第7回
RV32I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, sll,srl,sra,xor,or,and,slt,sltu
- イミーディエイト命令
  - addi, slli,srli,srai,xori,ori,andi,slti,sltiu, lui
- 分岐命令
  - beq, bne, bgt, ble, bgtu, bleu

今までrv32i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。

最大値を選ぶプログラム例 max.asm

```
        add x1,x0,x0  //      ポインタはx1
        add x3,x0,x0  //      x3は暫定チャンピオン
        addi x2,x0,8  //    調べる数は8つ
loop:   lw x4,x1,0    //      x4は挑戦者
        blt x4,x3,skip  //    チャンピオンが勝てばスキップ
        add x3,x0,x4   //     挑戦者をチャンピオンに
skip:   addi x1,x1,4   //     ポインタを進める
        addi x2,x2,-1  //     カウンタを減らす
        bne x2,x0,loop //     8個調べたらおしまい
end:    beq x0,x0, 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。x1はポインタ、x2はカウンタでそれぞれ0と8を入れておきます。x3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x3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。x4にx1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、x5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 x3,x0,x4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x3が8個のデータのうちの最大値です。

Flagを使った分岐（RV32Iでは使えないので注意！）

- Flagを使う方法
  - Flag：演算結果の性質を示す小規模な専用レジスタ
    - Zero Flag　演算の結果が0ならば1（セット：立つ）
    - Minus Flag　演算の結果がマイナスならば1（立つ）
    - Carry Flag　演算の結果が桁溢れならば1（立つ）
  - 分岐はFlagをチェックして行う
    - BZ　Zero Flagが1ならば飛ぶ　など
  - 比較命令（Compare, CMP）
    - 比較してFlagのみをセット→レジスタを破壊しない

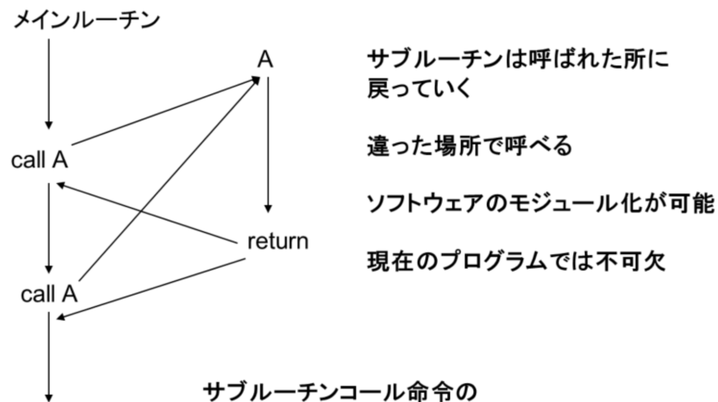
  ○実装が簡単で効率が良い

  ×命令コードの入れ替えが難しい

  Flagセットオプションやグループ化で改善する

RV32Iでは、比較と分岐が一体化したCompare & Branchという方法を使っています。これは命令がやや複雑になる問題点があります。プロセッサの中では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。

