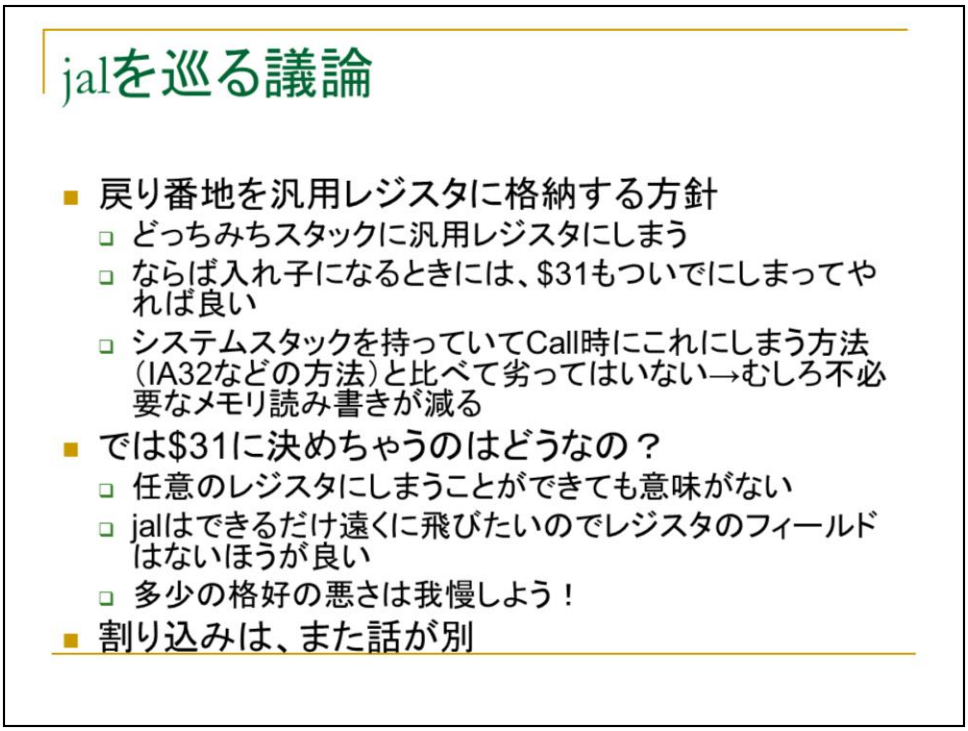


スタックは番地の小さくなる方向に(この図では上に向かって)伸びるのが普通ですが、これは昔からの習慣に基づいています。昔はある場所に境界を引いて、それより小さい番地はスタック領域とし、大きな番地はプログラム領域としました。これはプログラムというのは開発を進めると大きくなるので、番地が増える方向に伸びます。スタックを番地の増える方向に成長させると方向が同じになり、成長したスタックがプログラム領域を食い荒らす可能性があります。そこで、スタックは番地が小さい方向に伸ばす習慣ができました。もちろん領域を食いつぶしてしまえば、どちらの方向でもトラブルになります。

掛け算用サブルーチン

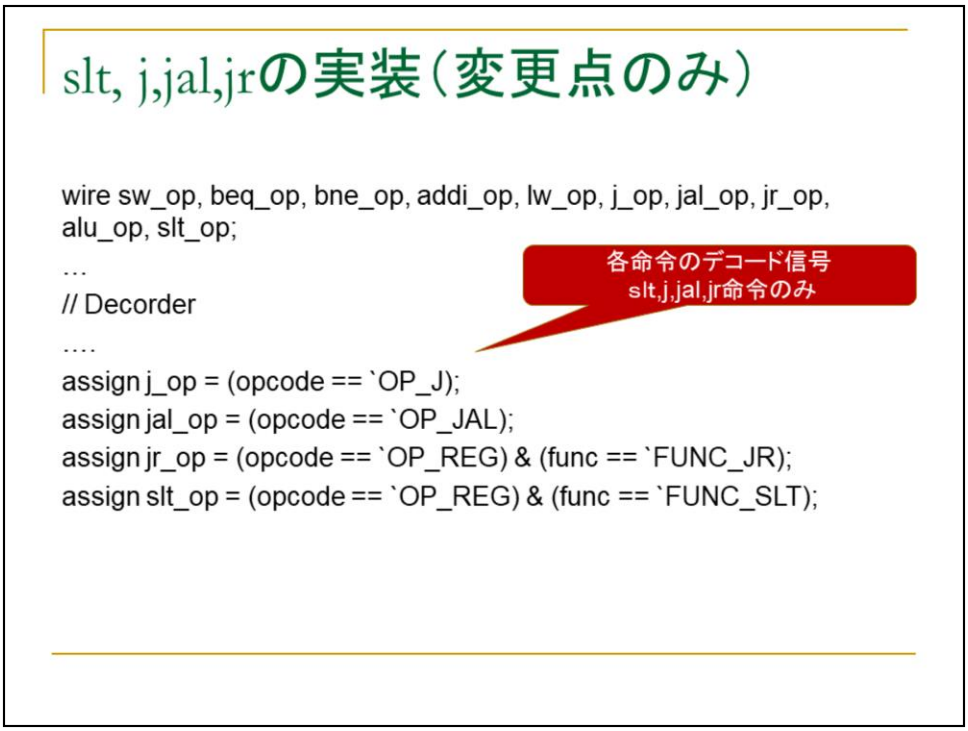
```
$1を破壊しないサブルーチンコール
$30はメインルーチンで初期化する必要がある
// Subroutine Mult  r3 ← $2 × $3
mult:   addi $30,$30,-4  この2行で$3をスタックにpush
        sw $1,0($30)     $1の値がスタックに退避される
        add $3,$0,$0
loop:   add $3,$3,$2
        addi $1,$1,-1
        bne $1, $0,loop
        lw $1,0($30)    この2行で$1をスタックからpop
        addi $30,$30,4  $1の値がスタックから復帰する
        jr $31          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。$3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、$2同様、メインルーチンで値を使うことができます。

