『はじめて読むMIPS(リローデッド)』　by 中森章

●はじめに

MIPSといえば，かつてはワークステーションに使用されているMPUという認識でしかなかったが，PlayStationやNintendo64などのゲーム機に採用されたり，組み込み制御向けのMPUとして脚光を浴びることが多くなった．一時期はハイエンドから組み込み分野まで全方位展開を行っていたMIPSの戦略は縮小された感があるが，現在では，ARMと並ぶ組み込み系のアーキテクチャとして，世間に認知されている．

その昔，アセンブリ言語でプログラムを書く人々にとってZ80とMC68000の命令セットを知っておくことが重要だったように，現在ではMIPSの命令セットも教養の一部であるといえるかもしれない．そこで，MIPSの命令に親しんでもらうべく，本稿ではMIPSの命令セットアーキテクチャについて解説してみたい．

●MIPS命令セットアーキテクチャ

MIPS RISCの命令セットは，MIPS IからMIPS Vまで，いくつかのカテゴリーに分類される(図1)．IからVへと数字が大きくなるにつれ(ユーザーモードでの)上位互換を有している．これらは，MIPS RISCがバージョンアップされる過程で，徐々に拡張されて来たものだ(表1)．また，その過程でMDMX(MIPS Digital Media Extension)やMIPS16という命令セットも派生している．

これらの命令セットは，現在では32ビット系のMIPS32と64ビット系のMIPS64の2種類に整理され，MIPS16など特定分野に特化した派生命令セットはASE(Application Specific Extension)として定義されている(図1)．従来は特権レジスタの形式などに互換性がなかったが，MIPS32とMIPS64では，ユーザーモードだけでなく，カーネルモード(特権命令)の互換性を取ることも目的としている．

最初のMIPS IはR2000/R3000のために設計された．MIPS命令セットの基本である．

MIPS IIはR6000のために設計された．倍精度の浮動小数点をロード／ストアする命令，Branch Likely(適当な日本語がない)命令，トラップ命令が新設された．現在の，多くのR3000の派生品は，このMIPS IIをサポートしている．

MIPS IIIは命令セットの64ビット拡張である．R4000のために設計された．

MIPS IVはハイエンドRISC向けである．主として浮動小数点命令の高機能化を重点に拡張されている．R5000，R8000，R10000に採用されている．

MIPS Vは３次元クラフィックなどの用途に単精度の浮動小数点演算を強化したものだ．ただし，MIPS Vをインプリメントしている MPUは現在のところ存在しない．

MDMXはビット長の短い整数を用いたベクトル演算を提供する．グラフィックス系の応用分野でピクセル単位での計算が可能になる．

MIPS16は16ビット長の命令を基本とする命令セットである．メモリ容量に制限がある組み込み制御分野向けである．

身近な例でいえば，PlayStationのMPUはMIPS I，Nintendo64のMPUであるR4300はMIPS III，Playstation2のEmotion EngineはMIPS IVのサブセットである．かつてWindowsCE機に搭載された東芝のTMPR3922はMIPS II，NECのVR41xxはMIPS IIIである．

上述のように，現在ではMIPS I～Vは廃止され，新たにMIPS32，MIPS64という命令セットが定義されている．MIPS32はMIPS IIを基本とし，MIPS IVやMIPS Vから，32ビットの範囲で，有用な命令を取り入れたものである．MIPS64はMIPS Vをそのまま流用したものである．まあ，命令自体が変更されるわけではないので，旧来のMIPS I～Vをサポートする

MPU に対しての影響はあまりない.

●命令セットアーキテクチャの特徴

MIPS の命令セットは RISC(Reduced Instruction Set Computer)であるから, MC68000 系などの高機能な命令やアドレッシングモードを持った MPU と比較すると, 命令の機能は単純化されている. その主な特徴は次のとおり.

・ロード／ストアアーキテクチャ
・3 オペランド命令
・ゼロレジスタと 31 本の汎用レジスタ
・遅延分岐

以下に詳しく見てみよう.

▲ロード／ストアアーキテクチャ

ロード／ストアアーキテクチャは RISC の最大の特徴である. 時間のかかるメモリへのアクセスはロード命令, ストア命令のみで行い, それ以外の演算はレジスタ間で行なう. メモリへのアドレッシングモードは１種類しかなく, ベースレジスタにオフセット(ディスプレースメント)を加えて実効アドレスを計算するという実に単純なものだ. メモリの内容に 1 加算する場合には MIPS では次のようになる.

```
        la      r2,mem     // mem のアドレスを r2 にロード.
        lw      r3,0(r2)   // r2 が指すメモリの内容を r3 にロード.
        addiu   r3,r3,1    // r3 に１を加え, r3 に格納.
        sw      r3,0(r2)   // r3 の値を r2 が指すメモリにストア.
        ……
mem:
      .word    0
```

▲3 オペランド命令

3 オペランド命令は MIPS の命令セットの最大の特徴である. ほとんどすべての演算は 3 つのレジスタ間で行われる. つまり, ソースレジスタ(rs)とターゲットレジスタ(rt)の内容を演算し, デスティネーションレジスタ(rd)に格納する.

多くの MPU では２オペランド方式が採用されており, ターゲットレジスタとデスティネーションレジスタが同じである. この場合, 入力となる値の片方を破壊することになり, プログラムの自由度が制限される. コンパイラでは, 式の最適化のため, 演算の内部表現が 3 オペランドになっていることが多いが, MIPS ではコンパイラとの相性を考慮して３オペランドになっているのかもしれない.

MIPS の命令がなぜ 3 オペランドになったのかは, R2000 をモデルとしている Hennessy と Patterson 著の『コンピュータアーキテクチャ』にも理由がないのではっきりとはわからないが, 考え方としては自然である.

さて, 加算命令について説明する.

```
      addu    r3,r4,r5
```

というのは,r4 の値と r5 の値を加算(Add Unsigned)して r3 に格納するという命令である．Unsigned(符号なし)というのは，入力が無符号整数として扱われるということである．加算の結果は符号があろうとなかろうと同一になるので，符号付(Signed)の add 命令との違いは，現象的にはオーバーフロー例外が発生するか否かの違いしかない．

C コンパイラでは原則的に，加減算でオーバーフローは発生しないことになっているので，add 命令よりも addu 命令が用いられる．入力の片方はレジスタでなく，即値(イミーディエト値)であることもある．この場合は，ソースレジスタの値とイミーディエト値を演算してターゲットレジスタに格納する．たとえば，

```
    addiu   r3,r4,0x1234
```

は，r4 の値にイミーディエト値の 0x1234 を加算(Add Immediate Unsigned)して r3 に格納する．命令長の関係から 1 つの命令で指定できるイミーディエト値は 16 ビット長に限られる．

▲ゼロレジスタと 31 本の汎用レジスタ

一般に，MIPS アーキテクチャは 32 本の汎用レジスタを持っているといわれる．しかし，これは厳密には正しくない．32 本のレジスタの内，0 番(r0)はゼロレジスタと呼ばれ，値が 0 に固定されている．値が 0 に固定されているほかは任意のオペランドとして使用できる(書き込みもできるが，あまり意味はない)ので，汎用レジスタのとみなしても構わないのだが，(0 以外の値の)データの保持ができない点で，汎用と呼ぶには抵抗がある．

それはともかく，ゼロレジスタは
どのような用途に使われるのだろう．もっとも頻度の高い使用法はゼロとの比較である．MIPS(の CPU)には条件フラグというものはなく，条件分岐は指定されたレジスタの値に従って分岐／不分岐が決定される．条件セット命令(これは MC680X0 にもある)で条件の成立をテストし結果(0 か 1)をレジスタにセットする．それとゼロとの比較を行って分岐判定をするわけだ．具体的には，

```
      li        r5,9        // r5 に 9 を格納.
loop:
       .
       .
       .
      bne       r5,r0,loop  // r5 の値が 0 でないとき分岐.
      addiu     r5,r5,-1    // r5 の値を 1 減じる．遅延スロット.
```

は，r5 の値が 9 から 0 までの間，10 回のループとなる．また，

```
loop:
       .
       .
       .
      slt    r5,r3,r4     // r3<r4 なら r5 が 1.
      bne    r5,r0,loop   // r5 の値が 0 でないとき分岐.
      nop                 // 遅延スロット．何もしない.
```

は，r3 の値が r4 の値よりも小さい間，分岐を繰り返す．ゼロレジスタは 0 という定数値を生成するためにも用いられる．たとえば，次の 2 つの命令を考える．

```
or      r3,r4,r0
addiu   r3,r0,0x123
```

最初の例は，r4 の値と r0 の値の論理和を r3 に格納する．いわば，r4 から r3 への転送命令である．

2 番目の例は r0 の値とイミーディエト値 0x123 を加算して r3 に格納する．これは r3 へのイミーディエト値のロード命令とみなせる．

汎用レジスタの内で特殊な役割をもっているのが r31 である．これは，関数の呼び出し命令(JAL=Jump And Link)によって，戻りアドレスが自動的に格納される．関数からは r31 のアドレスにジャンプすることで戻ってくる．具体的には，

```
    jal  Target
    .
    .
    .

Target:
    .
    .
    .
    jr   r31
```

という命令シーケンスになる．r31 は 1 本しかないので，関数の中でまた別の関数を呼び出す場合は，r31 の値をメモリ(多くの場合スタック)に退避する．たとえば，r29 がスタックポインタに割り当てられている場合

```
    jal  Target1
    .
    .
    .

Target1:
    addiu  r29,r29,-4
    sw     r31,0(r29)
    .
    .
    .
    jal    Target2
    .
    .
    .
```

```
lw      r31,0(r29)
jr      r31
addiu   r29,r29,4
```

といった使い方をする．

r0 と r31 以外は特殊な役割はなく，本当に汎用なレジスタである．アセンブリ言語でのプログラミングではこれらを自由に使用できる．ただし，C コンパイラなどではそれぞれのレジスタに役割分担がある(表 2)．コンパイラによるコードとリンクして動作するプログラムを作成するときには，これらを考慮する必要がある．

▲遅延分岐

遅延分岐はパイプラインを有効利用しようとする RISC の最大の特徴である(なんか，このフレーズばっか)．

通常，分岐先の命令フェッチは実行ステージでの分岐アドレスの計算または条件比較の後でなければ実行できない．つまり，命令の実行ステージから分岐先の命令フェッチまでに最低 1 クロックの空きが生じる．これが分岐遅延である．また，この空きを遅延スロットという．

従来の CISC ではこの部分に実行される命令を意図的に抹殺していた．最終的に，パイプラインの実行が分岐遅延の分だけ無駄になる．RISC では命令を抹殺する制御をする代わりに，遅延スロットの命令(分岐命令の直後の命令)を積極的に実行する．その分，パイプラインが有効に実行される．

さて，MIPS II では Branch Likely 命令が導入された．無理矢理，日本語に訳すと「分岐する傾向がある分岐命令」ということになろうか．これは，分岐命令が実際に分岐するときのみ遅延スロットの命令を実行するというものだ．分岐しない場合は，遅延スロットの命令は無視される．

この命令は，遅延スロットに分岐先の先頭の 1 命令を置く，という使い方をする．分岐する傾向があるのだから，このような命令配置にすることで，高い確率で分岐遅延の無駄をなくすことができる．たとえば，次のような命令シーケンス(ループ処理)を考える．

```
loop:
    addu    r4,r5,r6
    addu    r7,r8,r9
    ……
    beql    r2,r0,loop  // Branch Likely 命令
    nop                 // 遅延スロット
```

この命令シーケンスは次と等価である．

```
loop:
    addu    r7,r8,r9
    ……
    beql    r2,r0,loop  // Branch Likely 命令
    addu    r4,r5,r6    // 遅延スロット
```

つまり，Branch Likely 命令の遅延スロットに分岐先の 1 命令をもってくることで，1 回

のループ内で実行する命令を 1 命令削減できる．つまり，1 回のループ当り 1 クロックの実行時間の節約になる．

また，通常の分岐命令では，遅延スロットに置くのに適当な命令がない場合もある．そのような場合は NOP 命令を置いておくが，分岐遅延はそのまま残ってしまう．このような場合も，Branch Likely 命令を使えば，分岐先の 1 命令を遅延スロットに置くことができるので，分岐遅延を有効利用できる．つまり，

```
    beq   r2,r3,Label   // 通常の分岐
    nop
      ・
      ・
      ・
Label:
    addu  r4,r5,r6
    addu  r7,r8,r9
```

というような命令シーケンスは

```
    beql  r2,r3,Label  // Branch Likely
    addu  r4,r5,r6
      ・
      ・
      ・
    addu  r4,r5,r6
Label:
    addu  r7,r8,r9
```

と置き換えることができる．分岐しない場合の実行クロック数は同じだが，分岐する場合は 1 クロック得をする．

●CPU 命令セットの概要

CPU の命令長はすべて 32 ビットで，命令形式には図 2 に示す 3 種類がある．命令形式を 3 種と単純化することで，命令のデコードを簡略化し，高い周波数での実行を可能にする．複雑な命令機能やアドレッシングは複数の命令を用いて実現するようになっている．

実際には，CPU の命令は次の 6 つのクラスに分類できる．

(1)ロード／ストア命令
(2)演算命令
(3)ジャンプ／分岐命令
(4)特殊命令
(5)システム制御コプロセッサ(CP0)命令

このうち，(2)の演算命令は次の 4 つに分類される．

(1)イミーディエト命令

(2) 3 オペランド命令
(3)シフト命令
(4)乗除算命令

これらの命令の概要を表 3～表 7 に示す．また，CPU 命令のオペコードマップを図 3 に示す．
また，以下に MIPS で特徴的な命令を説明する．

▲サブルーチンコール

CISC でいうところの関数呼び出しやサブルーチンコール命令はスタックに戻りアドレスを退避してターゲットであるサブルーチンや関数にジャンプする．しかし，暗黙的にスタックというメモリ領域を使用する CISC 式のサブルーチンコール命令は，ロード／ストア命令を基本とする RISC とは相容れない．また，メモリのアクセス時間の遅さが命令実行に影響を与えてしまう．
そこで RISC では戻りアドレスをスタックではなく汎用レジスタに格納してターゲットにジャンプするのが一般的である．MIPS アーキテクチャでもこの方式を採用する．

jal (Jump And Link)

命令がそれで，戻りアドレス（遅延スロットがあるので JAL のアドレス＋8 番地）を汎用レジスタである r31 に格納してターゲットにジャンプする．
JAL 命令では戻りアドレス（リンクアドレスという）を格納する汎用レジスタは r31 に固定されているが，ターゲットアドレスを汎用レジスタで指定する

jalr (Jump And Link Register)

命令では，リンクアドレスの格納に r31 以外のレジスタを指定することも可能である．しかし，一般には r31 が使用される．
JAL 命令はサブルーチンに無条件に分岐する．MIPS アーキテクチャでは条件サブルーチンコールというべき命令も存在する．つまり，条件（レジスタの値が 0 以上か，0 より小か）が成立する場合のみに分岐する命令である．そして，JAL 命令と同様に戻りアドレスを r31 に格納する．これらについては後で条件分岐と同時に説明する．

▲条件分岐命令

CISC では条件分岐命令といえば，ゼロ，符号，キャリーなどの条件フラグを参照して分岐の成立／不成立を判断する．しかし，MIPS アーキテクチャでは，通常は，条件フラグというものを定義せず 2 つの汎用レジスタの値を比較してその大小関係で分岐の成立／不成立を判断する．
これは条件分岐命令の種類を削減するのが大きな理由ではないかと推測される．たとえば，条件フラグが 4 ビットあれば 16 通りの組み合わせが考えられ，その組み合わせの数だけ条件分岐命令が必要である．しかし，レジスタ同士の比較だと，＝，≠，≦，≧，＜，＞の６種の関係だけでこと足りる．
まあ，条件フラグを定義すると，汎用レジスタと同様に，パイプラインのステージ間でフォワーディング（値のバイパス）が必要になり，制御回路が複雑になるのを避ける意味もあるのかもしれない．また，条件フラグが存在しないことにより，命令のスケジューリ

ング(最適になるように並び替えること)が容易になる．これはコンパイラにとっても都合がいい．

(1)レジスタ比較による分岐

MIPS アーキテクチャで定義されている，レジスタ比較による条件分岐は

```
beq     (Branch on EQual)
bne     (Branch on Not Equal)
blez    (Branch on LEss than or Equal to Zero)
bgtz    (Branch on Greater Than Zero)
bltz    (Branch on Less Than Zero)
bgez    (Branch on Greater or Equal to Zero)
bltzal  (Branch on Less Than Zero And Link)
bgezal  (Branch on Greater or Equal to Zero And Link)
```

の 8 種である（正確には，これらすべてに Likely 分岐があるので 16 種）．その意味は，簡単な英語なので，動作はわかるだろう．

最後の 2 つはサブルーチンコール用の条件分岐である．beq/bne 以外は，1 つの汎用レジスタをオペランドとし，そのレジスタと r0（ゼロレジスタ）との暗黙的な比較で分岐条件を決定する．

2 つの汎用レジスタ間の大小比較で条件分岐する場合は条件セット命令と組み合わせて使用することが多い．条件セット命令は

```
slt     (Set on Less Than)
sltu    (Set on Less Than Unsigned)
```