サブルーチンコール



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。

サブルーチンコール

jal rd,offset　　jump and link

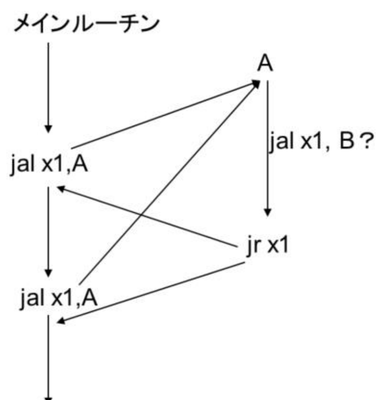
- 戻り番地(PC+4)をx[rd]レジスタに保存してジャンプ
  - 飛び方は分岐命令と同じPC相対指定だが、offsetは21ビット分（最下位0は省略）
  - rdにx0を指定すると、単純なジャンプ命令になる。

jalr rd,rs1,offset (jalr rd,offset(rs1))　jump and link register

- 　x[rs1]に符号拡張したoffsetを加えて、飛び先を計算、この際最下位ビットを強制的には0にする。戻り番地（PC+4)をx[rd]レジスタに保存
- offsetを0にすると単純なレジスタ間接サブルーチンコール
- rdもx0にすると単純なレジスタ間接ジャンプ
- 以降面倒なので、jalr x0,x1,0をjr x1と書く（疑似命令）

jal(jump and link)はサブルーチンコールで、戻り番地（PC+4)をrdに入れます。飛び方は分岐命令と同じPC相対指定ですが、遠くに飛べるようにoffsetは21ビット分あり最下位の0は命令中では省略されています。戻り番地を保存する必要のない無条件ジャンプにするためにはrdにx0を指定します。beq x0,x0,offsetでも無条件ジャンプになりますが、こちらの方が遠くに飛べます。jalr(jump and link register)は、x[rs1]に12ビットのoffsetを加算したものが飛び先になります。この時、最下位ビットが1になると困るので、強制的に0にします。同時にx[rd]に戻り番地を格納します。この命令は、offsetを0にすると単純なレジスタ間接サブルーチンコールになり、rdをx0にすると単純なレジスタ間接ジャンプになります。以降、面倒なので、jalr x0,x1,0をjr x0（jump register)と書きます。このように、実際は他の命令に置き換えて実行される命令を、あたかも存在するかのように扱う方法を疑似命令と呼びます。疑似命令はアセンブラで置き換えられます。

RV32Iのサブルーチンコール



リターン命令にはjr x1(jalr x0,x1,0)を使う
ここで疑問:
サブルーチンの中で別のサブルーチンを呼ぶ度に戻り番地のレジスタを変えていると足りなくなるのでは？

その通り！
しかし他にも壊れては困るレジスタはあるはず

→保存するためにスタックが必要

jal x1,Aはリターン命令としてjr x1を使えば、PCが復帰して戻ってこれます。これを使ってサブルーチンを呼び出すことが可能になります。しかし、このやり方は、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を呼ぶ時、同じrdを指定すると内容が破壊されてしまうため、サブルーチンAの最後にjr x1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、x1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、x1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。

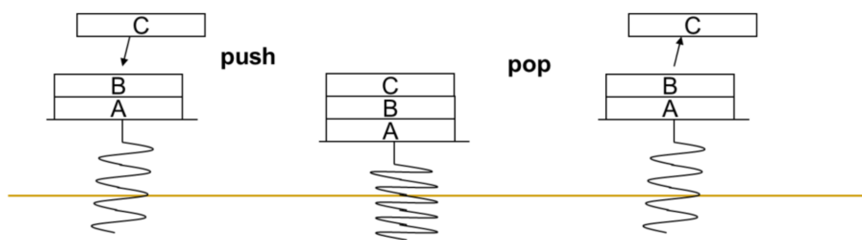