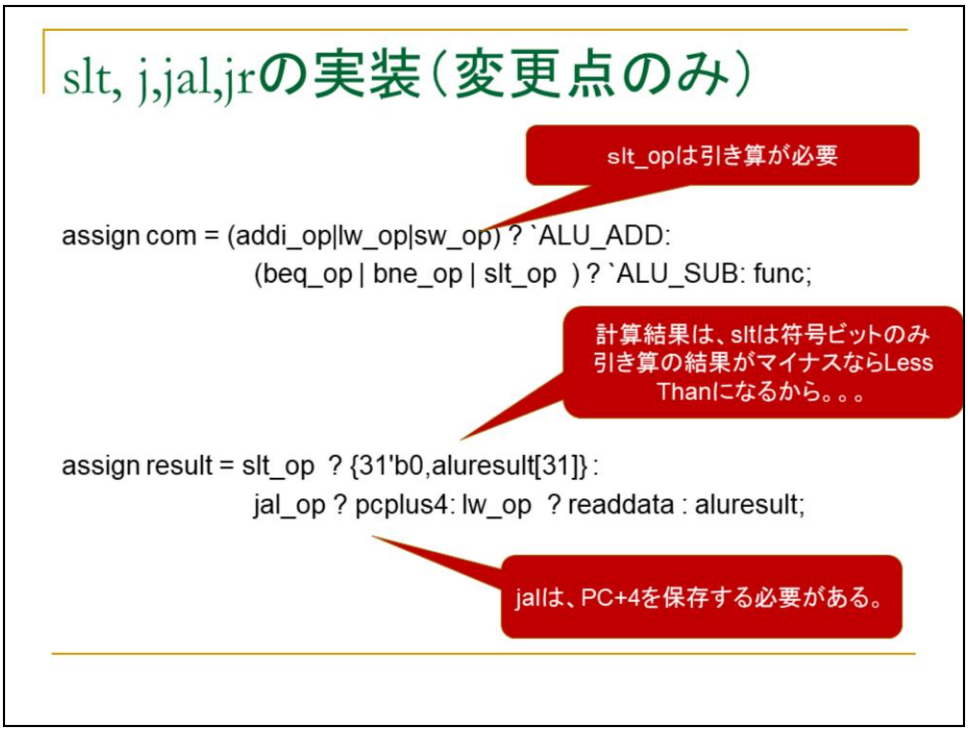


jalは戻り番地を汎用レジスタ$31にしまうため、サブルーチンが入れ子になると壊れてしまいます。しかい、どっちみち他の汎用レジスタだってメインルーチンとサブルーチンで両方使う場合は、壊れるのでスタックにしまう必要があります。サブルーチンの入れ子になる場合はこれと一緒に$31もしまっていまえばいい、という考え方です。これはリーゾナブルだと思います。Intelのx86（IA32)などでは、システムスタックというシステムで管理するスタックを持っていてサブルーチンコール時に戻り番地をこれに自動的にpushする方法を取ります。これはサブルーチンの入れ子に対応可能ですが、入れ子でない場合も強制的にスタック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。

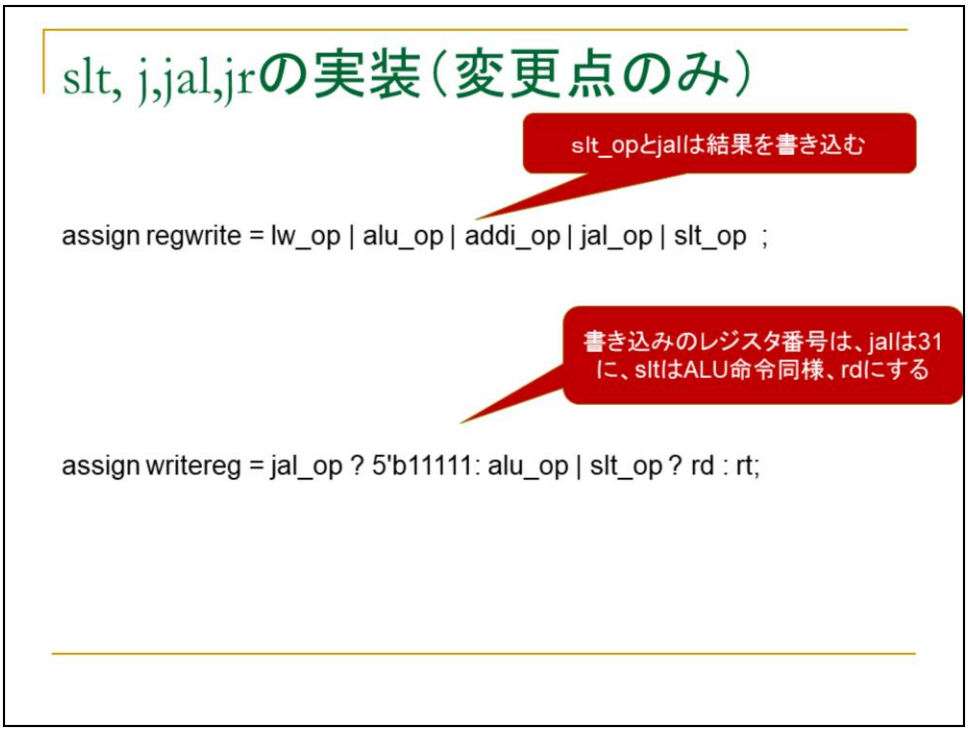
次に$31に決めてしまっていますが、これはどうでしょう？jalは出来る限り遠くに飛びたいです。これは、サブルーチンは、ライブラリとして、ユーザープログラムとは別の番地に置かれる場合が多いためです。戻り番地をしまうレジスタはどっちみちどこかに決めてしまいます。ならば、これを$31に決めてしまい、残りの全てのビットを飛び先を決めるアドレスに充てた方が良いです。このことにより命令の直交性が低下します。直交性とは、ある操作をレジスタ番号や命令の種類に関係なしに施すことができるかどうかを示す性質です。直交性の高い命令は美しい命令になります。$31のみ戻り番地をしまえることにより直交性は低下しますが、この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。