の 2 種類であるが，組み合わせる条件分岐命令を beq/bne で使い分けることで逆の条件もテストできる．つまり，

```
slt   r2,r3,r4     // r3＜r4 なら r2 に 1 を格納
bne   r2,r0,target // r3＜r4 なら target に分岐
```

の逆の条件は

```
slt   r2,r3,r4     // r3＜r4 なら r2 に 1 を格納
beq   r2,r0,target // r3≧r4 なら target に分岐
```

でテストできる．なお，条件セット命令にはイミーディエト値との比較である

```
slti    (Set on Less Than Immediate)
sltiu   (Set on Less Than Immediate Unsigned)
```

も用意されている．

一方，MIPS16 モードの条件分岐は，1 つの汎用レジスタと 0 との比較である

```
beqz    (Branch on Equal to Zero)
bnez    (Branch on Not Equal to Zero)
bteqz   (Branch on T is Equal to Zero)
btnez   (Branch on T is Not Equal to Zero)
```

の 4 種しかない．Ｔというのは r24 の値のことで，r24 の値が 0 か否かで条件分岐する．

MIPS16 において 2 つの汎用レジスタ間の関係で条件分岐するには，減算命令か排他的論理和命令と組み合わせて使用する．MIPS16 では比較命令も用意されており，これは２つの汎用レジスタの排他的論理和を r24 に格納する．ただし，この場合，汎用レジスタの値の一致／不一致しかテストできない．値の大小をテストするには条件セット命令を使用する．条件セット命令は比較結果（0 か１か）を r24 に格納する．

つまり，MIPS16 では 2 オペランド命令が基本なので r24 という条件格納用のレジスタを使用するのである．これは，一種の条件フラグと言えなくもない．

(2)条件フラグによる分岐

MIPS アーキテクチャにおいてコプロセッサの条件判定に関しては条件フラグを使用する．コプロセッサの制御レジスタ内に最低１ビットの条件フラグ（条件コードという）を持ち，その値が０か１かで分岐の成立／不成立を決める．しかし，条件フラグに値をセットするのは比較命令のみ(直接ステータスレジスタに値を書き込んでも設定できるが)なので，条件フラグがあっても，コンパイラのスケジューリングの妨げにはなりにくい．

そのための条件分岐命令が

```
bczt    (Branch on Coprocessor Z is True)
bczf    (Branch on Coprocessor Z is False)
```

である（Likely 分岐も存在する）．ここで，ｚは 0 または１である．アーキテクチャ的にはｚの値として 2 も定義されるが，その挙動はインプリメント依存である．ｚが 0，つまりコプロセッサ 0（CP0）は MPU に内蔵されているシステム制御ユニット，ｚが 1，つまりコプロセッサ１（CP1）は FPU（浮動小数点演算ユニット）である．ｚが 2 のコプロセッサ 2（CP2）は，MPU の製造メーカがオリジナルな命令セットを定義するために予約されている．

CP0 の場合，条件コードはステータスレジスタのビット 18（CH ビット）である．このビットは R4000/R4400 以外では特別な意味はない．CH ビットは，ソフトウェアでリード／ライト可能である．その使い道はユーザー（というか OS）任せである．R4000/R4400 では最後に実行した 2 次キャッシュに対するキャッシュ命令がキャッシュのタグにヒットしたか否かを示す．使い道はよくわからない．MIPS32/MIPS64 では，CP0 の条件分岐は削除されてしまった．

CP1 の場合，条件コードは CP1 内のステータスレジスタ（FCR31）にある．通常は１ビット（ビット 23）であるが MIPS IV 以降では 8 ビットに拡張された．これらのビットは浮動小数点比較命令の比較結果が格納される．たとえば，

```
c.eq.s    fp0,fp2   // fp0 と fp1 の値が等しいとき条件コードが１
nop
bc1t      target    // 条件コードが１なら分岐
```

のように使用する．なお，古いプロセッサでは，bc1t/bc1f での条件コードのサンプリングは直前の命令の実行中に行われるので，比較命令と条件分岐命令の間には少なくとも無関係な 1 命令を挿入しなければならない．

▲乗除算命令

MIPS ではほとんどすべての命令を 1 クロックで処理することを目標としている．当然例外もある．この範疇に当てはまる命令は，浮動小数点演算と一部のシステム制御命令を除けば，乗除算命令がそれにあたる．

乗除算命令は，一般には，1 クロックで処理できない．これを通常のパイプラインに組み込むとパイプラインが乱れて性能低下につながる．これを回避するため，MIPS では乗除算を通常のパイプラインとは切り離し，他の演算と並列に処理するようになっている．このため，乗除算の出力（デスティネーションオペランド）として，汎用レジスタとは別の専用レジスタを用意している．こうすることで汎用レジスタとの依存性をなくする．

その専用レジスタが HI レジスタと LO レジスタである．32 ビット×32 ビットの乗算では積は 64 ビットであり，上位 32 ビットが HI レジスタに，下位 32 ビットが LO レジスタに格納される．同様に，64 ビット×64 ビットの乗算では積は 128 ビットであり，上位 64 ビットが HI レジスタに，下位 64 ビットが LO レジスタに格納される．

一方，32 ビット÷32 ビットの除算では 32 ビットの商が LO レジスタに，32 ビットの剰余が HI レジスタに格納される．64 ビット÷64 ビットの除算では 64 ビットの商が LO レジスタに，64 ビットの剰余が HI レジスタに格納される．プログラムでは，乗除算命令の後，数命令後に（乗除算の計算が終了したのを待って），HI レジスタまたは LO レジスタから結果を汎用レジスタに転送することになる．こうすると，パイプライン処理に乱れは生じない．HI レジスタの値を汎用レジスタに転送する命令が

```
mfhi  (Move From HI)
```

であり，LO レジスタの値を汎用レジスタに転送する命令が

```
mflo  (Move From LO)
```

である．今，32 ビット乗算に 3 クロック必要と仮定する．この場合は，

```
mult  r2,r3
nop
nop
mflo  r4   // r4 には r2 と r3 の積（下位）が転送される
```

というように，少なくとも 3 命令後で HI/LO レジスタを参照することが推奨される．nop の部分は乗除算に無関係な命令を入れてよい．

パイプラインを乱さないために，HI/LO レジスタへのアクセスは乗除算終了後に行なうことが推奨されてはいるが，これは強制ではない．プログラム的には乗除算命令の直後から mflo/mfhi を置くことも可能である．その場合は，パイプラインはインタロックし，乗除算の実行終了まで待ち合わせが生じる．

MIPS アーキテクチャではパイプラインをインタロックしないことが特徴である（特に

R2000/R3000 では）が，この場合は唯一の例外である．表 8 に各 MIPS RISC における乗除算処理の実行クロック数を示す．

MIPS アーキテクチャでは汎用レジスタから HI/LO レジスタに値を転送する命令もある．それが，

```
mtlo  (Move To LO)
mthi  (Move To HI)
```

である．プログラム上はこのような命令は不要である．しかし，HI/LO レジスタは，汎用レジスタと同じく，タスクを特定するコンテキストの一部なので，タスク切り替えの際に OS が使用する．

乗除算命令はデコード時にソースオペランドのリードと同時に実行が開始される．パイプライン的に乗除算命令の前 2 命令には mflo/mfhi/mtlo/mthi を置くことはできない．mflo/mfhi は乗除算の途中結果を参照する可能性があり，mtlo/mthi は乗除算の途中結果を破壊する可能性があるためである．もっとも，（特にアウトオブオーダーな）スーパースカラを採用すると 2 命令前という制限が無意味になるので，後期の MPU である R10000 などではこのような制限はない．

ところで，乗算の結果を特殊レジスタに格納する MIPS の方式は使いにくいのか，MIPS32/MIPS64 アーキテクチャではデスティネーションに汎用レジスタを指定できる乗算命令である MUL が導入された．MUL 命令は 3 つのオペランドを持ち，その 1 つがデスティネーションレジスタとなる．たとえば，

```
mul  r3,r1,r2   // r3 <- r1 × r2
```

のように記述される．ただ，不可解なのは MUL 命令は 32 ビット×32 ビットの乗算のみのサポートで，64 ビットの乗算は MIPS64 の範囲でも存在しない．MIPS が決めた命令にしてはエレガントさを欠いている（DMUL があってもいいはず）．

噂では MIPS32 を初めて実装する Jade を開発したのは LSI Logic 社の技術者であるという．このため，拡張命令は 32 ビットに特化されているという説がある．事実，LSI Logic 社の TinyRISC シリーズの IP コアでは MIPS32 の拡張命令である MADD/MADDU/MSUB/MSUBU/MUL 命令を早い段階で実装していた（CLZ/CLO 命令は未実装）．

▲r1 と擬似命令

プログラムが利用できる 32 本の汎用レジスタのうち，幾つかは専用レジスタとしての特別な意味を持っている．r0 はゼロレジスタで値が常にゼロとなるレジスタである．r31 はリンクレジスタでサブルーチンコール時の戻りアドレスが格納される．これらについては既に説明した．あと，r29（スタックポインタ），r28（グローバルポインタ）というものがあるが，これらはプログラムに関する決めごとであって，C 言語とかのリンクを考えない限りは自由に使用してよい．

以上は，既に説明したが，このようなレジスタの仲間である r1 は少し特別な意味を持っている．r1 の別名は at(assembler temporary)で，アセンブラのマクロ命令で一時的に使用されるレジスタである．これを頭に入れておかないと予期せずに r1 の値が破壊されてしまうことがある．実際，アセンブラで r1 を使用するとエラーまたは警告が出る．では，プログラムで r1 を使用してはいけないかというとそうでもない．通常は使用禁止である r1 もアセンブラの擬似命令で使用可能になる．

```
    .set  noat
```

というのがそれ（大抵のアセンブラはこのシンタックスである）で，この 1 行を挿入することで，その行以降で r1 が使用可能になる．というか，エラーや警告が出なくなる．プログラマが自覚して r1 を使用する分には何の問題もない．

しかし，誤って r1 を使用することを避けるために，逆の意味の擬似命令もある．それが

```
    .set  at
```

である．

▲マクロ命令

MIPS RISC の CPU 命令は表 3～表 7 に示されるとおりだが，感の鋭い人なら何かが足りないことに気付くだろう．これらの図の中には NOP 命令やメモリのアドレスをレジスタにロードする命令は含まれていない．その理由は，これらの命令はほかの命令や幾つかの命令の組み合わせで実現可能だからだ．RISC というポリシー上，冗長な命令はサポートされていないのだ．

しかし，メモリアドレスをロードする命令がないと，アセンブリ言語でプログラムを書く場合は不便である．そこで，大抵のアセンブラは，ユーザーが要求する命令で，命令セットに含まれない命令をマクロ命令(疑似命令)としてサポートしている．

マクロ命令で主なものを以下に説明する．(1)～(4)はほとんどすべてのアセンブラでサポートされている．(5)，(6)に関しては MIPS の純正アセンブラのみのサポートかもしれないので悪しからず．

(1)NOP(No Operation)

これは何もしない疑似命令である．NOP はデスティネーションオペランドを r0 にすれば実現できるので，いろいろなバリエーションが考えられる．たとえば，

```
    add  r0,r0,r0
```

が考えられる．実際には，命令コードが 0x00000000 になって切りがいい(？)

```
    sll  r0,r0,r0
```

が使用される．

(2)MOVE(Move)

これは，レジスタの値を一方から他方へ移動する疑似命令である．

```
    move  r3,r4
```

という記述は，通常，

```
    or     r3,r0,r4
```

と展開される．MIPS IV 以降では条件転送命令があるので，同じ動作は

```
    movz   r3,r4,r0
```

と記述することもできる．しかし，こちらは疑似命令ではなく，正真正銘 CPU の命令である．

(3)LI(Load Immediate)
　これは，レジスタにイミーディエト値をロードする疑似命令である．通常は，

```
    lui + ori
```

または

```
    lui + addiu
```

によって実現される．すなわち，32 ビットデータを上位 16 ビットと下位 16 ビットに分割して，2 回に分けてレジスタに格納する．命令コードのイミディエートフィールドが 16 ビットなのでそうなっている．たとえば，イミーディエト値 0x12345678 を r2 にロードする疑似命令である

```
    li    r2,0x12345678
```

は，

```
    lui   r2,0x1234
    ori   r2,r2,0x5678
```

または，

```
    lui     r2,0x1234
    addiu   r2,r2,0x5678
```

と展開される．イミーディエト値によっては lui，ori，addiu が単体で用いられる．たとえば，

```
    li    r2,0x80000000
```

は，lui 命令ではレジスタの下位 16 ビットに 0 が格納されることを考えれば，

```
    lui   r2,0x8000
```

で十分だし，

```
    li    r2,0x1234
```

は，値が 16 ビット長に収まるので，

```
    ori   r2,r0,0x1234
```

または，

```
    addiu  r2,r0,0x1234
```

で十分である．ここら辺の選択はアセンブラが最適になるようにやってくれるので，li がどのような命令列に展開されるのかは，ユーザーは特に知る必要はない．

(4)LA(Load Address)

これは，レジスタにメモリアドレスをロードする疑似命令である．実現方法は上の li と同じである．ただ，異なるのは，lui や ori のイミーディエト値を決定するのはリンカであるという点である．

```
    la     r3,mem
```

という記述に対して，アセンブラはとりあえず，

```
    lui    r3,0x0000
    ori    r3,r3,0x0000
```

または

```
    lui    r3,0x0000
    addiu  r3,r3,0x0000
```

というコードを生成しておき，0x0000 の部分にリンカが値を挿入してくれることを期待する．

(5)ULW(Unaligned Load Word)

これは，データタイプの値(今の場合，4 バイト)に整列されていないアドレスから１ワード(4 バイト)の値をレジスタにロードするための疑似命令である．MIPS アーキテクチャではデータタイプに整列されていないアドレスに対してロード／ストアを実行することは禁止されている(アドレスエラー例外が発生する)のだが，過去のプログラムとのしがらみなどから，例外的なアクセスを余儀なくされる場合に使用される疑似命令である．

これは，lwr 命令と lwl 命令で実現される．たとえば，

```
   ulw  r3,1(r2)
```

という記述は

```
lwl  r3,4(r2)   // リトルエンディアンの場合
lwr  r3,1(r2)
```

または,

```
lwl  r3,1(r2)   // ビッグエンディアンの場合
lwr  r3,4(r2)
```

と展開される.

(6)ROR/ROL(Rotate Right/Rotate Left)
これは, いわゆるローテート命令である. あまり実用性がないか, 従来は, 疑似命令となっていた. これらは, srl と sll によって実現される. たとえば

```
ror   r3,r2,5
rol   r3,r3,5
```

は, それぞれ,

```
srl   r1,r2,5
sll   r3,r2,(32-5)
or    r3,r3,r1
```

および,

```
sll   r1,r2,5
srl   r3,r2,(32-5)
or    r3,r3,r1
```

と展開される. ここでは, アセンブラ用のテンポラリレジスタである r1 が暗黙的に使用されている. 普段でも r1 はあまり使用しない方がいいということか.
ただ, MIPS アーキテクチャのリリース 2 ではローテート命令が実際の命令として定義された.

●コプロセッサ 0 関連の命令
MIPS アーキテクチャでは例外処理やメモリ管理などの特権機能を CPU とは別のコプロセッサ 0 というユニットで行なうようになっている. コプロセッサ 0 は特権レジスタ郡と MMU(TLB)から構成される. 特権レジスタ (CP0 レジスタ) へのアクセスは, CPU とのレジスタ間と, 専用命令

```
mfc0  (Move from Coprocessor 0)
mtc0  (Move to Coprocessor 0)
dmfc0 (Doubleword Move from Coprocessor 0)
```

```
dmtc0 (Doubleword Move to Coprocessor 0)
```

などで行われる．MPU を正常に動作させるためには，リセット後に適当な値を CP0 レジスタに設定する必要がある．

CP0 のレジスタ間との転送を行なう命令には，

```
cfc0 (Control From Coprocessor 0)
ctc0 (Control To Coprocessor 0)
```

が命令コード的には定義される．しかし，現在の命令セットには存在しない．MFC0/MTC0/DMFC0/DMTC0 は CP0 の汎用レジスタにアクセスする命令であるが，CFC0/CTC0 は CP0 の制御レジスタにアクセスする命令である．システム制御レジスタという，本来の意味からは，CFC0/CTC0 を使う方が正しいのだろうが，なぜか，MFC0/MTC0/DMFC0/DMTC0 が使用される．R2000/R3000 では，MFC0/MTC0 と CFC0/CTC0 は同じ動作をしていたらしい．

これらの命令のほかに条件分岐命令もあるが，これらは既に説明した．

▲NOP と擬似命令

MIPS のアセンブラでは不要な NOP 命令の挿入によって性能が低下するのを避けるためか，デフォルトでは NOP 命令の使用を禁止している．それでは分岐の遅延スロットに配置するべき命令がない場合や，ロード遅延（R3000 の場合）を調節するためにはどうするればいいのだろうか．

実は何もしなくてよい．MIPS のアセンブラは命令コードの順序が最適になるように命令を並び替える機能を持っている．そのときに自動的に NOP 命令が挿入されるので，プログラムでは遅延スロットやロード遅延を意識する必要はない．逆に，分岐命令の次に何か命令を置いても遅延スロットとはみなされないので，意図しない結果になる．

しかし，命令を勝手に並び替えてもらいたくない場合や，NOP 命令によってタイミング調整を行ないたい場合がある．このような場合は擬似命令によって並び替えを禁止することができる．それが

```
.set    noreorder
```

である．この擬似命令を指定した以後の行からは命令の並び替えが行われない．NOP 命令を使用することもできる．ただし，分岐の遅延スロットやロード遅延は自分で管理しなければならない．逆に，命令の並び替えや NOP 命令の挿入をアセンブラ任せにするには

```
.set    reorder
```

を指定する．本稿では，特に説明してないが，

```
.set  noat
.set  noreorder
```

が指定されているものとしてサンプルプログラムを紹介している．

●メモリ管理

CP0の役割のひとつが仮想記憶によるメモリ管理機能である.内蔵されているTLBにより,プログラムで参照する仮想アドレスを物理アドレスに変換する．もし，与えられた仮想アドレスがTLB内になかったり(TLBミス/TLB無効)や書き込み保護違反(TLB変更)の場合は例外が発生する.