スタック

データを積む棚

push操作でデータを積み

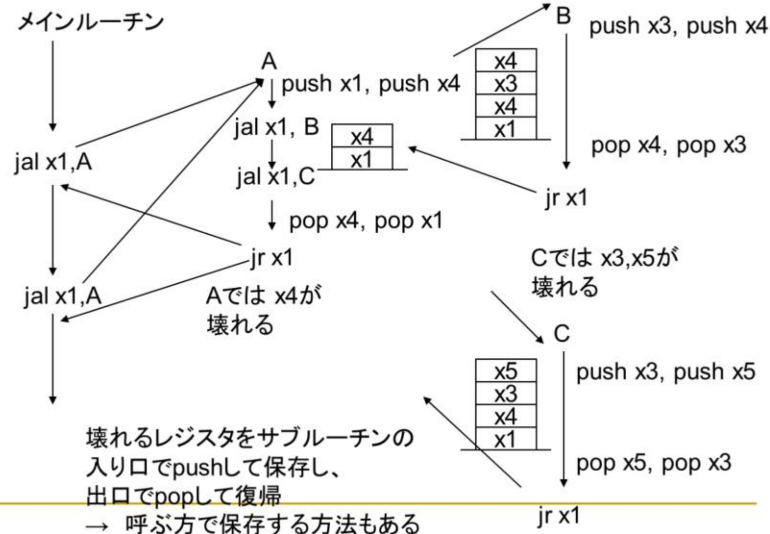
pop操作で取り出す

- 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。

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。

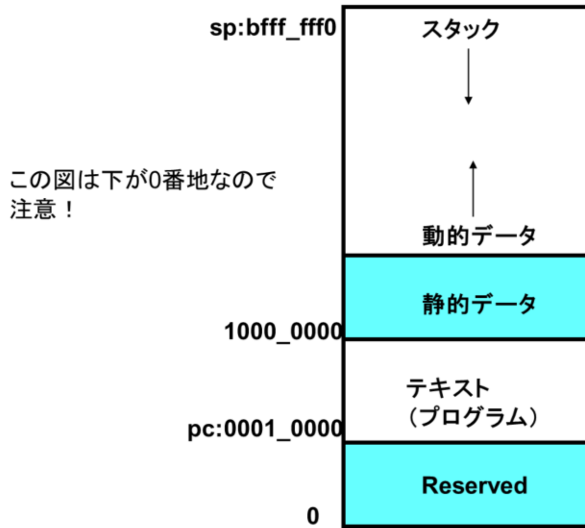


スタックを使ってレジスタを退避する様子を示します。この例では、サブルーチンの入り口で、中で使って壊れるレジスタを退避し、リターンする直前に復帰する方法を示します。これはコーリーサーブと呼びます。逆に呼ぶ側で、壊れて困るレジスタをスタックに積んでからサブルーチンを呼び出す方法(コーラーセーブ)もあります。サブルーチンAではx4を使います。中で別のサブルーチンを呼ぶのでx1も退避します。サブルーチンBを呼んだ際にx3、x4を退避します。この2つのレジスタはサブルーチンAを呼んだ際のx1、x4の上に積まれます。サブルーチンBの中でさらに別のサブルーチンを呼ぶ場合、さらにこの上に積み重なります。サブル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)

RV32Iのレジスタの標準的使用法

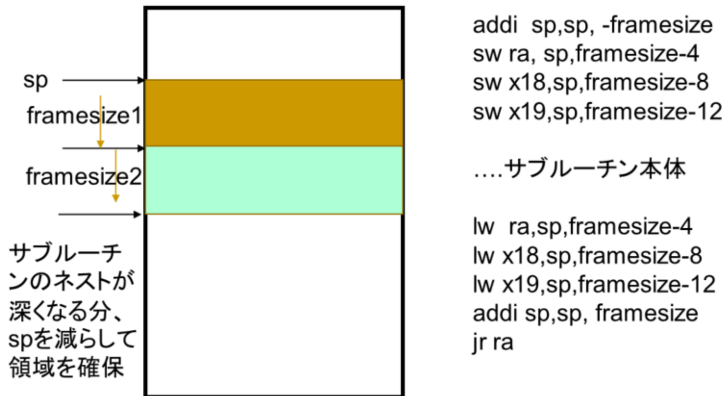
レジスタ	表記	利用法	呼び出し前後で保持
x0	zero	常に0	
x1	ra	戻りアドレス	
x2	sp	スタックポインタ	保存
x3	gp	グローバルポインタ	
x4	tp	スレッドポインタ	
x5	t0	一時/代替リンクレジスタ	
x6-7	t1-2	一時レジスタ	保存
x8	s0/fp	保存レジスタ/フレームポインタ	保存
x9	s1	保存レジスタ	
x10-11	a0-1	関数の引数/戻り値	
x12-17	a2-7	関数の引数	
x18-27	s2-11	保存レジスタ	保存
x23-31	t3-6	一時レジスタ	

サブルーチンコールを行う場合、場合によってはメモリに保存する必要はなく、呼ぶ側と呼ばれる側でレジスタを分離しておけば問題ないです。またサブルーチンコールを行う場合、一定のルールを設けておくのが普通です。このっ表にそれを示します。ここでは、x1を戻りアドレス、x2をスタックポインタとして使うことだけはこの表に従い、後は特に考えずに使うことにします。



RV32Iのメモリ領域を示します。この図は下が0番地で上に行くほど番地が増えるのでご注意ください。プログラムは0001_0000から貼り付けます。PCもリセットすると0001_0000になるように設定します。（ここでは0番地にしてしまいます。）1000_0000からは静的なデータを割り付け、スタックはbfff_ffffから番地の下の方に伸ばしていきます。

RV32Iにおけるスタックの実現



実際にスタックを作る場合の方法を示します。サブルーチンに入った時に、まずそのサブルーチンで必要なレジスタを数からフレームのサイズを計算し、その分スタックポインタを減らします。そしてその領域内にレジスタを格納して行きます。ここではディスプレースメントが威力を発揮します。サブルーチンの実行が終わったら、レジスタを復帰し、スタックポインタをフレームサイズ分だけ加えます。最後にjr (jalr)でメインルーチンに戻ります。この方法は、サブルーチン内で別のサブルーチンを呼ぶとその分spが下に下がり、リターンすると上がります。

実際に数値を入れた例