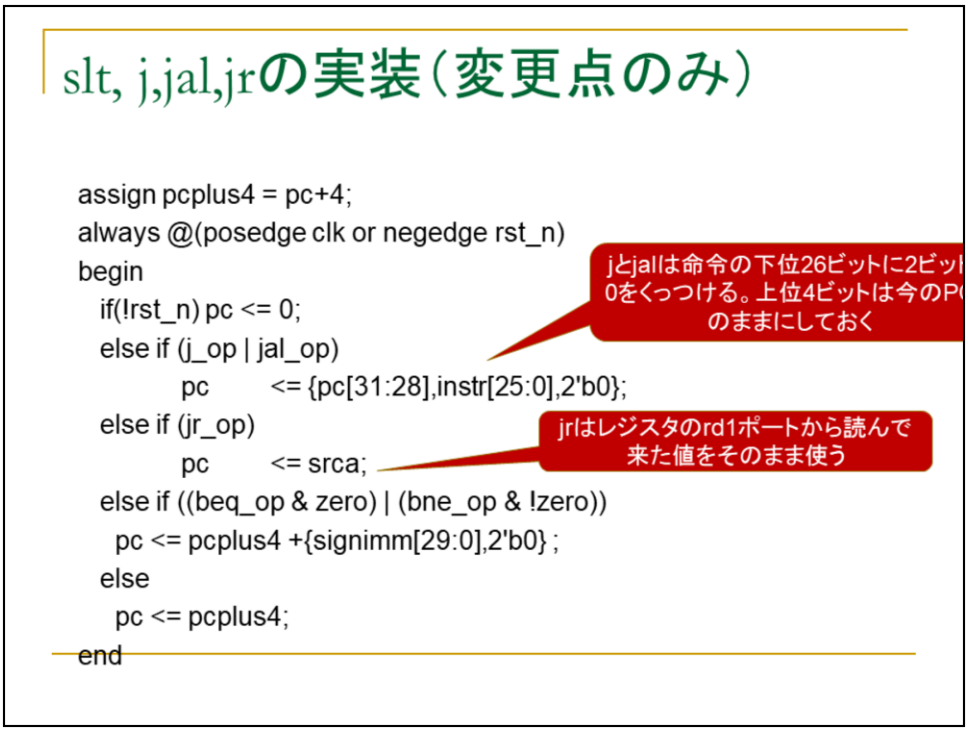
では、slt,j,jal,jrのVerilog記述の変更点のみを説明します。まずはデコード部分で、これは他の命令と変わりません。



次にALUのコマンドですが、slt命令では引き算をします。レジスタファイルに書き込む結果については、sltは比較の結果を書き込みます。これは実は簡単で、Less Than（小さい）が成立すると結果がマイナスになるので、その符号ビットだけを取って最下位に入れて他を0にすればいいです。jalは戻り番地であるpc+4を結果として書き込みます。



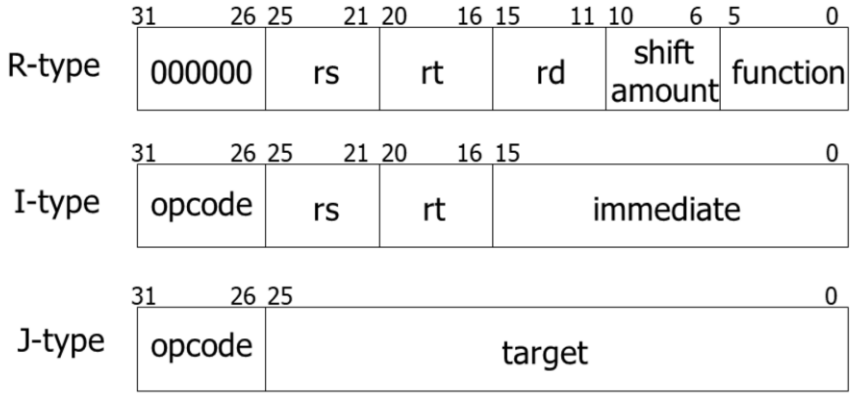
slt, jalは結果を書き込むのでregwriteが1になるようにします。書き込みレジスタの番号を示すwriteregはjalの場合31にしなければならないです。この辺が直交性を乱した罰が当たっており、恰好悪い記述です。sltの結果はALU演算命令同様にrdに書き込みますの。



pc周辺ではjとjalは同じJ型で、命令の下位26ビットに2ビットの0をくっつけて飛び先の下位28ビットとします。上位4ビットは今までのpcをそのままくっつけます。jrはもっと簡単で、レジスタファイルのrd1から読んできたsrcaをそのまま入れてやります。

命令フォーマット

- 3種類の基本フォーマットを持つ



命令フォーマットは前回紹介しましたが、jal, jはJ-typeを使っています。

現在使えるR型命令一覧

add rd,rs,rt	rd ←rs+rt	000000ssssstttttddddd100000