通常のMPUではTLBミスが発生すると自動的にメモリ内のテーブルサーチを行い，新たなPTE(Page Table Entry)をリードしてきて，TLBを更新する．しかし，MIPS系のMPU(というかRISC MPUの多く)では単に例外を発生するだけである．TLBの入れ替えや更新はソフトウェアに任されている.

これは，MPU内にTLBのエントリ入れ替えのための複雑な制御回路を持たなくとも，例外処理ハンドラでのソフトウェア処理でも十分高速に実行できるいう目論見があるためである.

TLBB無効とTLB変更に関しては保護違反なので，通常のMPUでも例外になる．例外処理の性格が異なるため，TLBミスとそれ以外では例外ベクタも異なっている.

▲TLBの構成

R4000以降，MIPS系のMPUの多くは64ビットプロセッサであり，アドレス空間に関して32ビットモードと64ビットモードを持っている．TLBの各エントリも32ビットモードと64ビットモードで若干異なる.

図4にTLBエントリの形式を示す．各エントリはCP0レジスタである，エントリHi，エントリLo0，エントリLo1，ページマスクレジスタに対応する領域を持っている．TLBは，32ビットモードにおいては32ビットの仮想アドレスを，64ビットモードにおいては64ビットの仮想アドレス(TLBには40ビット分の領域しかないが)を，通常は36ビットの物理アドレスに変換する.

物理アドレスのビット数はMPUによって異なる．由緒正しいR4000系のMPUでは36ビットであり，R5000系のMPUもこれを継承している．その中間に開発されたR4200は33ビットという中途半端(？)なビット数である．後継のR4300はシステムバス幅が，従来の64ビットから32ビットに変更されたため，物理アドレスは32ビットしかサポートできない．ところが，R10000では物理アドレスが40ビットに拡張された．こうなると，エントリLo0/エントリLo1レジスタのビット長は32ビットでは不足するので，32ビットモード，64ビットモードにかかわらず，64ビット長のレジスタとなっている.

MMUのサポートするページサイズはエントリごとに4Kバイトから16Mバイトの範囲で4の倍数で指定できる．これはエントリへの書き込み時にページマスクレジスタで指定する．アドレス変換時に，仮想アドレス番号の下位ビットをページマスクレジスタの値で無視して仮想アドレスの検索が行なわれる.

TLBは48エントリ(R4200，R4300では32エントリ)のフルアソシアティブ構成で，1エントリは連続する2ページ分(偶数ページと奇数ページ)を示す1つの仮想アドレスと，それに対応する2つの物理アドレスを保持している．仮想アドレスはエントリHiレジスタ，偶数ページ，奇数ページに対応する物理アドレスは，それぞれ，エントリLo0レジスタ，エントリLo1レジスタで指定する．このTLB形態は一般にダブルエントリ形式と呼ばれている.

この形態は，48エントリではあるが，実質的には，98エントリ相当，あるいは指定したページサイズの2倍のページサイズを持っているとみなせるためTLBのヒット率が高くなるといわれている.

なお，エントリHiレジスタは仮想アドレスの他にタスク番号に対応するASID(Address Space ID)を指定できる．エントリLo0，エントリLo1レジスタは物理アドレスの他にキャ

ッシュ情報，保護情報を指定できる．

▲TLB の設定

TLB の設定に用いるレジスタは，エントリ Hi，エントリ Lo0，エントリ Lo1，ページマスク，インデクス(またはランダム)の 5 種類である．それぞれのレジスタ(インデクスとランダムレジスタを除く)の形式を図 5 に示す．TLB の各エントリの設定を行なうためには，上述のレジスタに値を入れ，

```
tlbwi (TLB Write by Index Register)
```

または

```
tlbwr (TLB Write by Random Register)
```

という命令を実行する．その名が示すとおり，TLBWI 命令はインデクスレジスタで示される TLB のエントリ，TLBWR 命令はランダムレジスタで示される TLB エントリへの書き込みを行なう．

ランダムレジスタは 0 から(TLB のエントリ数-1)の間の値を示すレジスタで，リセットで(TLB のエントリ数-1)の値に初期化され，以後，命令実行ごとにデクリメントされる．そして，その値がワイアードレジスタに保持されている値に等しくなったら，再び(TLB のエントリ数-1)の値になる．

TLBWR 命令で TLB エントリへの書き込みを行なう場合，番号が 0 から(ワイアードレジス-1)の間のエントリ(これをワイアードエントリという)には書き込みができない．TLBWI 命令ではワイアードエントリと無関係に書き込みができる．ワイアードエントリには，TLB の中から追い出されては困る，OS 内のアドレス変換情報などを保持しておく．

TLB エントリの設定の具体例を示そう．ページサイズを 4K バイトとし，仮想アドレス 0x2000(および，0x3000．ダブルエントリのため)に対応する物理アドレスを，偶数ページが 0x333000，奇数ページが 0x555000 としよう．この情報を TLB のエントリ 1 に入れるものとする．

このとき，ソフトウェアとしては(大体)次のようになる．ここで，asid，c，d，v，g，は適当な値を取るものとする．

```
mtc0    r0,C0_PageMask
li      r1,(0x2000|asid)
mtc0    r1,C0_EntryHi
li      r2,((0x33300>>6)|(c<<3)|(d<<2)|(v<<1)|g)
mtc0    r2,C0_EntryLo0
li      r3,((0x55500>>6)|(c<<3)|(d<<2)|(v<<1)|g)
mtc0    r3,C0_EntryLo1
li      r4,1
mtc0    r4,C0_Index
tlbwi
```

また，リセット直後は TLB の内容は不定である．このため，TLB を初期化しないとアドレス変換を正しく行なえない．TLB の初期化は，たとえば，次のようなプログラムで行なうこ

とができる.

```
    li      r2,0x80000000
    mtc0    r2,C0_EntryHi
    mtc0    r0,C0_EntryLo0
    mtc0    r0,C0_EntryLo1
    mtc0    r0,C0_PageMask
    li      r2,47
1:
    mtc0    r2,C0_Index
    addiu   r2,r2,-1
    bgez    r2,1b
    tlbwi
```

▲TLB の参照

TLB の各エントリの内容を参照することもできる. インデクスレジスタで指定されるエントリの内容を, ページマスク, エントリ Hi, エントリ Lo0, エントリ Lo1 レジスタに読み出す命令が

```
    tlbr  (TLB Read)
```

である. 逆に, 指定する仮想アドレスに対応する, 変換情報が格納されている TLB エントリの番号をインデクスレジスタに書き込む命令が

```
   tlbp   (TLB Probe)
```

である. 仮想アドレスはエントリ Hi レジスタで指定する. もし, 一致するエントリがなかった場合は, インデクスレジスタの最上位ビットがセットされる.

▲グローバルビット

TLB エントリの中には G(グローバル)ビットがある. このビットがセットされているとき, TLB のタグ部が与えられた仮想アドレスと一致するか比較する場合に ASID の一致を無視する. すべてのタスクが参照する OS 内のサービスルーチンやランタイムライブラリなどのアドレス領域を指定するのに使用する.

G ビットは TLB エントリの中には 1 ビットしかないが, エントリ Lo0 レジスタとエントリ Lo1 レジスタの両方に G ビットを設定する領域がある. 両方の G ビットの指定が 1 のときのみ, TLB エントリ内の G ビットがセットされる.

▲コンテキストレジスタ

アドレス変換において, 仮想アドレスと物理アドレスの対応表はメインメモリ内に置かれている. これをページテーブルと呼び, タスクごとに別々の組み合わせを持っている. MIPS 系の MPU においては, TLB ミスが発生した場合は, メインメモリをアクセスしてミスした仮想アドレスに対応する物理アドレスをページテーブルの中から読み出してきて, その情報を TLB に設定しなければならない. その場合に利用するレジスタがコンテキストレジスタである.

図 6 にコンテキストレジスタの形式を示す．コンテキストレジスタは 2 つの領域からなる．読み書き可能な PTEBase 領域と，読みだし専用の BadVPN2 領域である．PTEBase にはタスク切り替え時に，そのタスクのページテーブルのベースアドレスを設定する．

TLB ミスが発生したとき，その仮想アドレス(のページ番号を 2 で割った値=仮想アドレスのビット 31-13)が BadVPN2 領域に，自動的に設定される．このため，コンテキストレジスタは，ページサイズが 4K バイトの場合は，8 バイトのページテーブルエントリのペア(1 つの仮想アドレスに対して 2 つの物理アドレスが必要なため，合計で 16 バイト)を参照するポインタとしてそのまま使用できる．

ページサイズが 4K バイトでない場合は，このレジスタの値を適当にシフトしたりマスクしたりして，ページテーブルエントリへのポインタを生成することができる．

コンテキストレジスタは 32 ビットモードと 64 ビットモードの 2 つが定義されているが，アドレス空間が 64 ビットモードの場合は X コンテキストレジスタ(図 7)の方を使用する．

▲TLB 不一致処理の概略

TLB 例外（TLB ミス，TLB 無効，TLB 変更)が発生した場合，例外を発生した仮想アドレスは，エントリ Hi，BadVAddr，コンテキストレジスタに自動的に設定される．このとき，OS での処理は(大体)次のような感じになる．

```
    mfc0    r1,C0_Context
    lw      r2,0(r1)        // 偶数ページの物理アドレス
    lw      r3,8(r1)        // 奇数ページの物理アドレス
    tlbp                    // TLB 内にある仮想アドレス？
    mfc0    r1,C0_Index
    bltz    r1,NOTFOUND     // なければ分岐
    nop
    tlbr                    // あれば内容を読み出す

    適当な処理

    mtc0    r2,C0_EntryLo0  // 物理アドレス情報を更新
    mtc0    r3,C0_EntryLo1  // 物理アドレス情報を更新
    tlbwi                   // 同じエントリにライト
    j       COMMON
    nop
NOTFOUND:
    mtc0    r2,C0_EntryLo0  // 物理アドレス情報を新設
    mtc0    r3,C0_EntryLo1  // 物理アドレス情報を新設
    tlbwr                   // ランダムにライト
COMMON:
```

ここで，TLB ミスの場合は，例外ベクタがほかの場合(TLB 無効，TLB 変更)と異なり，専用のアドレスになるので，とくに TLBP 命令を使用せず直接 TLBWR 命令で TLB エントリを設定すればよい．

▲64 ビット命令の許可

カーネルモードにおいてはすべての命令を実行可能である．しかし，スーパーバイザーモードとユーザーモードにおいて，DADDU や DMULT のような 64 ビット命令を実行するためには，64 ビット命令の実行が許可されてなければならない．さもなければ，未定義命令として例外が発生する．

MIPS III や MIPS IV の方式では，アドレス空間が 64 ビットモードであれば，64 ビット命令の実行を許可するというものであった．つまり，スーパーバイザーモードではステータスレジスタの SX ビットが 1，ユーザーモードではステータスレジスタ UX ビットが 1 の場合に実行できる．

しかし，アドレス空間が 64 ビットであるという事象と 64 ビット命令が実行できるという事象は，本来は独立のはずである．事実，カーネルモードでは，ステータスレジスタの KX ビットの値にかかわらず，64 ビット命令が実行できた．

MIPS のアーキテクトもこの方式は美しくないと考えていたようである．MIPS32/MIPS64 のリリース 2 では，ステータスレジスタに PX ビットなるビットが新設された．このビットが 1 であれば，アドレス空間が 32 ビット，64 ビットにかかわらず，ユーザーモードで 64 ビット命令が実行できる．スーパーバイザーモードに関しては，カーネルモードと同様に，無条件に 64 ビット命令が実行できるようになった．

ステータスレジスタの形式を図 8 に示す．

●キャッシュ

▲キャッシュの構成

MIPS 系の MPU においてキャッシュの存在は必須である(M4K のようにキャッシュレスのプロセッサもあるが)．ただし，その構成については特に規定がない．たとえば，R3000/R4000 はダイレクトマップ，R5000/R10000 は 2 ウェイセットアソシアティブ，R20000(Ruby)は 4 ウェイセットアソシアティブである．MIPS 社の IP コアである 4K(Jade)や 5K(Opal)は 1〜4 ウェイセットアソシアティブを設計時にユーザーが選択できるようになっている．

1 次キャッシュのアドレスの比較方式は，R3000 が物理アドレスキャッシュ，それ以外が，仮想アドレスインデクス，物理タグ比較である．

2 次キャッシュに関しては，MPU に外付けのため，物理アドレスキャッシュである．R4000 では 2 次キャッシュを命令とデータ用に 2 分割できる仕組みを持っていた．しかし，この機能はバグが取り切れなかったため廃止になった（と記憶している）．以降，2 次キャッシュは命令とデータが混合したユニファイドキャッシュ構成 1 本に絞られた．

キャッシュの書き込み制御は R3000 がライトスルー，それ以外が，基本的にライトバック方式である．R5000 や R5400 など，TLB の C 領域でライトスルーを指定できるものもあるが，あまり利用されていない．

実装されるキャッシュ容量に関してはコンフィグレジスタ(図 9)の IC 領域，DC 領域でソフトウェアから参照できる．この他にもコンフィグレジスタは MPU を性格付ける様々な情報を参照したり設定するために用いられる．コンフィグレジスタの形式は MPU ごとに異なり，R40000 系，R5000 系の MPU では似たような形式をしているのがわかる．R10000 系に関しては内容がかなり異なるので図 9 では示していない．

従来はコンフィグレジスタの形式はまちまちだったが，MIPS32/MIPS64 ではコンフィグレジスタ(セレクト 1)でキャッシュの容量，ウェイ数，ラインサイズを知ることができるようになっている．図 10 に MIPS32/MIPS64 のコンフィグレジスタを示す．

▲キャッシュ命令

MIPS 系の MPU ではキャッシュを操作するためにキャッシュ命令が命令セットとして定義

されている．キャッシュ命令はキャッシュのラインを無効化したり，ダーティなラインをメインメモリへ強制的にライトバックしたりできる．キャッシュ命令の形式は

```
cache    機能コード,offset(base)
```

となっている．baseで示されるベースレジスタの内容にoffsetの値を加えて生成される仮想アドレスによって，キャッシュを参照する．機能コードの種類はMPUによって異なる．表9にR4000系のMPUのキャッシュ命令で使用される機能コードを示す．機能コードは5ビットからなり，下位2ビットが操作対象になるキャッシュの種類，上位3ビットが操作の種類を示す．キャッシュの種類は次のようになっている．

0 … （Ｉ） １次命令キャッシュ
1 … （Ｄ） １次データキャッシュ
2 … （ＳＩ）２次命令キャッシュ
3 … （ＳＤ）２次データキャッシュ

なお，２次キャッシュはユニファイドキャッシュとして構成されるので，SIを指定することはほとんどなく，大抵はSDを指定してキャッシュの操作を行なう．

▲キャッシュの初期化

MIPS系のMPUの特徴として，キャッシュは自動的には初期化されないということを挙げることができる．つまり，リセット後にキャッシュをキャッシュ命令を使って初期化しなければ，キャッシュを使用することができない．リセットで自動的にキャッシュが初期化されるという，ほかのMPUの経験がある人には奇異に思えるかもしれないが，そこはRISCということで納得してもらおう．

ここでは１次キャッシュを初期化する手順について述べる．2次キャッシュを有するMPUでは当然２次キャッシュの初期化も行なわなければならないが，手順は1次キャッシュと同様なので省略する．

キャッシュの初期化とは全ラインのタグ部を無効化することである．そのためにはキャッシュ命令を用いて各ラインのタグ部に0を書き込めばよい．前準備として，まずキャッシュの容量とラインサイズを知る必要がある．

古い形式では，コンフィグレジスタのCS(ビット12)，IC(ビット11-9)，DC(ビット8-6)，IB(ビット5)，DB(ビット4)領域を参照すればよい．CSはキャッシュ容量が4Kバイトよりも小さなMPUのためにNECのR4100シリーズで新設されたビットである．R4100シリーズではCSの値は1固定となっている．他のR4000系のMPUでは0である(R5000系はCSの位置に別のビットが割り当てられている)．

CSの値が0ならキャッシュ容量は，命令キャッシュが2の(12+IC)乗バイト，データキャッシュが2の(12+DC)乗バイトである．CSの値が1ならキャッシュ容量は，命令キャッシュが2の(10+IC)乗バイト，データキャッシュが2の(10+DC)乗バイトである．