```
addi  x2,x2, -16   // framesize=16
sw x1, x2,12
sw x18,x2,8
sw x19,x2,4

….サブルーチン本体

lw  x1,x2,12
lw x18,x2,8
lw x19,x2,4
addi x2,x2,16
jr x1
```

x1: ra戻り値　x2: spスタックポインタ
jr x1はjalr x1,x0,0 (疑似命令）

実際に数値を入れた例を示します。スタックポインタはx2,戻り番地のレジスタにはx1を使います。ここではx18、x19、x1を保存/復帰しています。

例題3

- 掛け算のサブルーチン（x3とx4を掛けてx5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。

掛け算用サブルーチン

x3を破壊しないサブルーチンコール
x2はメインルーチンで初期化する必要がある

```
// Subroutine Mult  x5 ←  x3 × x4
mult:   addi x2,x2,-4   frameizeは4
        sw x3,x2,0      x3の値がスタックに退避される
        add x5,x0,x0
loop:   add x5,x5,x4
        addi x3,x3,-1
        bne x3, x0,loop
        lw x3,x2,0      x3をスタックからpop
        addi x2,x2,4    x2を元に戻す。
        jr x1           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x5は保存しません。x3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、x4同様、メインルーチンでも利用可能になります。

jal, jalrの実装

jal, jalrは共に独自のopcodeを持っている

```
assign jal_op = (opcode == `OP_JAL);
assign jalr_op = (opcode == `OP_JALR)& (funct3== 3'b000);
```

```
assign imm_j = {instr[31], instr[19:12],instr[20],instr[30:21],1'b0};
```

jal

イミーディエイト命令と同じ形式で、imm_iがそのまま利用可能　opcodeは違っている

jalr

ではjal, jalrの実装を示します。命令は両方とも独自のopcodeを持っています。これを、デコードするために、def.hでは`OP_JAL, `OP_JALRが定義されています。これを使ってjal_opとjalr_opを作ります。jalの方は20ビット分のオフセットフィールドを持つのですが、分岐命令同様、途中で分離、回転しているため、全体が掴みにくいです。最後に0を補って飛び先番地とします。これは今までとは独立したimm_jを設けます。一方、jalrは命令形式はイミーディエイト形式と同じですのでこれを利用できます。ただし、opcodeは独自の数値でfunct3は0になっています。なので、jalr_opdでこれを検出します。

jal,jalrの実装

ALUのB入力にはjalr,jal共にイミーディエイトを入れるが、拡張の仕方が違っている

```
assign srcb = imm_op | lw_op |jalr_op ? {sext, imm_i}:
    bra_op ? {sext[18:0],imm_b}:   sw_op ? {sext, imm_s}:
                                   jal_op ? {sext[10:0], imm_j}:  reg2;
```

jalは飛び先の演算

```
assign srca = bra_op | jal_op ? pcplus4: reg1;
```

jal, jalr共に戻り番地をレジスタへ

```
assign result = (jal_op| jalr_op) ? pcplus4 :
                                   lw_op ? readdata : aluresult ;
```