sub rd,rs,rt	rd ←rs-rt	000000ssssstttttddddd100010
and rd,rs,rt	rd ←rs&rt	000000ssssstttttddddd100101
or rd,rs,rt	rd ←rs|rt	000000ssssstttttddddd101010
slt rd,rs,rt	rs<rt rd←1 else rd←0	000000ssssstttttddddd101010
jr rs	pc ← rs	000000sssss0000000000001000

では現在使える命令の一覧を示します。

I型命令一覧

lw rt,offset(base)	ワードロード	100011tttttbbbbbb offset
sw rt,offset(base)	ワードストア	**101011tttttbbbbbb offset**
addi rt,rs,imm	rt←rs+(符号拡張) imm	001000ttttsssss imm
slti rt,rs,imm	rs<(符号拡張) imm rd←1else rd←0	**001010ttttsssss imm**
beq rs,rt, offset	rs=rtで分岐	**000100tttttssssss offset**
bne rs,rt, offset	rs≠rtで分岐	**000101tttttssssss offset**

lw, swはディスプレースメントを伴うため、I型になります。

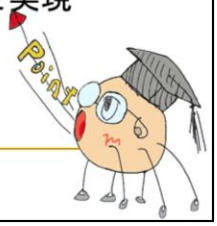
J型命令一覧

j offset	不完全絶対分岐	**000010 offset**
jal offset	不完全絶対指定のサブルーチンコール $31←pc+4	**000011 offset**