IB，DBに関しては0のときが4ワード(16バイト)のラインサイズ，1のときが8ワード(32バイト)のラインサイズである．これらを考慮すると，R4000系における1次キャッシュの初期化は次のような命令列で行なえる．

```
mtc0    r0,C0_TagHi
mtc0    r0,C0_TagLo
```

```
    mfc0   r1,C0_Config
    srl    r2,r1,9
    andi   r2,r2,0x7     // IC 領域
    srl    r3,r1,6
    andi   r3,r3,0x7     // DC 領域
    li     r4,16
    andi   r9,r1,0x20    // IB 領域
    bnel   r9,r0,1f      // Brach Likely
    addu   r4,r4,r4      // IB が 0 ならスキップ
1:
    li     r5,16
    andi   r9,r1,0x10    // DB 領域
    bnel   r9,r0,2f      // Brach Likely
    addu   r5,r5,r5      // DB が 0 ならスキップ
2:
    li     r4,12
    andi   r1,r1,0x1000  // CS ビット
    bnel   r1,r0,3f      // Brach Likely
    li     r4,10         // CS が 0 ならスキップ
3:
    li     r1,1
    sllv   r6,r1,r2      // 命令キャッシュの容量
    sllv   r7,r1,r3      // データキャッシュの容量
    li     r1,0x80000000 // キャッシュ領域の開始アドレス
    addu   r2,r1,r6      // キャッシュ領域の終了アドレス
4:
    cache  Index_Store_Tag_I,0(r1)
    addu   r1,r1,r4      // ラインサイズを加算
    bne    r1,r2,4b
    nop
    li     r1,0x80000000 // キャッシュ領域の開始アドレス
    addu   r2,r1,r7      // キャッシュ領域の終了アドレス
5:
    cache  Index_Store_Tag_D,0(r1)
    addiu  r1,r1,r5      // ラインサイズを加算
    bne    r1,r2,5b
    nop
```

上述の命令列は，MPU 依存なので，すべての場合に当てはまるわけではない．MIPS32/MIPS64 で定義されたコンフィグレジスタを使用したキャッシュの初期化は次のようになる．

```
    mtc0   r0,C0_TagHi
    mtc0   r0,C0_TagLo
    mfc0   r1,C0_Config,1 // Sel 1
```

```
    srl     r2,r1,22
    andi    r2,r2,0x7       // IS 領域 (Entries/Way)
    li      r4,0x40
    sllv    r2,r4,r2
    srl     r3,r1,13
    andi    r3,r3,0x7       // DS 領域 (Entries/Way)
    sllv    r3,r4,r3
    srl     r2,r1,19
    andi    r2,r2,0x7       // IL 領域
    li      r4,2
    sllv    r4,r4,r2        // IC Bytes/Line
    srl     r2,r1,10
    andi    r2,r2,0x7       // DL 領域
    li      r5,2
    sllv    r5,r5,r2        // DC Bytes/Line
    srl     r6,r1,16
    andi    r6,r6,0x7       // IA 領域 (Way-1)
    srl     r7,r1,7
    andi    r7,r7,0x7       // DA 領域 (Way-1)
// 命令キャッシュの初期化
    li      r1,0x80000000 // キャッシュ領域の開始アドレス
1:
    addiu   r9,r2,-1        // IC Entries-1
2:
    cache   Index_Store_Tag_I,0(r1)
    addu    r1,r1,r4        // ラインサイズを加算
    bne     r9,r0,2b
    addiu   r9,r9,-1
    bne     r6,r0,1b
    addiu   r6,r6,-1
// データキャッシュの初期化
    li      r1,0x80000000 // キャッシュ領域の開始アドレス
1:
    addiu   r9,r3,-1        // DC Entries-1
2:
    cache   Index_Store_Tag_D,0(r1)
    addu    r1,r1,r5        // ラインサイズを加算
    bne     r9,r0,2b
    addiu   r9,r9,-1
    bne     r7,r0,1b
    addiu   r7,r7,-1
```

ここまでに示したプログラムは，キャッシュのタグを無効化する処理しか行っていない．キャッシュにパリティや ECC を有する場合は，それらのビットも初期化する必要があるが，ここでは省略する．

▲インデクスによるウェイの指定方法

キャッシュ命令で，キャッシュのインデクスを指定して処理を行なう場合，ダイレクトマップ方式のキャッシュでは，指定した仮想アドレスの 1 部がそのままインデクスとして用いられる．しかし，2 ウェイ以上のセットアソシアティブ方式のキャッシュにおいては，インデクスに加え，ウェイを指定しなければ，キャッシュラインが一意に定まらない．このウェイの指定には 2 通りの方法がある．

(1)R5000 方式

キャッシュ容量が 2 の n 乗バイトであるとき，仮想アドレスのビット(n-1)の値でウェイを指定する．R5000 ではキャッシュ容量が 32K バイト(n=15)であるから，仮想アドレスのビット 14 がウェイの指定になる．この方式だと，キャッシュ構成が 2 ウェイになったからといって，ダイレクトマップ用から，キャッシュの初期化のプログラムを変更する必要はない．同じプログラムで初期化を行なえる．

この方式の欠点としては，たとえば

```
cache    Index_Store_Tag_D,0x4000(r10)
```

などによって，ウェイ 1 を指定したつもりでも，r10 が示すベースアドレスのビット 14 の値が 1 であれば，オフセットとベースアドレスの加算によって生成される実効アドレスのビット 14 は 0 になってしまうので，ウェイ 0 の方を指定したことになることである．つまり，注意しないと，意図と違うウェイを指定してしまう恐れがある．

また，ベースアドレスが，アドレス範囲の境界を示しているような場合は，0x4000 というオフセットを加算することで桁の繰り上がりが発生し，アドレスエラーを引き起こすような実効アドレスが生成されてしまうこともある．これらの状況を避けるためには，ベースアドレスのビット 14 の値をあらかじめ 0 にしておく，などのプログラム処理が必要になる．

なお，MIPS32/MIPS64 で採用されたウェイの指定方式は，この R5000 方式と同じである．

(2)R10000 方式

指定する仮想アドレスの最下位ビットでウェイを指定する．たとえば，

```
cache    Index_Store_Tag_D,0x1(r10)
```

によって，確実にウェイ 1 を指定できる．なぜなら，キャッシュ命令に与えられる仮想アドレスは 4 の倍数であることが保障されているので,少なくとも下位 2 ビットは 0 であり，1 を加算しても桁の繰り上がりは発生しないためである．

欠点としては，キャッシュの初期化でウェイ 0 と 1 を別個に指定するプログラムを書かなければならないくらいで点であろうか．また，仮想アドレスの空きが 2 ビットしかないので 4 ウェイまでにしか対応できない．

●コプロセッサ 1(FPU)の命令セットの概要

FPU は MIPS の MPU の中ではコプロセッサ 1(CP1)として実装されている．R2000, R3000 では R2010, R3010 という別チップで供給されていたが, R4000 以降は 1 チップに内蔵された．組み込み制御用に FPU をサポートしない MPU も存在する．それらに外付けの FPU は存在し

ない(多分)．組み込み用途では浮動小数点演算は不要という神話が根強く生き残っているためと思われる．実際，ほとんどのことは固定小数点(整数)で代替できてしまうので，本当のことかもしれないが．しかし，３次元グラフィックスは浮動小数点でないと不便だろう．

浮動小数点演算はコプロセッサ 0 のステータスレジスタの CU1 ビットがセットされているときのみ実行可能である．MIPS IV 命令に関しては，CU3 ビット(XX ビットとか MIPS4 ビットとも呼ばれる)もセットされている必要がある．ただし，MIPS32/MIPS64 では，MIPS IV 命令とそれ以外という区別がなくなったので，CU3 ビットは意味がなくなった．

それはともかく，MIPS の FPU 命令セットについて説明する．CPU と同様に，FPU にも 3 種類の基本形式がある(図 11)．

・I-Type(イミーディエト形式)はロード／ストア命令などに使用される．
・R-Type(レジスタ形式)は２または３オペランドの演算命令に使用される．
・その他は分岐命令や転送命令に使用される．
　M-Type と記述している文献もある．

なお，浮動小数点のデータ形式は実質的な標準である IEEE-754 の 2 進表現に準拠している. FPU 命令のオペコードマップを図 12 に示す. また, FPU 命令の概要を表 10-1 から表 10-4 に示す.

▲レジスタセット

FPU のレジスタセットは 32 本の 64 ビット長の汎用レジスタと 2 本の制御レジスタで構成される．制御レジスタはステータスレジスタ（FCR31/FCSR)とプロセッサのリビジョンレジスタ（FCR0/FIR)が定義されているのみである．ただし，FCR31 は，条件ビット，例外発生フラグ，例外許可フラグの部分を仮想的なレジスタを通して部分的にアクセスできる．これらは，それぞれ，FCR25(FCCR)，FCR26(FEXR)，FCR28(FENR)に割り当てられている．

汎用レジスタ（FGR)へのアクセスは，CPU の汎用レジスタ間と，専用命令

```
mfc1  (Move from Coprocessor 1)
mtc1  (Move to Coprocessor 1)
dmfc1 (Doubleword Move from Coprocessor 1)
dmtc1 (Doubleword Move to Coprocessor 1)
```

を使用して行なえる．汎用レジスタはメモリ間とデータを直接ロード／ストアすることも可能である．そのための命令が，

```
lwc1  (Load Word to Coprocessor 1)
ldc1  (Load Doubleword to Coprocessor 1)
swc1  (Store Word from Coprocessor 1)
sdc1  (Store Doubleword from Coprocessor 1)
lwxc1 (Load Word Indexed to Coprocessor 1)         [MIPS-IV]
ldxc1 (Load Doubleword Indexed to Coprocessor 1)   [MIPS-IV]
swxc1 (Store Word Indexed from Coprocessor 1)      [MIPS-IV]
sdxc1 (Store Doubleword Indexed from Coprocessor 1)[MIPS-IV]
```

である．制御レジスタ（FCR)と CPU レジスタ間のデータ転送には

```
cfc1  (Move Control Word from Coprocessor 1)
ctc1  (Move Control Word to Coprocessor 1)
```

という命令を使用する．

浮動小数点データは単精度(32 ビット)と倍精度(64 ビット)をサポートする．ステータスレジスタ(図 8)の FR ビットが 0 の場合，浮動小数点レジスタ（FPR)は 16 本である．連続する偶数番号と奇数番号の汎用レジスタ（FGR)の下位 32 ビットを組にして 64 ビットの浮動小数点レジスタとみなす．つまり，

```
FPR[2n] = {FGR[2n+1], FGR[2n]}
```

という関係である．

これは CPU の汎用レジスタ長が 32 ビットだった R2000/R3000 での仕様の名残で，コプロセッサ 1 との間のデータ転送を 32 ビット単位でしか行なえなかったためである．R4000 以降も互換性を保っているわけだ．

ただし，R4000 以降は 64 ビットプロセッサになったため，それに対応して 64 ビット単位のデータ転送が可能になった．ステータスレジスタの FR ビットが 1 の場合，浮動小数点レジスタは 32 本となる．つまり，

```
FPR[n] = FGR[n]
```

という関係である．

浮動小数点レジスタは，単精度または倍精度の浮動小数点データを保持する．FR ビットが 0 の場合は偶数番号のレジスタのみが指定可能である．FR ビットが 1 の場合はすべてのレジスタが使用可能である．

FR ビットが 0 の場合，多くの実装では，奇数番号のレジスタを指定した場合は(奇数番号-1)で示される 64 ビットレジスタの上位 32 ビットを示す．しかし，MIPS32/MIPS64 のリリース 2 では，

```
mfhc1   (Move Word From High Half of Floating Point Register)
mthc1   (Move Word To High Half of Floating Point Register)
```

という，上位 32 ビットをアクセスする命令が定義された．この場合，レジスタとして奇数番号を指定した場合の挙動は不定である．FR ビットが 1 の場合は，任意の 64 ビットレジスタの上位 32 ビットにアクセスできる(あまり，使い道を考えられないが)．

▲ロード／ストア命令

ロード／ストア命令はメモリ上にある浮動小数点データを FPU レジスタにロードしたり，FPU レジスタのデータをメモリにストアするのに用いられる．通常はベースとなる CPU レジスタにオフセット(ディスプレースメント)を加算してメモリのアドレスを指定するが，MIPS IV ではオフセットの変わりに CPU レジスタ(インデクスとして利用)も指定できるようになった．ロード／ストア命令のレパートリーは上述の通りである．以下に使用例を示す．

```
lwc1    fp2,0(r10)
swc1    fp2,0(r10)
```

は 32 ビット長のデータのロード／ストア，

```
ldc1    fp2,0(r10)
sdc1    fp2,0(r10)
```

は 64 ビット長のデータのロード／ストアである．lwc1/swc1 については，R2000 時代からのしがらみで，64 ビットの浮動小数点データを上下 32 ビットずつに分離してロード／ストアすることも可能であるが，話がややこしくなるのでここでは説明しない．また，

```
lwxc1   fp2,r4(r10)
swxc1   fp2,r4(r10)
ldxc1   fp2,r4(r10)
sdxc1   fp2,r4(r10)
```

は r4 をインデクスとするロード／ストア命令の例である．これはテーブルや行列要素のアクセスを高速化する．

▲演算命令

加減乗除，平方根，逆数，積和演算がある．基本的に 3 オペランド演算で，転送，平方根，変換，比較は 2 オペランド，積和演算は 4 オペランドである．

▲比較と条件分岐，および条件転送

FPU には条件フラグともいえる条件ビット(cc)がある．MIPS III までは，この条件ビットは 1 ビットであったが，MIPS IV 以降では 8 ビットに増加された．FPU の条件分岐命令は，この条件ビットが 1 であるか 0 であるかを判定して分岐する．たとえば，

```
c.eq.s  fp2,fp4   // fp2 と fp4 が等しいか比較．条件ビットは cc0.
bc1t    target    // 条件ビット(cc0)が真(1)なら target へ分岐.
```

という命令シーケンスになる．条件ビットとして cc0 以外を使用する場合は，

```
c.eq.s  cc3,fp2,fp4
bc1t    cc3,target
```

などと記述する．浮動小数点の比較の種類を表 11 に示す．表 11 で Unordered とは，片方，または両方のオペランドが非数や無限大だったりして，「比較不能」という関係を表す．

条件転送は，パイプラインの流れを乱す分岐命令を削減するために，MIPS IV 以降で導入された．条件が真のときに FPU レジスタ間の転送を行なう．条件の指定方式としては，CPU レジスタが 0 であるか 0 でないか，浮動小数点の条件ビットが真(1)であるか偽(0)であるかの４とおりがある．例としては，

```
movz.s  fp1,fp2,r3  // r3 が 0 のとき転送
movn.s  fp1,fp2,r3  // r3 が 0 でないとき転送
movt.s  fp1,fp2,cc3 // cc3 が真のとき転送
movf.s  fp1,fp2,cc3 // cc3 が偽のとき転送
```

などがある.