ALUは両者共に加算に設定

```
assign addcom = (lw_op | sw_op| bra_op| jal_op | jalr_op);
```

jal, jalr共に戻り番地を書き込む

```
assign rwe = lw_op | alu_op | imm_op | jal_op | jalr_op  ;
```

jalrはimm_iを使い、jalはimm_jを使います。jalは、飛び先を計算するためにALUのA入力にはPC+4、B入力にはimm_jを入れます。またレジスタに戻り番地をしまわなければならないです。一方、jalrは、ALUではレジスタの値にイミーディエイトを足したものが飛び先になるので、これをALUで計算させます。同様に戻り番地をしまわなければならないです。

PC周辺

```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if (jal_op |
          ( (beq_op & (reg1==reg2)) | (bne_op & (reg1!=reg2)) |
                (blt_op & (sreg1<sreg2) ) | (bge_op & (sreg1>=sreg2) ) |
                (bltu_op & (reg1<reg2) ) | (bgeu_op & (reg1>=reg2) ) )  )
         pc <= aluresult;
   else if(jalr_op)
          pc <= {aluresult[31:1],1'b0};
  else
    pc <= pcplus4;
end
```

jalはALUでの結果をそのまま使う

jalrでは0ビット目を強制的に0に

pcの設定には、jalは他の分岐命令同様、ALUでの計算結果をそのまま使います。
一方、jalrは最下位ビットに強制的に0を入れます。

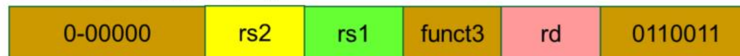
疑似命令

- j offset jal x0,offset
- jr rs jalr x0,rs,0
- beqz rs,offset beq rs,x0,offset
- bnez rs,offset bne rs,x0,offset
- bgt rs,rt,offset blt rt,rs,offset
- ble rs,rt,offset bge rt,rs,offset
- li rd,imm addi rd,x0,imm
- mv rd,rs addi rd,rs,0 add rd,rs,x0

RV32Iでは、直接その命令を持っておらず、他の命令により実現する場合があります。このような場合、アセンブラ上では、あたかもその命令が実際に存在するかのように扱います。この命令のことを疑似命令と呼ぶ。疑似命令はいちいち元の命令で書くよりも便利です。

レジスタ間演算命令

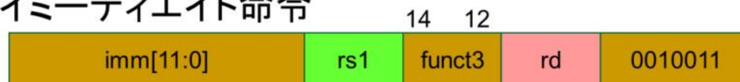
14 12



000	add,sub(30bit目で判別)	101	srl, srra(30bit目で判別)
001	sll	110	or
010	slt	111	and
011	srl		
100	xor		