J型はjとjalのみです。

本日のまとめ

- MIPSは32ビットのRISC、アドレス、データ共に32ビット
- アドレスはバイトアドレッシング、32ビット命令、データは4の倍数のみ
- 32ビットのレジスタを32本持つ。レジスタ0は常に0
- ディスプレースメント付きレジスタ間接指定でメモリのアドレスを指定
- 3オペランド
- 条件分岐はレジスタ二つを比較、PC相対指定
- 大小比較は比較命令と分岐命令の組み合わせで実現
- jとjalは制限付きの絶対指定



インフォ丸が教えてくれる今日のまとめです。

演習1

- 0番地から並んだ8つの値（正の数）の最小値を選ぶプログラムを書け。ただしこの8つの数には0は含まれないとする。
- 結果はどこかのレジスタに入れておけばよい

演習2

- 0番地の内容をXとしたとき、掛け算のサブルーチンを利用してXの3乗を計算せよ。
- 結果はどこかのレジスタに入れておけばよい

5×5×5＝125(7D)になるはず

演習3

- jalr rsを実装せよ

opcode 000000 func 001001

jijo2.asmを実行して動作することを確認せよ

提出物：jalrの付いたmipse.v(他のが付いていてもかまわない)

覚えておくと便利

tarの解凍
tar xvf file.tar

アセンブラ
./asm.pl file.asm –o imem.dat

論理シミュレーションiverilog
iverilog *.v
vvp a.out (./a.out | more)

波形ビューアgtkwave
gtkwave mipse.vcd

資料
http://www.am.ics.keio.ac.jp

レポート提出
keio.jp経由で提出のこと