▲積和演算

MIPS IV 以降は浮動小数点演算の性能向上を目的としている．その最大の目玉はインデクス付きロード／ストア命令と積和演算である．インデクス付きロード／ストア命令の例は既に説明した．積和演算は，MIPS にはめずらしく，４オペランドを取る．通常の FPU 命令ではフォーマット(fmt)の部分がレジスタ指定(fr)となっているので，機能フィールド(func)でフォーマットまでを指定している．

```
madd.s   fd,fr,fs,ft
msub.s   fd,fr,fs,ft
```

は，それぞれ，

$$fd \leftarrow (fs \times ft) + fr$$

および

$$fd \leftarrow (fs \times ft) - fr$$

という演算(単精度)を実行する．

●CP2 と CP3

MIPS アーキテクチャでは CP0 はシステム制御コプロセッサ，CP1 は浮動小数点演算ユニット(FPU)であることは既に述べた．しかし，アーキテクチャでは CP2 と CP3 も定義されている．厳密に言うと，R2000/R3000/R4000 では CP2 と CP3 が定義されていたが，R5000/R10000(MIPS IV)以降では CP3 の命令コードは COP1X(拡張 FPU)に割り当てられたので，CP2 のみが残っている．

さて，この CP2 とか CP3 とはどういうコプロセッサであろうか．実はこれといった規定はない．一応

```
COP2   (命令コードのビット 31-26 が 010010 であるものすべて)
mfc2   (Move From Coprocessor 2)
mtc2   (Move To Coprocessor 2)
dmfc2  (Move Doubleword From Coprocessor 2)
dmtc2  (Move Doubleword To Coprocessor 2)
cfc2   (move Control From Coprocessor 2)
ctc2   (move Control To Coprocessor 2)
bc2t   (Branch on Coprocessor 2 is True)
bc2f   (Branch on Coprocessor 2 is False)
bc2tl  (Branch on Coprocessor 2 is True Likely)
bc2fl  (Branch on Coprocessor 2 is False Likely)
```

という命令コードは定義されているので，これを利用して各メーカが独自にオリジナルな

コプロセッサを実装できる．たとえば，PlayStation2 に使用されている EmotionEngine では CP2 にベクタ演算ユニットを割り当てている．R5432 では CP2 に SIMD タイプのマルチメディア命令を割り当てている．これ以外に CP2 や CP3 を積極的に活用している例は知らない．大抵の MPU では CP2 や CP3 の命令コードを実行しようとしたときの挙動は未定義である．多くの場合，コプロセッサ使用不可例外ではなく，予約済み命令例外が発生するはずである．

さて，R2000/R3000 には CpCond という外部端子が 4 本あり，それぞれ CP0 から CP3 の状態に対応している．CpCond[0]や CpCond[1]の値は bczt/bczf(ｚ＝0,1)命令でソフトウェアから参照できる(CP1 である FPU は外付けであったことに注意).特に CpCond[0]は bc0t/bc0f 命令と組み合わせて外部回路と同期をとるために使用できる．それでは，CpCond[2]や CpCond[3]は何かというと，実はマルチプロセッサストールのために割り当てられていた．マルチプロセッサストールとは，マルチプロセッサ構成が可能なように，MPU の外部からアドレスを指定してデータキャッシュの値をリードしたり無効化するための仕組み(いわゆるバススヌープ)である．具体的には，CpCond[3]はパイプラインを強制的にストールさせるために使用し，CpCond[2]はそのときの操作(リードか無効化か)を指定する．どうやら CpCond[2]や CpCond[3]はコプロセッサとは直接関係ないようである．その端子状態が bc2t/bc2f/bc3t/bc3f 命令で参照できるか否かは不明である．

●MIPS16 の概要

▲MIPS16 とは

MIPS16 命令セットは ARM 社が自社の命令セットを拡張して 16 ビットの命令長に対応させた Thumb 命令セットに対抗するものとして LSI ロジック社によって提唱された．その目的は，組み込み制御分野におけるプログラムのメモリ占有率を低減することである．それを MIPS 社が自社のアーキテクチャの拡張として認定した形になっている．

基本命令長は 16 ビットで，32 ビット長の MIPS 本来の命令セット(以下では，仮に MIPS32 と呼ぶ)と混在して使用できる．

MIPS プロセッサを製造していた Lexra 社の試算によると，命令コードの容量は MIPS32 に比べて 60%の大きさになるという．MIPS32 の命令が 32 ビット長なので 50%ではないかと思う人がいるかもしれない．現実には 16 ビット命令だけではなく，32 ビット長命令との混合になるのでまったく半分のコードサイズにはならない．

ともあれ，MIPS16 は PalmSize 用の WindowsCE などで実際に採用された．

MIPS16 の特徴を箇条書きにすると次のようになる．

－16 ビット長の命令形式
－MIPS32 と混在可能
－8/16/32/64 ビット長のデータをサポート
－8 本の汎用レジスタと，いくつかの特殊レジスタを使用可能
－PC 相対命令を使用可能
－特権命令は定義されていない(必要な場合は MIPS32 で行なう)

MIPS16 をサポートしている MPU は結構存在する.本家である LSI ジック社の TR4101/4102，Lexra 社の LX4180，東芝の TX19，NEC の VR4100 シリーズなどがある．MIPS 社の SmartMIPS アーキテクチャを採用する 4KSc や 4KSd では積極的に MIPS16 を使用する．

ただし，NEC の VR4100 シリーズ以外は，32 ビットモードのみのサポートであり，64 ビット命令と 64 ビットデータを使用することはできない．MIPS 社の IP コアである 64 ビットの

5K (Opal)でも MIPS16 のサポートを匂わせていたが, 現状どうなっているのか不明である.

▲レジスタセット

表 12 に MIPS16 のレジスタセットを示す. これらのレジスタは MIPS32 のレジスタの一部分であり, MIPS16 から MIPS32, あるいは, MIPS32 から MIPS16 へのモード遷移を行っても保存される.

MIPS16 における汎用レジスタは, MIPS32 の内から 8 本が使用できる. 単純には r0 から r7 が割り当てられるところであるが, MIPS32 において, r0 は値が 0 固定, r1 はアセンブラのテンポラリレジスタであるので, それらの割り当てを避け, 代わりに r16 と r17 を利用する. これら 8 本の汎用レジスタに加え, 命令によっては, スタックポインタ(sp, r29), リターンアドレスレジスタ(ra, r31), 条件コードレジスタ(r24), プログラムカウンタ(pc)を参照できる. また, move 命令を使用すれば, MIPS32 のすべての汎用レジスタとの間でデータの転送が可能である.

MIPS16 では, 汎用レジスタが 8 本であるため, 実際のプログラムではスタック領域へのレジスタの退避と回復が頻繁に出現する. メモリアクセスが増加すればプログラムの処理性能は低下するが, move 命令を使用して MIPS32 のレジスタを退避領域に用いればメモリアクセスをなくすことができる. 後述する MIPS16e では複数レジスタを一括して退避/回復を行なう SAVE/RESTORE 命令が新設された.

▲モード遷移

MPU は PC 最下位ビットが 1 のとき, MIPS16 モードで命令を実行する. したがって, PC の最下位ビットをセット／リセットすることで,MIPS16 と MIPS32 間でモード遷移を行なえる. これを実現する方法には次のようなものがある.

－JAL 命令はモードを保持する. ra(リターンレジスタ)の最下位ビットに現在のモードが設定される.
－JALX 命令はモードを逆転する. ra(リターンレジスタ)の最下位ビットに現在のモードが設定される.
－JALR 命令は指定するレジスタの最下位ビットがモードを決定する. ra(リターンレジスタ)の最下位ビットに現在のモードが設定される.

－JR 命令は指定するレジスタの最下位ビットがモードを決定する.
－ERET 命令(MIPS32 のみ)は EPC の最下位ビットがモードを決定する.
－例外／割り込みが発生すると強制的に MIPS32 モードになる.

▲MIPS16 の命令形式

MIPS16 の命令は 16 ビット長を持ち, ハーフワード(16 ビット)境界に配置される. 図 13 に MIPS16 の命令形式を示す. 種類が多いようにも思えるが, 基本的には 2 オペランド形式(MIPS32 は 3 オペランド形式)で, 16 ビットのうち, 5 ビットが命令の種類, 3 ビット 2 つが各レジスタ番号, 残る 5 ビットがイミーディエト値を示す. JAL と JALX については, 16 ビット長では分岐先を指定できないので, 32 ビット長となっている.

ところで, イミーディエト値が 5 ビットでは, 0 から 31 までの値しか指定できないので, プログラムを書く上で不便である. そこで, EXTEND プリフィックスが用意されている. イミーディエト領域を持つ命令は EXTEND プリフィクスを付けることで 32 ビット長に拡張できる. この場合, イミーディエト値は 16 ビットの範囲まで指定できる. 図 14 に EXTEND さ

れた命令形式を示す.

▲MIPS16 の命令セット

MIPS16 の命令セットの詳細を表 13-1 から表 13-7 に示す．表を見ればわかるが，MIPS16 の命令は，ニーモニック的には，MIPS32 の命令のサブセットである．MIPS32 と同じく，ロード／ストアアーキテクチャを採用している．MIPS32 に慣れ親しんでいる人には馴染みやすいであろう．違いは，MIPS32 が 3 オペランド形式を標準としているのに対し，MIPS16 は 2 オペランド形式である．とはいえ，必要性の高い加減算命令は 3 オペランド形式を維持している.

MIPS32 の命令セットとの最大の違いは，分岐命令に遅延スロットがないことである．RISC のパイプラインの性格上，分岐命令が分岐先の命令を実行するまでに，1〜2 クロックのアイドル期間が生じる．この期間に分岐命令の後続 1 命令を実行して，パイプライン処理の無駄をなくすというのが遅延分岐の考え方である．MIPS16 の分岐命令においても，同様にアイドル期間は存在するが，分岐が成立する場合でも，遅延スロットの命令は抹殺される．なぜ，このような無駄を生じる仕様にしているのか，LSI ロジック社に聞いて見なければ真相はわからない．おそらく，遅延スロットがない方がコンパイラでの最適化が容易であるためだとは思うのだが．そのくせ，ジャンプ命令にはしっかりと遅延スロットが存在する．

また，PC 相対アドレッシングを使用する命令は MIPS32 にはない．これはなくても特に不自由はしないのだが，命令コード中に埋め込まれた定数値やテーブルへのアクセスを簡潔に実現するために導入されたのであろう(コンパイラでは SWITCH 文の実現に利用できる)その証拠に，ロード命令には PC 相対アドレッシングがあるのに，ストア命令にはない．読み込み専用を仮定しているのは明らかである.

この他にも，コンパイラでの使用を仮定したと思われる命令がいくつかある．まず，加算命令が符合なしのみである．MIPS アーキテクチャでは符合つき加算の実行でオーバーフローが発生すると例外を発生する．これは C 言語の仕様とは相容れないので，コンパイラは使用しない．また，ra(リターンアドレス)レジスタの特別扱いや，スタックポインタ相対アドレッシングなどもコンパイラ用と思われる.

▲MIPS16 の神話

巷には，MIPS16 モードを使用するとシステムの性能が向上するという神話がある．これについて考察しよう．MIPS16 では，命令長が基本的に 16 ビットになる．このため，命令長が 32 ビットの MIPS32 に対して，命令フェッチにおける単位バスサイクル当たりの転送命令数が倍になる．また，小規模システムでデータバスが 16 ビット幅の場合に，効率的な命令フェッチが可能になる．…といったイメージが浮かび，それに伴い，命令の処理速度も向上すると思ってしまう.

しかし，これは誤りである．命令フェッチは命令キャッシュから 1 命令単位に行われるので，外部バスのビット幅には影響を受けない．MIPS 互換プロセッサを製造している Lexra 社の試算による圧縮効率は 60%であるが，MIPS16 にしたからといってデータサイズは変わらない．トータルなサイズとしてどの程度節約できるか疑問がある.

また，命令のコードサイズに関しても 50%にならないということは，命令数としては 1.2 倍(60%/50%)に増加していることになる．基本的に 1 命令の実行時間は 1 クロックというのは変わらない(シングルパイプラインの場合)ので，単純に計算しても，命令の処理時間は 20%増加する.

実際には，イミーディエト値やディスプレースメントのビット数を拡張する EXTEND 命令は 16 ビット命令と結合されて 32 ビット長になるが，EXTEND された命令の実行には 2 クロ

ックを要するので，さらに実行時間は長くなる．

加えて,プログラムで使用できる汎用レジスタが,MIPS32では32本であったのに対して，MIPS16では8本に減少するので必然的にレジスタのメモリへの退避／回復の頻度が増加する．これも性能低下の要因になる．

こう考えてくると，MIPS16を積極的に採用する理由は見つからない．敢えて利点を探すとしたら，命令セットが比較的Z80に似ているので，歴史的に十分にこなれたコンパイラ技術を適用できるということくらいか．実際に，レジスタをあまり使用しないようなアプリケーションではMIPS16のコンパイラは結構優秀なコードを出力する.といっても,MIPS32のコンパイラ技術はそれよりもさらに優れていると思うのだが．なにしろ，RISC全盛の牽引力となったのがMIPSコンパイラであるという説もある位であるから．

MIPS16とよく似た経緯で誕生したARMアーキテクチャのThumb命令セットでは，コードサイズは70%になるが処理時間は45%低下するというのがARM社の見解である．MIPS16にもこれと同じことが言えるだろう．経験的にはMIPS32の半分程度の性能になるときもあるが，ほとんど変わらない場合もある．平均的に45%の性能低下ということはない．せいぜい，20〜30%の性能低下か．

しかし，このようなネガティブな論理が成り立つということは，性能が低下してもコードサイズが重要なアプリケーションが存在することを意味する．たとえば，ICカードなどの応用では数10MHzで動作させれば十分という話もある．

●新しい命令セット

▲MIPS32/MIPS64，そしてリリース2

MIPSの命令セットアーキテクチャであるMIPS I〜Vを，MIPS社が注力していこうとしている組み込み制御分野に適合するように見直しをしたのがMIPS32，MIPS64という命令セットである．

MIPS32はMIPS I，MIPS IIのスーパーセットで，従来の命令セットのうち，32ビットモードの命令を基準にしながら，メモリ管理や特権モードはR4000やR5000で用いられている命令を採用して定義されている．メモリ管理を簡単にするために,MMUがない構成や,TLBの代わりにブロックアドレス変換（BAT)と呼ばれる簡略版のMMUを採用することも可能である．通信やマルチメディアのアプリケーションでのデータのスループットを向上する目的で，MIPS IVから条件転送命令やプリフェッチ命令が採り入れられている．また，AV機器やマルチメディア用途で必要になる固定小数点(早い話が整数だが)専用のDSP機能の命令が新設された．具体的には，汎用レジスタに結果を格納する乗算（MUL)，積和演算(Multiply-Add/Multiply-Subtract：MADD，MADDU，MSUB，MSUBU）と連続1/0検出(Count Leading Zero/One：CLZ，CLO)命令である．

ところで，これらの命令機能は既に各社が独自に実装していて，同じMIPS系MPUでも互換性がない部分だった．MIPS社が正式に命令セットに組み込むことでプログラムの互換性が生まれることが期待される．

MIPS64はMIPS IV，MIPS Vのスーパーセットである．MIPS32とも下位互換があり，64ビット命令と64ビットの仮想アドレス空間をサポートしている．

また，MIPS32/MIPS64では，装置のデバッグ用にEJTAG(Enhanced JATG)と呼ばれるICE機能も標準装備している(トレース機能はオプション)．

MIPS32/MIPS64の追加命令はオペコードマップのSpecial2という領域を使用している．

今後，MIPSアーキテクチャを採用するMPUはMIPS32またはMIPS64のいずれかをサポートすることになるであろう．

2001年10月15日，MIPS社はMIPS32/MIPS64の組み込み制御向けの拡張アーキテクチャ

を発表した．これは，主として，割り込み応答を高速化するもので，レジスタバンクを 16 個持ち，16 種類の割り込みベクタを生成する．ライバルである ARM に近いアーキテクチャとなったが，MIPS 系では割り込み応答が遅いという不平に応えたものであろう．さらに、ビットフィールド命令，ローテート命令，バイトスワップ命令の追加，FPU などのコプロセッサ接続方法の仕様化，MMU のページサイズを 1KB から 256MB に拡張，L2，L3 キャッシュインタフェースの定義を行っている．これらによって使い易さを増大させた．この命令セットは俗にリリース 2 と呼ばれる．リリース 2 の命令セットのうち，主要なものを表 14 に示す．リリース 2 の追加命令はオペコードマップの Special3 という領域を使用している．

現在，リリース 2 を実装しているのは，M4K(コードネーム Quartz)と 24K(コードネーム Topaz)のみである．また，どちらも MIPS32 ベースであり，MIPS64 ベースのリリース 2 を実装する MPU は現在のところ存在しない．

▲MIPS-3D ASE

MIPS-3D ASE(Application Specific Extension)は，3 次元グラフィックアプリケーションを高速に処理するために MIPS64 の仕様を拡張したものである．

3 次元ジオメトリ処理専用に，従来の命令セットに新たな 13 命令を追加した(表 15)．これは，MIPS V のスーパーセットになっている．もともと MIPS V は固定小数点や浮動小数点を DSP と同様の方式で処理するために強化されたものである．

MIPS64 の浮動小数点命令は，単精度，倍精度，ペアドシングル(Paired Single)のデータ型を処理できる．ペアドシングルとは，1 つの 64 ビットレジスタに 2 つの単精度浮動小数点データを格納し，2 ウェイの SIMD(Single Instruction Multiple Data)方式での処理(要は並列実行)を可能にするデータ型である．これらを更に有効活用できるようにしたのが MIPS-3D であるといえる．

ペアドシングル間の加減乗除，平方根演算などを行なう命令は，従来 MIPS V で定義されていた．それらがそのまま MIPS64 に移行された．

MIPS-3D 命令を用いると，画像処理アプリケーションにおいて，もっとも内側の処理ループのコードサイズを 30%削減できるので，1 秒間に処理できるポリゴン数が 45%増加するそうだ．たとえば，頂点の座標変換での行列の乗算を高速化するために，ペアドシングル・リダクション加算命令(ADDR)が定義された．画像のクリッピングは，ペアドシングル絶対値比較命令(CABS)と多重条件コード分岐命令（BC1ANYnx）によって簡略化できる．透視変換には逆数命令(RECIP1，RECIP2)が使用できる．光源処理には逆数平方根命令(RSQRT1，RSQRT2)が使用できる．

また，ペアドシングルの他にペアドワード(Paired Word)というデータ型も定義されていて，ペアドシングルとペアドワード間でデータ変換を高速に行なう命令(CVT.PS.PW，CVT.PW.PS)もある．

MIPS-3D の応用分野としては，デジタルセットトップボックス，ゲーム機，PDA，DVD などの低価格な組み込み分野での 3 次元グラフィック処理を目的にしている．MIPS 社の性能試算では，500MHz 動作の MPU で，秒間 2500 万ポリゴン(光源付きだと 1000 万ポリゴン)を処理できるそうである．

この命令セットを最初に実装したのが，R20000(Ruby)である．しかし，Ruby とその後継機種の 25Kf(Amethyst)は，現在では保守(?)扱いになっているようであり，MIPS 社のロードマップから消えている．

▲MIPS16e ASE

MIPS16e は MIPS16 のコードサイズを更に削減するために定義された．発表時期は 2001 年

の 4 月である．数種の命令が MIPS16 に追加され，従来の MIPS16 よりも 10〜15%のコードサイズの縮小になるという話である．

具体的には，遅延スロットのないジャンプ命令，複数レジスタの一括退避/回復，レジスタの符号/ゼロ拡張命令が追加されている．表 16 に MIPS16e で追加された命令を示す．