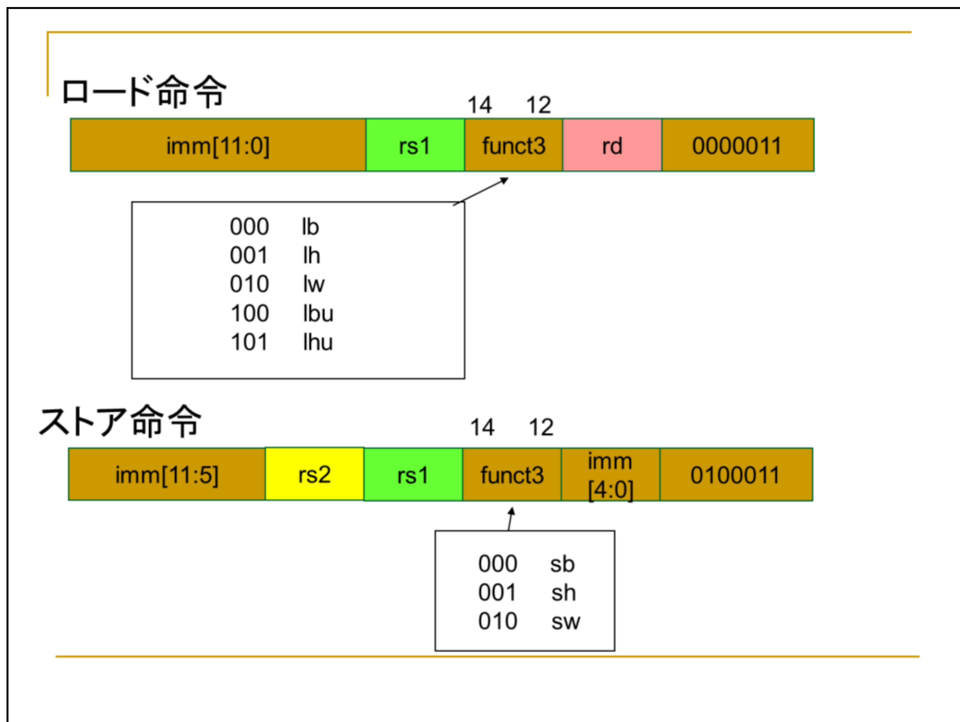
イミーディエイト命令

14 12

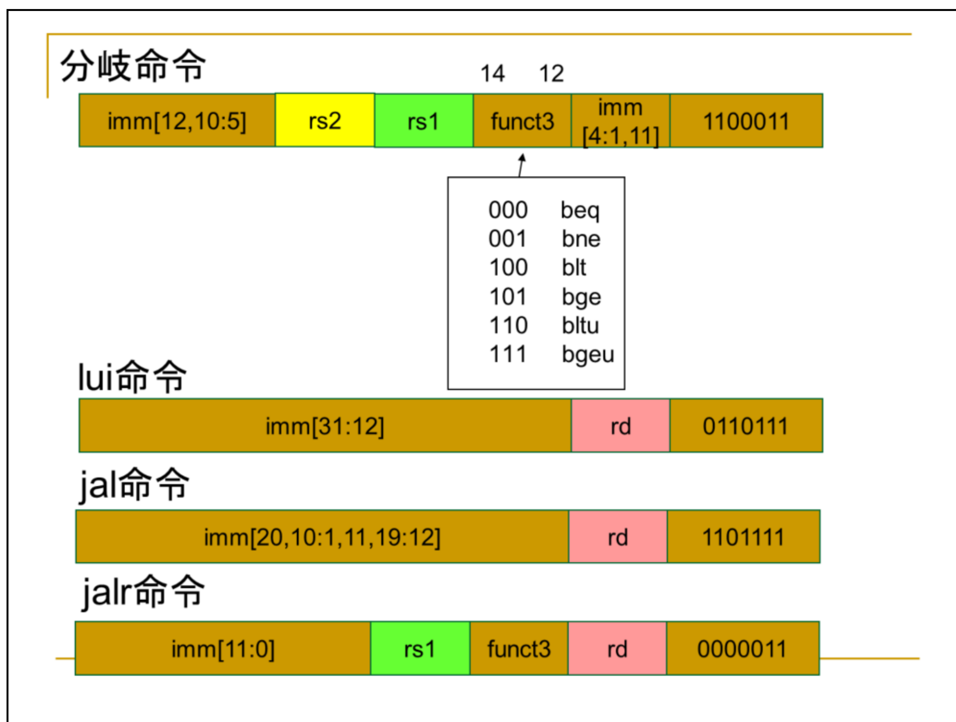


000	addi	101	srli, srrai (30bit目で判別)
001	slli	110	ori
010	slti	111	andi
011	srli		
100	xori		

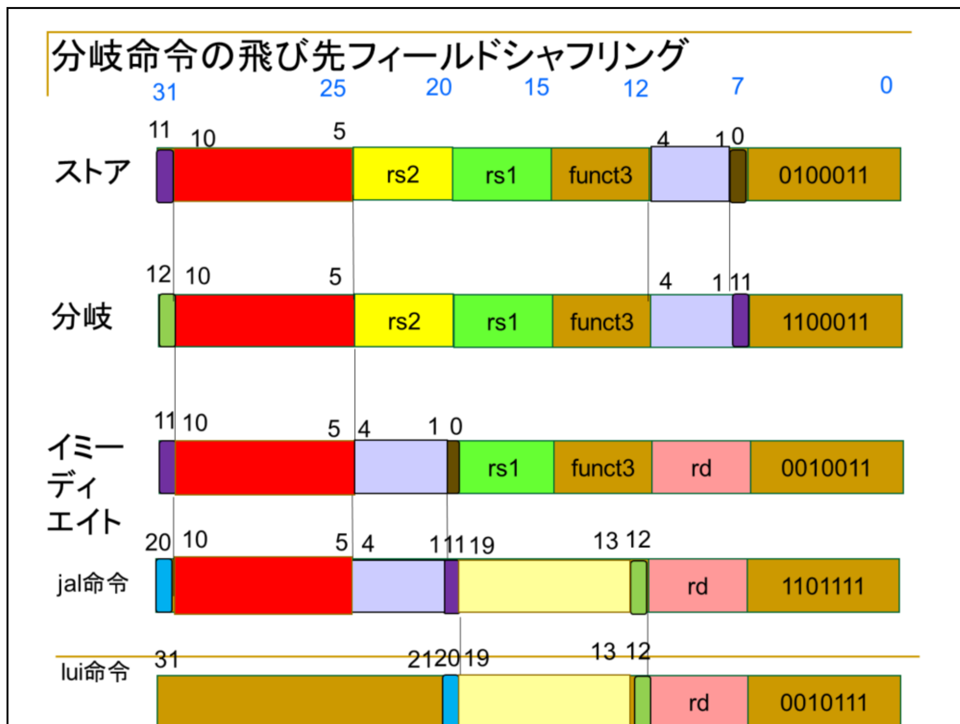
以下、今までの命令のまとめです。レジスタ間演算命令とイミーディエイト命令はfunct3の値と演算の種類が揃っています。イミーディエイト値は、シフト命令においては下位5ビットのみが有効で、30ビット目はsrlとsraの識別に使います。



ロード命令はイミーディエイト命令と形式が同じです。ストア命令はrs2の位置を他と揃えるために、イミーディエイト値が分割されています。



分岐命令は、飛び先を示すimmが分割され、ビット順が入れ替わっています。jal命令はビット順の入れ替わりのみ行われています。luiはjalと同じフィールドの配置ですが、bit順を入れ替えてはいません。jalrはイミーディエイト命令と同じ並びです。



前回の図の再掲です。jal命令のビット入れ替えも、他とフィールドを揃えるためです
。

auipc

lui命令と形式は同じ（opcodeが違う）

- immを左に12ビットシフトしてpcに加算し、結果をx[rd]に書き込む

- 飛び先に細工をした後jalrを使ってサブルーチンコール
- PCをレジスタに読み込む

- 他にもfence、csr、ebreak, ecallなど制御用の命令が用意されている

RV32Iは、他のISAにない命令を持っています。これがauipcで、lui命令と同じ形式で20ビットのイミーディエイト値を持ちます。これを12ビットシフトしてpcに加算し、結果をxdに書き込みます。この命令を使って飛び先をレジスタに入れて細工した後にjalrを使ってサブルーチンコールをしたり、PCをレジスタに読み込むために使います。他にもRV32Iは、システム管理用の命令をいくつか持っていますが、これは省略します。

RV32M 乗算への拡張

全てレジスタ間演算命令　例) mul rd,rs1,rs2

記号	操作
mul	x[rs1]とx[rs2]の乗算結果をx[rd]に入れる。溢れた分は無視
mulh	値を符号付き数値として扱い、x[rs1]とx[rs2]の乗算結果の上半分をx[rd]に入れる
mulhu	値を符号なし数値として扱い、x[rs1]にx[rs2]を掛け、積の上半分をx[rd]に書き込む
mulhsu	x[rs1]は2の補数、x[rs2]を符号なし数値として扱い、x[rs1]にx[rs2]を掛けて、積の上半分をx[rd]に書き込む

乗算と除算はRM32Iに含まれておらず、拡張命令セットRM32Mが定義されています。乗算は、二つ面倒なことがあります。一つは答えのサイズが増えることです。これに対応するため、上位半分をレジスタに入れる命令が装備されています。もう一つは、符号付きと符号なしを考えなければならない点です。これに対応してRM32Mは複数の乗算命令を用意してあります。特に興味深いのはmulhsuで片方は2の補数、もう片方は符号なし数として乗算します。