現在，MIPS16e は 32 ビット命令の範囲で定義されている．仕様書では，64 ビット命令での定義もあると記述されているが，公開されていない．また，従来の MIPS16 は存在しなかったものとされ，今では MIPS16 というと MIPS16e のことを指す．MIPS16e は MIPS16 のスーパーセットなので特に問題はない．

2002 年 1 月には，MIPS16e を拡張した MIPS16e+を東芝が発表した．自社の TX19N に実装するという話だった．MIPS16e+は，MIPS16e に対して，単一ビット操作命令やビットフィールド命令が追加され，より組み込み制御分野に特化したのになっている．

この発表で不思議だったのが，その時点で MIPS16 の命令コードマップは既存の命令でほとんど埋め尽くされており，拡張した命令を割り当てる場所がないことである．その解決策はこうだ．詳細は不明だが，64 ビット命令を潰して，その位置に拡張した命令を割り当てているらしい．つまり，MIPS16e+は 32 ビットモード限定ということになる．

●おわりに

MIPS の命令セットとその具体的な使い方について説明してきた．これからの傾向としてアセンブリ言語でプログラムを書く人はあまりいないと思う．しかし，真に処理を高速化したい場合など，アセンブリ言語が有用な場合もあると思われる．

私見であるが，MIPS の命令セットは Z80 に比べれば数段，MC68000 と比べてもかなり洗練されていると思う．みなさんはどう思われたであろうか．これを機会に MIPS RISC に対する興味が深まってもらえれば幸いである．

※本稿はかつて Oh!X 誌に連載した『はじめて読む MIPS』を最新の情報を付加して再構成したものである．

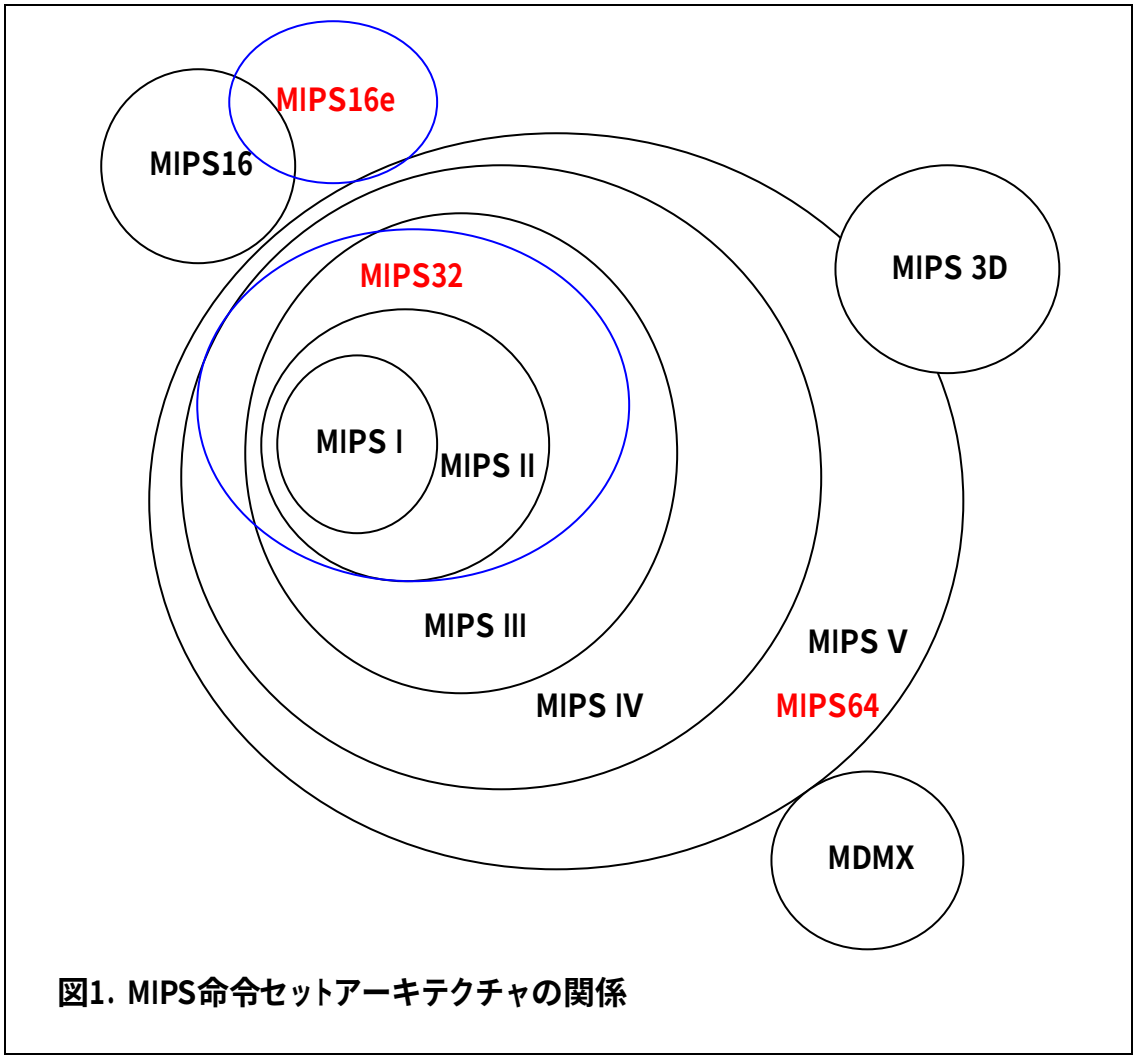


図1. MIPS命令セットアーキテクチャの関係

表 1. 命令セットの変遷

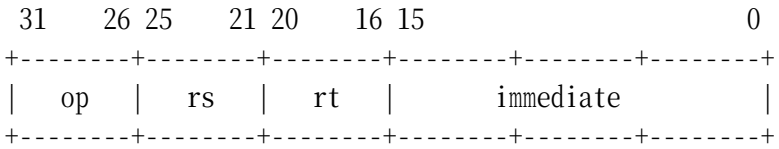
バージョン	発表時期	最初に実装されたプロセッサ
MIPS I	1984	R2000, R3000
MIPS II	1990	R6000
MIPS III	1991	R4xxx
MIPS IV	1994	R5xxx, R7xxx, R8000, R10000
MIPS V	1996	まだない

表 2. 汎用レジスタの割り当て

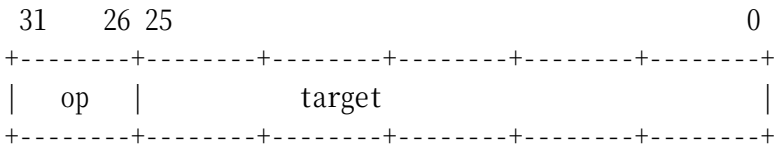
レジスタ名	ソフトウェア名	使用目的
r0	zero	常に０.
r1	at	アセンブラのテンポラリレジスタ.
r2,r3	v0,v1	関数の戻り値.
r4-r7	a0-a3	関数への引数.
r8-r15	t0-t7	テンポラリレジスタ. 関数呼び出しで破壊.
r16-r23	s0-s7	関数呼び出しで不変なレジスタ.
r24,r25	t8,t9	テンポラリレジスタ. 関数呼び出しで破壊.
r26,r27	k0,k1	ＯＳ用に予約済み.
r28	gp	大域変数領域のベースアドレス.
r29	sp	スタックポインタ.
r30	s8(fp)	s0-s7 と同様. 必要ならフレームポインタ.
r31	ra	関数からの戻り値.

図 2. CPU の命令形式

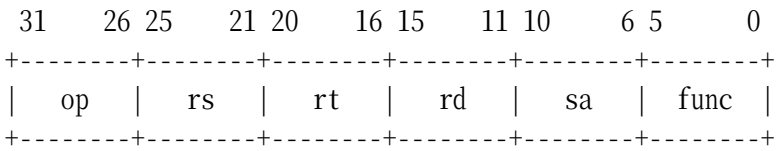
●I-Type(イミーディエト形式)



●J-Type(ジャンプ形式)



●R-Type(レジスタ形式)



op: 6 ビットの命令コード.
rs: 5 ビットのソースレジスタ.
rt: 5 ビットのターゲット(ソース／デスティネーション)レジスタ. 分岐条件.
immediate:16 ビットのイミーディエト値. 分岐, アドレスディスプレースメント.
target:26 ビットの無条件分岐ターゲットアドレス.
rd: 5 ビットのデスティネーションレジスタ.
sa: 5 ビットのシフト量.
func: 6 ビットの機能フィールド.

表 3. ロード／ストア命令

命令	形式と説明	op	base	rt/hint	offset	
Load Byte	LB rt,offset(base) 符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する. アドレスで指定されたバイトデータを符号拡張して，レジスタ rt にロードする.					
Load Byte Unsigned	LBU rt,offset(base) 符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する. アドレスで指定されたバイトデータをゼロ拡張して，レジスタ rt にロードする.					
Load Halfword	LH rt,offset(base) 符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する. アドレスで指定されたハーフワードデータを符号拡張して，レジスタ rt にロードする.					
Load Halfword Unsigned	LHU rt,offset(base) 符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する. アドレスで指定されたハーフワードデータをゼロ拡張して，レジスタ rt にロードする.					
Load Word	LW rt,offset(base) 符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する. アドレスで指定されたワードデータを符号拡張して，レジスタ rt にロードする.					
Load Word Left	LWL rt,offset(base) 符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する. アドレスで指定されたバイトがワードの最左端になるようにワードデータを左シフトする．シフト結果とレジスタ rt の内容をマージし，符号拡張して，レジスタ rt にロードする.					
	LWR rt,offset(base)					

Load Word Right	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．アドレスで指定されたバイトがワードの最右端になるようにワードデータを右シフトする．シフト結果とレジスタ rt の内容をマージし，符号拡張して，レジスタ rt にロードする．	
	SB rt,offset(base)	
Store Byte	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．アドレスで指定されたメモリに，レジスタ rt の最下位バイトの内容をストアする．	
	SH rt,offset(base)	
Store Halfword	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．アドレスで指定されたメモリに，レジスタ rt の最下位ハーフワードの内容をストアする．	
	SW rt,offset(base)	
Store Word	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．アドレスで指定されたメモリに，レジスタ rt の最下位ワードの内容をストアする．	
	SWL rt,offset(base)	
Store Word Left	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．ワードの最左端バイトがアドレス指定されたバイト位置になるように，レジスタ rt の内容を右シフトする．シフト結果をメモリ中のワードの下位部分にストアする．	
	SWR rt,offset(base)	
Store Word Right	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．ワードの最右端バイトがアドレス指定されたバイト位置になるように，レジスタ rt の内容を左シフトする．シフト結果をメモリ中のワードの上位部分にストアする．	
	LD rt,offset(base)	
Load Double-word	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．アドレスで指定されたダブルワードデータをレジスタ rt にロードする．	
	LDL rt,offset(base)	

Load	符号拡張した offset をレジスタ base の内容に加算しアドレス
Double-	を生成する.
word	アドレスで指定されたバイトがダブルワードの最左端になる
Left	ようにダブルワードデータを左シフトする. シフト結果とレ
	ジスタ rt の内容をマージし, レジスタ rt にロードする.

	LDR rt,offset(base)
Load	符号拡張した offset をレジスタ base の内容に加算しアドレス
Double-	を生成する.
word	アドレスで指定されたバイトがダブルワードの最右端になる
Right	ようにダブルワードデータを右シフトする. シフト結果とレ
	ジスタ rt の内容をマージし, レジスタ rt にロードする.

	LWU rt,offset(base)
Load	符号拡張した offset をレジスタ base の内容に加算しアドレス
Word	を生成する.
Unsigned	アドレスで指定されたワードデータをワード位置でゼロ拡張
	して, レジスタ rt にロードする.

	SD rt,offset(base)
Store	符号拡張した offset をレジスタ base の内容に加算しアドレス
Double-	を生成する.
word	アドレスで指定されたメモリに, レジスタ rt の内容をストア
	する.

	SDL rt,offset(base)
Store	符号拡張した offset をレジスタ base の内容に加算しアドレス
Double-	を生成する.
word	ダブルワードの最左端バイトがアドレス指定されたバイト位
Left	置になるように, レジスタ rt の内容を右シフトする. シフト
	結果をメモリ中のダブルワードの下位部分にストアする.

	SDR rt,offset(base)
Store	符号拡張した offset をレジスタ base の内容に加算しアドレス

Double-	を生成する.
word	ダブルワードの最右端バイトがアドレス指定されたバイト位
Right	置になるように, レジスタ rt の内容を左シフトする. シフト
	結果をメモリ中のダブルワードの上位部分にストアする.

	LL rt,offset(base)
Load	符号拡張した offset をレジスタ base の内容に加算しアドレス
Linked	を生成する.
	アドレスで指定されたワードデータを符号拡張して, レジス
	タ rt にロードする. CPU 内部資源の LL ビットを 1 に設定する.

	SC rt,offset(base)	
Store Conditional	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．CPU 内部資源の LL ビットが 1 のとき，アドレスで指定されたメモリに，レジスタ rt の最下位ワードの内容をストアする．同時にレジスタ rt に 1 を格納する．LL ビットが 0 のとき，メモリにストアは行なわず，レジスタ rt に 0 を格納する．	
	LLD rt,offset(base)	
Load Linked Double-word	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．アドレスで指定されたダブルワードデータをレジスタ rt にロードする．CPU 内部資源の LL ビットを 1 に設定する．	
	SCD rt,offset(base)	
Store Conditional Double-word	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．CPU 内部資源の LL ビットが 1 のとき，アドレスで指定されたメモリに，レジスタ rt の内容をストアする．同時にレジスタ rt に 1 を格納する．LL ビットが 0 のとき，メモリにストアは行わず，レジスタ rt に 0 を格納する．	
	PREF hint,offset(base)	
Prefetch	符号拡張した offset をレジスタ base の内容に加算しアドレスを生成する．hint が示す情報に従い，アドレスで指定されたデータをキャッシュにプリフェッチする．	

表 4-1. イミーディエト命令

命令	形式と説明 | op | rs | rt | immediate	
ADD Immediate	ADDI rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と加算する．加算結果をワードで符号拡張して，レジスタ rt に格納する．オーバーフローが発生すると例外になる．	
ADD Immediate Unsigned	ADDIU rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と加算する．加算結果をワードで符号拡張して，レジスタ rt に格納	

	する．オーバーフローが発生しても例外にならない．
Set on Less Than Immediate	SLTI rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，符号付き整数としてレジスタ rs と比較する．rs がイミーディエトより小さい場合はレジスタ rs に 1 を格納する．そうでない場合は 0 を格納する．
Set on Less Than Immediate Unsigned	SLTIU rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，符号なし整数としてレジスタ rs と比較する．rs がイミーディエトより小さい場合はレジスタ rs に 1 を格納する．そうでない場合は 0 を格納する．
AND Immediate	ANDI rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と AND をとる．演算結果をレジスタ rt に格納する．
OR Immediate	ORI rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と OR をとる．演算結果をレジスタ rt に格納する．
Exclusive OR Immediate	XORI rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と Ex. OR をとる．演算結果をレジスタ rt に格納する．
Load Upper Immediate	LUI rt,immediate 16 ビットのイミーディエトを 16 ビット左シフトし，ワードで符号拡張してレジスタ rt に格納する．
Double-word ADD Immediate	DADDI rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と加算する．加算結果をレジスタ rt に格納する．オーバーフローが発生すると例外になる．
Double-word ADD Immediate Unsigned	DADDIU rt,rs,immediate 16 ビットのイミーディエトを符号拡張し，レジスタ rs と加算する．加算結果をレジスタ rt に格納する．オーバーフローが発生しても例外にならない．

表 4-2. 3 オペランド命令

命令	形式と説明	op	rs	rt/cc	rd	sa	func

		+------+------+------+------+------+------+
	ADD rd,rs,rt	
ADD	レジスタ rs と rt の内容を加算する．加算結果をワードで符号	
	拡張して，レジスタ rd に格納する．オーバーフローが発生す	
	ると例外になる．	
	ADDU rd,rs,rt	
ADD Unsigned	レジスタ rs と rt の内容を加算する．加算結果をワードで符号	
	拡張して，レジスタ rd に格納する．オーバーフローが発生し	
	ても例外にならない．	
	SUB rd,rs,rt	
Subtract	レジスタ rs から rt の内容を減算する．減算結果をワードで符	
	号拡張して，レジスタ rd に格納する．オーバーフローが発生	
	すると例外になる．	
	SUBU rd,rs,rt	
Subtract Unsigned	レジスタ rs から rt の内容を減算する．加算結果をワードで符	
	号拡張して，レジスタ rd に格納する．オーバーフローが発生	
	しても例外にならない．	
	SLT rd,rs,rt	
Set Less Than	レジスタ rs と rt の内容を符号付き整数として比較する．rs が	
	rt よりも小さい場合は，レジスタ rd に 1 を格納する．そうで	
	ない場合は 0 を格納する．	
	SLTU rd,rs,rt	
Set Less Than Unsigned	レジスタ rs と rt の内容を符号なし整数として比較する．rs が	
	rt よりも小さい場合は，レジスタ rd に 1 を格納する．そうで	
	ない場合は 0 を格納する．	
	AND rd,rs,rt	
AND	レジスタ rs と rt の内容のビットごとの AND をとり，演算結果	
	をレジスタ rd に格納する．	
	OR rd,rs,rt	
OR	レジスタ rs と rt の内容のビットごとの OR をとり，演算結果	
	をレジスタ rd に格納する．	
	XOR rd,rs,rt	
Exclusive OR	レジスタ rs と rt の内容のビットごとの Exclusive OR をとり，	
	演算結果をレジスタ rd に格納する．	
	NOR rd,rs,rt	

NOR	レジスタ rs と rt の内容のビットごとの NOR をとり，演算結果をレジスタ rd に格納する．
Double-word ADD	DADD rd,rs,rt レジスタ rs と rt の内容を加算する．加算結果をレジスタ rd に格納する．オーバーフローが発生すると例外になる．
Double-word ADD Unsigned	DADDU rd,rs,rt レジスタ rs と rt の内容を加算する．加算結果をレジスタ rd に格納する．オーバーフローが発生しても例外にならない．
Double-word Subtract	DSUB rd,rs,rt レジスタ rs から rt の内容を減算する．減算結果をレジスタ rd に格納する．オーバーフローが発生すると例外になる．
Double-word Subtract Unsigned	DSUBU rd,rs,rt レジスタ rs から rt の内容を減算する．減算結果をレジスタ rd に格納する．オーバーフローが発生しても例外にならない．
※１	MOVT rd,rs,cc 浮動小数点条件 cc が１なら，レジスタ rs の内容をレジスタ rd に格納する．
※２	MOVF rd,rs,cc 浮動小数点条件 cc が０なら，レジスタ rs の内容をレジスタ rd に格納する．
※３	MOVZ rd,rs,rt レジスタ rt の値が０なら，レジスタ rs の内容をレジスタ rd に格納する．
※４	MOVN rd,rs,rt レジスタ rt の値が０以外なら，レジスタ rs の内容をレジスタ rd に格納する．

※１　Move Conditional on Floating Point True
※２　Move Conditional on Floating Point False
※３　Move Conditional on Zero
※４　Move Conditional on Not Zero

表 4-3. シフト命令

命令	形式と説明	op	rs	rt	rd	sa	func

	+------+------+------+------+------+------+
Shift Left Logical	SLL rd,rt,sa レジスタ rt の内容を sa ビット左シフトする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．
Shift Right Logical	SRL rd,rt,sa レジスタ rt の内容を sa ビット論理右シフトする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．
Shift Right Arithmetic	SRA rd,rt,sa レジスタ rt の内容を sa ビット算術右シフトする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．
Shift Left Logical Variable	SLLV rd,rt,rs レジスタ rt の内容をレジスタ rs の下位 5 ビットで指定される ビット数だけ左シフトする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．
Shift Right Logical Variable	SRLV rd,rt,rs レジスタ rt の内容をレジスタ rs の下位 5 ビットで指定される ビット数だけ論理右シフトする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．
Shift Right Arithmetic Variable	SRAV rd,rt,rs レジスタ rt の内容をレジスタ rs の下位 5 ビットで指定される ビット数だけ算術右シフトする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．
Double-word Shift Left Logical	DSLL rd,rt,sa レジスタ rt の内容を sa ビット左シフトする．シフト結果をレジスタ rd に格納する．
Double-word Shift Right Logical	DSRL rd,rt,sa レジスタ rt の内容を sa ビット論理右シフトする．シフト結果をレジスタ rd に格納する．
Double-word Shift Right	DSRA rd,rt,sa レジスタ rt の内容を sa ビット算術右シフトする．シフト結果をレジスタ rd に格納する．

|Arithmetic|

Double- word Shift Left Logical Variable	DSLLV rd,rt,rs レジスタ rt の内容をレジスタ rs の下位 6 ビットで指定される ビット数だけ左シフトする．シフト結果レジスタ rd に格納す る． ．
Double- word Shift Right Logical Variable	DSRLV rd,rt,rs レジスタ rt の内容をレジスタ rs の下位 6 ビットで指定される ビット数だけ論理右シフトする．シフト結果レジスタ rd に格 納する．
Double- word Shift Right Arithmetic Variable	DSRAV rd,rt,rs レジスタ rt の内容をレジスタ rs の下位 6 ビットで指定される ビット数だけ算術右シフトする．シフト結果レジスタ rd に格 納する．
Double- word Shift Left Logical +32	DSLL32 rd,rt,sa レジスタ rt の内容を(sa+32)ビット左シフトする．シフト結 果をレジスタ rd に格納する．
Double- word Shift Right Logical +32	DSRL32 rd,rt,sa レジスタ rt の内容を(sa+32)ビット論理右シフトする．シフ ト結果をレジスタ rd に格納する．
Double- word Shift Right Arithmetic +32	DSRA32 rd,rt,sa レジスタ rt の内容を(sa+32)ビット算術右シフトする．シフ ト結果をレジスタ rd に格納する．

表 4-4. 乗除算命令

命令	形式と説明	op	rs	rt	rd	sa	func
Multiply	MULT rs,rt レジスタ rs と rt の内容を符号付き 32 ビットデータとして乗算する．積の下位 32 ビットを符号拡張して LO レジスタに，積の上位 32 ビットを符号拡張して HI レジスタに格納する．						
Multiply Unsigned	MULTU rs,rt レジスタ rs と rt の内容を符号なし 32 ビットデータとして乗算する．積の下位 32 ビットを符号拡張して LO レジスタに，積の上位 32 ビットを符号拡張して HI レジスタに格納する．						
Divide	DIV rs,rt レジスタ rs を rt の内容で符号付き 32 ビットデータとして除算する．商を符号拡張して LO レジスタに，剰余を符号拡張して HI レジスタに格納する．						
Divide Unsigned	DIVU rs,rt レジスタ rs を rt の内容で符号なし 32 ビットデータとして除算する．商を符号拡張して LO レジスタに，剰余を符号拡張して HI レジスタに格納する．						
Double-word Multiply	DMULT rs,rt レジスタ rs と rt の内容を符号付き 64 ビットデータとして乗算する．積の下位 64 ビットを LO レジスタに，積の上位 64 ビットを HI レジスタに格納する．						
Double-word Multiply Unsigned	DMULTU rs,rt レジスタ rs と rt の内容を符号なし 64 ビットデータとして乗算する．積の下位 64 ビットを LO レジスタに，積の上位 64 ビットを HI レジスタに格納する．						
Double-word Divide	DDIV rs,rt レジスタ rs を rt の内容で符号付き 64 ビットデータとして除算する．商を LO レジスタに，剰余を HI レジスタに格納する．						
Double-word Divide Unsigned	DDIVU rs,rt レジスタ rs を rt の内容で符号なし 64 ビットデータとして除算する．商を LO レジスタに，剰余を HI レジスタに格納する．						

Move From HI	MFHI rd HI レジスタの内容をレジスタ rd に格納する.
Move From LO	MFLO rd LO レジスタの内容をレジスタ rd に格納する.
Move To HI	MTHI rs レジスタ rs の内容を HI レジス格納する.
Move To LO	MTLO rs レジスタ rs の内容を LO レジス格納する.
Multiply Word to Register	MUL rd,rs,rt レジスタ rs と rt の内容を符号付き 32 ビットデータとして乗算する. 積の下位 32 ビットをレジスタ rd に格納する.
Multiply And Add	MADD rs,rt レジスタ rs と rt の内容を符号付き 32 ビットデータとして乗算する. 積の下位 32 ビットを LO レジスタの内容と加算して LO レジスタに格納する. 積の上位 32 ビットを HI レジスタの内容と加算して HI レジスタに格納する.
Multiply And Add Unsigned	MADDU rs,rt レジスタ rs と rt の内容を符号なし 32 ビットデータとして乗算する. 積の下位 32 ビットを LO レジスタの内容と加算して LO レジスタに格納する. 積の上位 32 ビットを HI レジスタの内容と加算して HI レジスタに格納する.
Multiply And Subtract	MSUB rs,rt レジスタ rs と rt の内容を符号付き 32 ビットデータとして乗算する. 積の下位 32 ビットを LO レジスタの内容から減算して LO レジスタに格納する. 積の上位 32 ビットを HI レジスタの内容から減算して HI レジスタに格納する.
Multiply And Subtract Unsigned	MSUBU rs,rt レジスタ rs と rt の内容を符号なし 32 ビットデータとして乗算する. 積の下位 32 ビットを LO レジスタの内容から減算して LO レジスタに格納する. 積の上位 32 ビットを HI レジスタの内容から減算して HI レジスタに格納する.

表 5. ジャンプ／分岐命令

命令	形式と説明	op	target				
Jump	J target 26 ビットのターゲットアドレスを左に 2 ビット分シフトし，PC の上位 4 ビットと結合したアドレスへ，1 命令遅れてジャンプする．						
Jump And Link	JAL target 26 ビットのターゲットアドレスを左に 2 ビット分シフトし，PC の上位 4 ビットと結合したアドレスへ，1 命令遅れてジャンプする．遅延スロットに続く命令のアドレスを r31 (リンクレジスタ)に格納する．						
Jump And Link Exchange	JALX target 26 ビットのターゲットアドレスを左に 2 ビット分シフトし，PC の上位 4 ビットと結合したアドレスへ，1 命令遅れてジャンプする．遅延スロットに続く命令のアドレスを r31 (リンクレジスタ)に格納する．ジャンプ先の命令は MIPS16 として実行する．						

命令	形式と説明	op	rs	rt	rd	sa	func
Jump Register	JR rs レジスタ rs の内容が示すアドレスに，1 命令遅れてジャンプする．						
Jump And Link Register	JALR rd,rs レジスタ rs の内容が示すアドレスに，1 命令遅れてジャンプする．遅延スロットに続く命令のアドレスをレジスタ rd に格納する．						

命令	形式と説明	op	rs	rt	offset
Branch on Equal	BEQ rs,rt,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容とレジスタ rt の内容が等しいとき，1 命令						

	遅れて分岐する.
Branch on Not Equal	BNE rs,rt,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容とレジスタ rt の内容が等しくないとき, 1 命令遅れて分岐する.
Branch on Less Than or Equal to Zero	BLEZ rs,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容が 0 以下の場合, 1 命令遅れて分岐する.
Branch on Greater Than Zero	BGTZ rs,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容が 0 より大きい場合, 1 命令遅れて分岐する.
Branch on Less Than Zero	BLTZ rs,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容が 0 より小さい場合, 1 命令遅れて分岐する.
Branch on Greater Than or Equal to Zero	BGEZ rs,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容が 0 以上の場合, 1 命令遅れて分岐する.
Branch on Less Than Zero And Link	BLTZAL rs,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容が 0 より小さい場合, 1 命令遅れて分岐する. 分岐遅延スロットに続く命令のアドレスをレジスタ r31 (リンクレジスタ)に格納する.
Branch on Greater Than or Equal to	BGEZAL rs,offset 16 ビットの offset を符号拡張し, 2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する. レジスタ rs の内容が 0 以上の場合, 1 命令遅れて分岐する. 分岐遅延スロットに続く命令のアドレスをレジスタ r31(リン

Zero And Link	クレジスタ)に格納する.
Branch on CP0 True	BC0T offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．ステータスレジスタの CH ビットが 1 の場合，1 命令遅れて分岐する.
Branch on CP0 False	BC0F offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．ステータスレジスタの CH ビットが 0 の場合，1 命令遅れて分岐する.
Branch on Equal Likely	BEQL rs,rt,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容とレジスタ rt の内容が等しいとき，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない.
Branch on Not Equal Likely	BNEL rs,rt,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容とレジスタ rt の内容が等しくないとき，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない.
Branch on Less Than or Equal to Zero Likely	BLEZL rs,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容が 0 以下の場合，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない.
Branch on Greater Than Zero Likely	BGTZL rs,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容が 0 より大きい場合，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない.
Branch on Less	BLTZL rs,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と

Than Zero Likely	分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容が 0 より小さい場合，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない．
Branch on Greater Than or Equal to Zero Likely	BGEZL rs,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容が 0 以上の場合，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない．
Branch on Less Than Zero And Link Likely	BLTZALL rs,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容が 0 より小さい場合，1 命令遅れて分岐する．分岐遅延スロットに続く命令のアドレスをレジスタ r31 (リンクレジスタ)に格納する．分岐しない場合は，分岐遅延スロットの命令は実行しない．
Branch on Greater Than or Equal to Zero And Link Likely	BGEZALL rs,offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．レジスタ rs の内容が 0 以上の場合，1 命令遅れて分岐する．分岐遅延スロットに続く命令のアドレスをレジスタ r31(リンクレジスタ)に格納する．分岐しない場合は，分岐遅延スロットの命令は実行しない．
Branch on CP0 True Likely	BC0TL offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．ステータスレジスタの CH ビットが 1 の場合，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない．
Branch on CP0 False Likely	BC0FL offset 16 ビットの offset を符号拡張し，2 ビット左シフトした値と分岐遅延スロットのアドレスを加算して分岐先を計算する．ステータスレジスタの CH ビットが 0 の場合，1 命令遅れて分岐する．分岐しない場合は，分岐遅延スロットの命令は実行しない．

表 6. 特殊命令

命令	形式と説明	op	rs	rt	rd	sa	func
Synchronize	SYNC 現在パイプライン内にあるロード／ストア命令を，新たなロード／ストア命令が実行される前に完結させる.						
System Call	SYSCALL システムコール例外を発生させる.						
Break Point	BREAK ブレークポイント例外を発生させる.						
Count Leading Zero	CLZ rd,rs レジスタ rs のビット 31 からビット 0 方向にビットを走査し，連続する 0 の数をレジスタ rd に格納する.						
Count Leading One	CLO rd,rs レジスタ rs のビット 31 からビット 0 方向にビットを走査し，連続する 1 の数をレジスタ rd に格納する.						
Double-word Count Leading Zero	DCLZ rd,rs レジスタ rs のビット 63 からビット 0 方向にビットを走査し，連続する 0 の数をレジスタ rd に格納する.						
Double-word Count Leading One	DCLO rd,rs レジスタ rs のビット 63 からビット 0 方向にビットを走査し，連続する 1 の数をレジスタ rd に格納する.						
Trap if Greater Than or Equal	TGE rs,rt レジスタ rs と rt の内容を符号付き整数として比較する．rs が rt 以上であれば，トラップ例外を発生させる.						
Trap if Greater Than or Equal	TGEU rs,rt レジスタ rs と rt の内容を符号なし整数として比較する．rs が rt 以上であれば，トラップ例外を発生させる.						

|Unsigned|

Trap if Less Than Unsigned	TLT rs,rt レジスタ rs と rt の内容を符号付き整数として比較する．rs が rt より小さい場合，トラップ例外を発生させる．
Trap if Less Than Unsigned	TLTU rs,rt レジスタ rs と rt の内容を符号なし整数として比較する．rs が rt より小さい場合，トラップ例外を発生させる．
Trap if Equal	TEQ rs,rt レジスタ rs と rt の内容が等しい場合，トラップ例外を発生させる．
Trap if Not Equal	TNE rs,rt レジスタ rs と rt の内容が等しくない場合，トラップ例外を発生させる．

命令	形式と説明	op	rs	rt	immediate
Trap if Greater Than or Equal Immediate	TGEI rs,immediate レジスタ rs の内容と，16 ビットのイミーディエトを符号拡張した値を符号付き整数として比較する．rs がイミーディエト以上であれば，トラップ例外を発生する．
Trap if Greater Than or Equal Unsigned Immediate	TGEIU rs,immediate レジスタ rs の内容と，16 ビットのイミーディエトをゼロ拡張した値を符号なし整数として比較する．rs がイミーディエト以上であれば，トラップ例外を発生する．
Trap if Less Than Immediate	TLTI rs,immediate レジスタ rs の内容と，16 ビットのイミーディエトを符号拡張した値を符号付き整数として比較する．rs がイミーディエトより小さい場合，トラップ例外を発生する．
Trap if Less	TGEIU rs,immediate レジスタ rs の内容と，16 ビットのイミーディエトをゼロ拡張

Than Unsigned Immediate	した値を符号なし整数として比較する．rs がイミーディエトより小さい場合，トラップ例外を発生する．
Trap if Equal To Immediate	TEQI rs,immediate レジスタ rs の内容と，16 ビットのイミーディエトを符号拡張した値が等しい場合，トラップ例外を発生する．
Trap if Equal Not To Immediate	TEQI rs,immediate レジスタ rs の内容と，16 ビットのイミーディエトを符号拡張した値が等しくない場合，トラップ例外を発生する．

表 7．システム制御コプロセッサ命令

命令	形式と説明	COP0	sub	rt	rd	0	sel
Move To CP0	MTC0 rt,rd CPU レジスタ rt のワードの内容を CP0 レジスタ rd に転送する．						
Move From CP0	MFC0 rt,rd CP0 レジスタ rd のワードの内容を CPU レジスタ rt に転送する．						
Double-word Move To CP0	DMTC0 rt,rd CPU レジスタ rt のダブルワードの内容を CP0 レジスタ rd に転送する．						
Double-word Move From CP0	DMFC0 rt,rd CP0 レジスタ rd のダブルワードの内容を CPU レジスタ rt に転送する．						

命令	形式と説明	COP0	CO	func
TLB Read	TLBR インデクスレジスタで指定された TLB エントリの内容を，エントリ Hi，エントリ Lo0，エントリ Lo1，ページマスクレジスタに格納する．			