RV32M　除算への拡張

全てレジスタ間演算命令　例) div rd,rs1,rs2

記号	操作
div	値を2の補数値として扱い、x[rs1]とx[rs2]の商をゼロ方向に丸めてx[rd]に入れる。
divu	値を符号無し数値として扱い、x[rs1]とx[rs2]の商をゼロ方向に丸めてx[rd]に入れる
rem	値を2の補数値として扱い、x[rs1]とx[rs2]の余りをx[rd]に入れる。
remu	値を符号無し数値として扱い、x[rs1]とx[rs2]の余りをx[rd]に入れる

除算は、商を求める演算と、余りを求める演算を完全に分離した点に特徴があります。これも符号付き、符号無しがきっちり整備されています。

浮動小数点演算拡張RV32F, RV32D 基本命令

浮動小数点レジスタf0-f31が32本　RV32Dを装備する場合は、64ビットレジスタの下位32ビットを単精度で用いる。f0は普通のレジスタと同様に使える

記号	操作	形式
flw　fld	単精度データのロード	ロードと同じ
fsw fsd	単精度データのストア	ストアと同じ
fadd.s fadd.d	加算	レジスタ間演算と類似
fsub.s　fsub.d	減算	レジスタ間演算と類似
fmul.s fmul.d	乗算	レジスタ間演算と類似
fdiv.s　fdiv.d	除算	レジスタ間演算と類似
fmadd.s fmadd.d	積和演算	4オペランド
fmsub.s fmsub.d	積差演算	4オペランド
fmmadd.s fmmadd.d	積和で加算前に積の符号を反転	4オペランド
fmmsub.s fmmsub.d	積差で減算前に積の符号を反転	4オペランド

浮動小数点演算拡張は単精度用、倍精度用があり、両方を実装するのが普通です。この場合、64ビットの浮動小数点レジスタが32本用意されており、下位32ビットが単精度の場合に使われ、f0は普通のレジスタ同様に使うことができます。基本命令はIEEE標準規格に沿って行われます。これに加えて豊富は積和演算、積差演算命令が用意されています。

浮動小数点演算拡張RV32F, RV32D

記号	操作	形式
fsqrt.s fsqrt.d	平行根演算	1オペランド
fmax.s fmax.d	大きい方をf[rd]に	3オペランド
fmin.s fmin.d	小さい方をf[rd]に	3オペランド
fle.s fle.d	比較、f[rs1]<=f[rs2]の時x[rd]に1	3オペランド
flt.s flt.d	比較、f[rs1]<f[rs2]の時x[rd]に1	3オペランド
feq.s feq.d	比較、f[rs1]=f[rs2]の時x[rd]に1	2オペランド
fsgnj.s fsgnj.d 他2種類	符号を取ってインジェクトする レジスタ間転送に使える	2オペランド
fcvt.s.w fcvt.w.s fcvt.d.w fcvt.w.d 他数種類	形式変換、符号付き整数、符号無整数、単精度、倍精度間の変換	3オペランド

これに加えて平方根、最大、最小、比較がある。fsgnjは符号をインジェクトする命令ですが、実際はレジスタ間転送に使えます。各型式間の変換命令は全ての組み合わせについて用意されています。

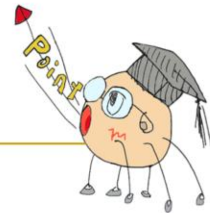
他の拡張命令セット

- RV32A: マルチコア用アトミック命令
- RV32C:短縮命令、命令コードの16ビット化、2オペランド、レジスタ数削減
- RV32V:ベクトル拡張
- RV64:64ビット化
  - RV64I、RV64M、RV64F,RV64D,RV64Aが定義されている。

ここでは省略しますが、RISC Vにはこの他にも様々な拡張命令セットが存在します。
これは基本命令セットを拡張する形で提供しているのが特徴です。

本日のまとめ

- RV32Iは32ビットのRISC、アドレス、データ共に32ビット
- 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
- jal,jalrは戻り番地を任意のレジスタに設定可能



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
- 提出物はmin.asm

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
- 提出物はsanjo.asm

5×5×5＝125（7D)になるはず

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