TLB Probe	TLBP 仮想アドレスがエントリ Hi レジスタの内容と一致する TLB のエントリ番号をインデクスレジスタに格納する．一致するエントリがない場合は，インデクスレジスタのビット 31 をセットする．
TLB Write Index	TLBWI インデクスレジスタで指定された TLB エントリに，エントリ Hi，エントリ Lo0，エントリ Lo1，ページマスクレジスタの内容を格納する．
TLB Write Random	TLBWR ランダムレジスタで指定された TLB エントリに，エントリ Hi，エントリ Lo0，エントリ Lo1，ページマスクレジスタの内容を格納する．
Return From Exception	ERET 例外／割り込み／トラップから復帰する．復帰先はエラー EPC レジスタまたは EPC レジスタから取り出す．
Wait	WAIT CPU を低消費電力状態に移行させ，割り込みを待つ．

命令	形式と説明
Cache Operation	CACHE op, offset(base) レジスタ base と，16 ビットの offset を符号拡張をした値を加算して仮想アドレスを計算する．その仮想アドレスに対応するキャッシュのエントリに対して op で定める処理を行なう．

図 3. CPU 命令のオペコードマップ

opcode									
	bit28:26	0	1	2	3	4	5	6	7
bit31:29		000	001	010	011	100	101	110	111
0	000	SPECIAL	REGIMM	J	J	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	COP0	COP1	COP2	COP1X	BEQL	BNEL	BLEZL	BGTZL
3	011	DADDI	DADDIU	LDL	LDR	SPECIAL2	JALX	MDMX	SPECIAL3
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5	101	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE
6	110	LL	LWC1	LWC2	PREF	LLD	LDC1	LDC2	LD
7	111	SC	SWC1	SWC2		SCD	SDC1	SDC2	SD

SPECIAL

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	SLL	MOVCI	SRL	SRA	SLLV		SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK		SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV		DSRLV	DSRAV
3	011	MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101			SLT	SLTU	DADD	DADDU	DSUB	DSUBU
6	110	TGE	TGEU	TLT	TLTU	TEQ		TNE	
7	111	DSLL		SDRL	DSRA	DSLL32		DSRL32	DSRA32

REGIMM

	bit18:16	0	1	2	3	4	5	6	7
bit20:19		000	001	010	011	100	101	110	111
0	000	BLTZ	BGEZ	BLTZL	BGEZL				
1	001	TGEI	TGEIU	TLTI	TLTIU	TEQI		TNEI	
2	010	BLTZAL	BGEZAL	BLTZALL	BGEZALL			DSRLV	
3	011								SYNCI

SPECIAL2

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL		MSUB	MSUBU		
1	001								
2	010								
3	011								
4	100	CLZ	CLO			DCLZ	DCLO		
5	101								
6	110								
7	111								SDBBP

SPECIAL3

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	EXT	DEXTM	DEXTU	DEXT	INS	DINSM	DINSU	DINS
1	001								
2	010								
3	011								
4	100	BSHFL				DBSHFL	DCLO		
5	101								
6	110								
7	111				RDHWR				SDBBP

MOVCI

bit16	
0	1
MOVF	MOVT

SRL

bit21	
0	1
SRL	ROTR

SRLV

bit6	
0	1
SRLV	ROTRV

DSRL

bit21	
0	1
DSRL	DROTR

DSRLV

bit6	
0	1
DSRLV	DROTRV

DSRL32

bit21	
0	1
DSRL32	DROTR32

BSHFL

	bit8:6	0	1	2	3	4	5	6	7
bit10:9		000	001	010	011	100	101	110	111
0	000			WSBH					
1	001								
2	010	SEB							
3	011	SHE							

DBSHFL

	bit8:6	0	1	2	3	4	5	6	7
bit10:9		000	001	010	011	100	101	110	111
0	000			DSBH		DSHD			
1	001								
2	010								
3	011								

COP0

	bit23:21	0	1	2	3	4	5	6	7
bit25:24		000	001	010	011	100	101	110	111
0	000	MFC0	DMFC0			MTC0	DMTC0		
1	001	BC0		RDPGPR	MFMC0			WRPGPR	
2	010	CO	CO	CO	CO	CO	CO	CO	CO
3	011	CO	CO	CO	CO	CO	CO	CO	CO

CO

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000		TLBR	TLBWI				TLBWR	
1	001	TLBP							
2	010								
3	011	ERET							DERET
4	100	WAIT							
5	101								
6	110								
7	111								

BC0 bit16

bit17	0	1
0	BC0F	BC0T
1	BC0FL	BC0TL

MFMC0 bit5

	0	1
	DI	EI

表 8. 乗除算命令の実行クロック数

命令	R3000	R4000	R5000	R10000	R4300	R4100	R4700
mult	12	10	5	6	5	1	6-9
multu	12	10	5	7	5	1	6-9
div	35	69	36	35	37	35	42
divu	35	69	36	35	37	35	42
dmult	N/A	22	9	10	8	4	7-10
dmultu	N/A	22	9	11	8	4	7-10
ddiv	N/A	135	68	67	69	67	74
ddivu	N/A	135	68	67	69	67	74

図4．TLBのエントリ形式

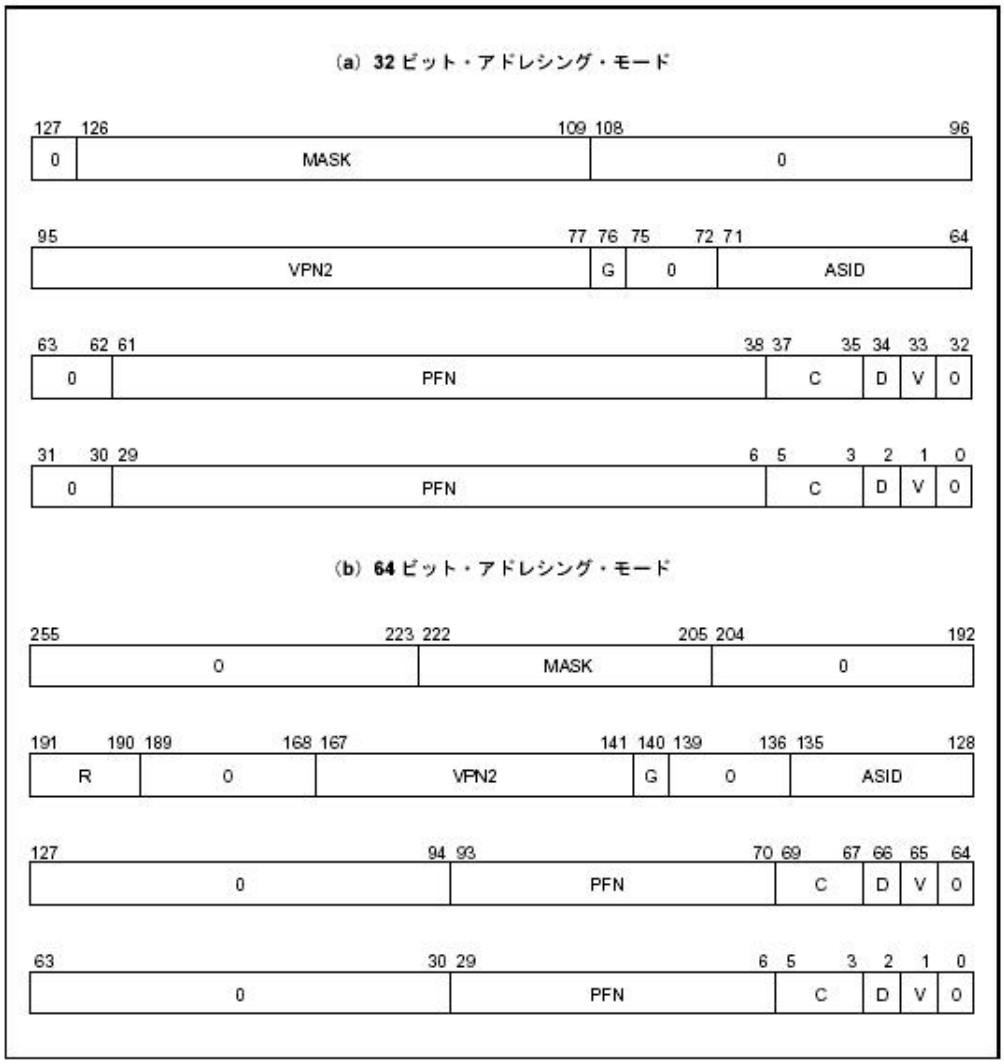
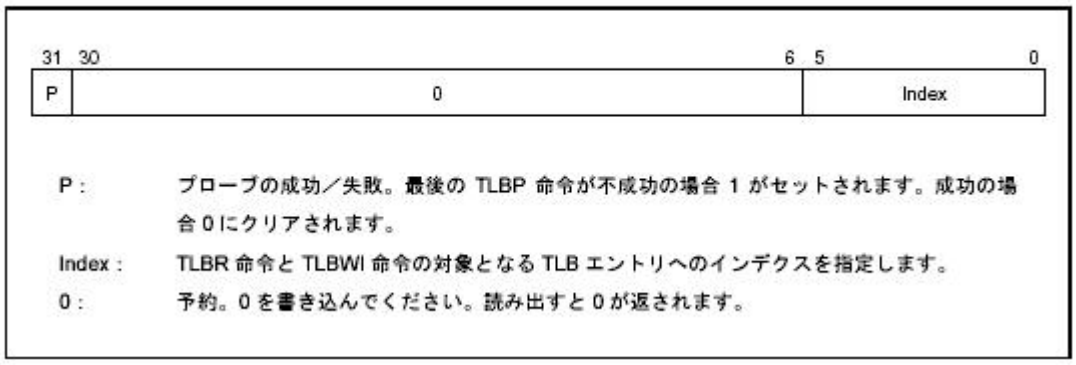


図 5. アドレス変換に必要なレジスタ

(a)インデクスレジスタ



P : プローブの成功／失敗。最後の TLBP 命令が不成功の場合 1 がセットされます。成功の場合 0 にクリアされます。

Index : TLBR 命令と TLBWI 命令の対象となる TLB エントリへのインデクスを指定します。

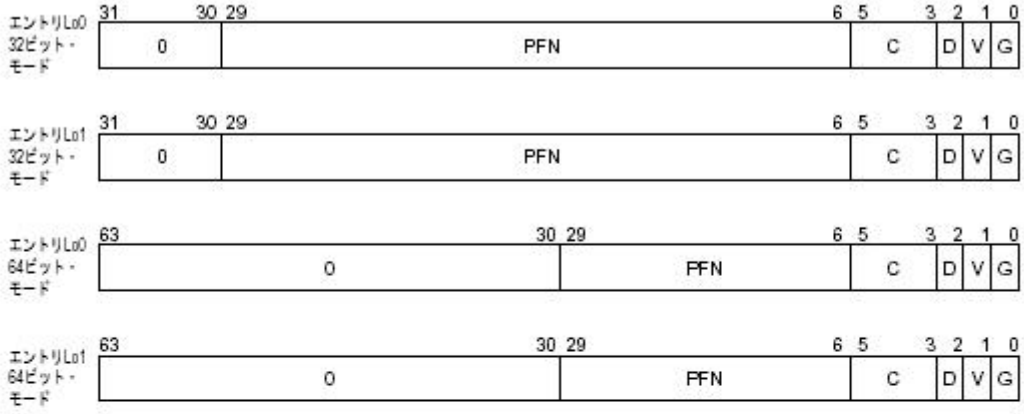
0 : 予約。0 を書き込んでください。読み出すと 0 が返されます。

(b)ランダムレジスタ

Random : TLB ランダム・インデクス。

0 : 予約。0 を書き込んでください。読み出すと 0 が返されます。

(c)エントリLo0，エントリLo1レジスタ



PFN : ページ・フレーム番号。物理アドレスの上位ビット。

C : TLB のページ属性を指定します（表 **5−10** 参照）。

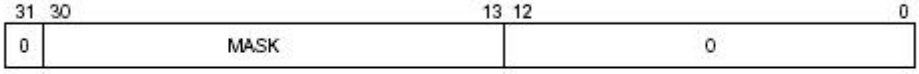
D : Dirty。このビットを 1 にセットするとページは「Dirty」としてマークされ書き込み可能となります。実際このビットは「書き込み保護」ビットとしてデータの変更を防ぐためにソフトウエアが使用します。

V : Valid。このビットを 1 にセットすると TLB エントリが有効であることを示します。V ビットがセットされないままこのエントリにヒットすると，TLB 無効例外（TLBL または TLBS）が発生します。

G : Global。エントリ Lo0，エントリ Lo1 の両方のグローバル・ビットがセットされていると，TLB 参照時に ASID は無視されます。

0 : 予約。0 を書き込んでください。読み出すと 0 が返されます。

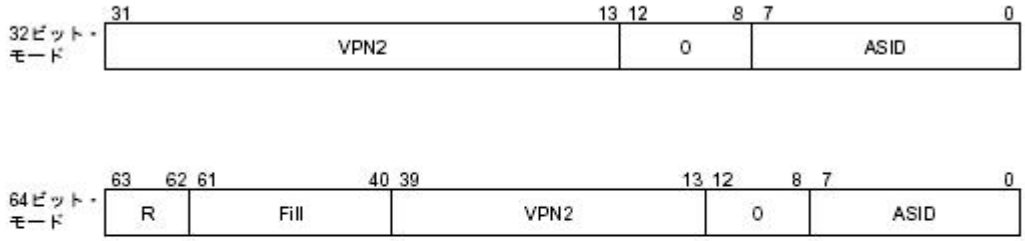
(d)ページマスクレジスタ



MASK : ページ比較マスク。該当するエントリの仮想ページ・サイズを決定します。

0 : 予約。0 を書き込んでください。読み出すと 0 が返されます。

(e)エントリHiレジスタ



VPN2 : 仮想ページ番号を 2 で割った値（2 ページにマッピングされる）。

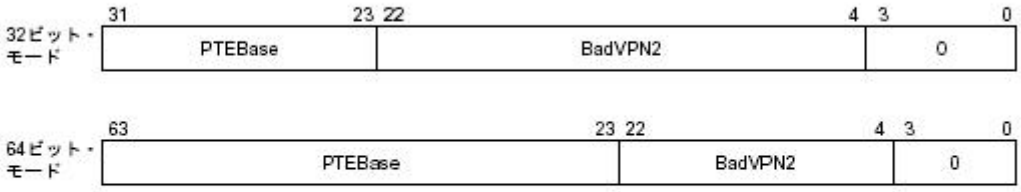
ASID : アドレス空間 ID 領域。8 ビットの ASID 領域は TLB をマルチプロセスで共用できます。各プロセスの仮想アドレスは重なってもかまいません。

R : 空間種別（00→ユーザ，01→スーパバイザ，11→カーネル）。仮想アドレスのビット 63, 62 と一致します。

Fill : 予約。書き込みは無視され，読み出すと 0 が返されます。

0 : 予約。0 を書き込んでください。読み出すと 0 が返されます。

図6. コンテキストレジスタ

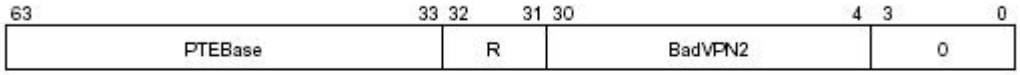


PTEBase： ページ・エントリ・テーブルのベース・アドレス。

BadVPN2： 変換が無効となった仮想アドレスの仮想ページ番号を 2 で割った値。

0： 予約。0 を書き込んでください。読み出すと 0 が返されます。

図7. Xコンテキストレジスタ



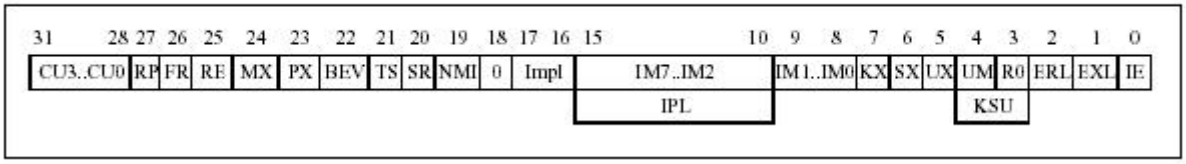
PTEBase： ページ・テーブル・エントリのベース・アドレス。

R： 空間種別（00→ユーザ，01→スーパバイザ，11→カーネル）。仮想アドレスのビット 63, 62 と一致します。

BadVPN2： 変換が無効となった仮想アドレス（ビット 39-13）。

0： 予約。0 を書き込んでください。読み出すと 0 が返されます。

図8. ステータスレジスタ



CU(CU3..CU0) コプロセッサ 3，2，1，0 へのアクセス制御.
CU3 は，従来は MIPS IV 命令の許可ビットだったが，MIPS32/MIPS64 では
MIPS IV 命令という区別がなくなったため，このビットは意味がなくなった.

RP 省電力モードの許可(実装依存).

FR 浮動小数点レジスタの本数を示す．MIPS32/MIPS64 では意味がなくなった.

RE User Mode でのエンディアン反転.

MX MDMX 命令の実行許可.

PX 32 ビットアドレッシング時の User Mode における 64 ビット命令の許可.

BEV Boot Strap Mode 時の割り込み/例外ベクタ指定.

TS TLB Shutdown 状態を表示．TLB の多重ヒットが発生.

SR Soft Rest が入ったことを表示.

NMI NMI が入ったことを表示.

Impl 実装依存.

IM7..IM2 ハードウェア割り込みの許可ビット.

IM1..IM0 ソフトウェア割り込みの許可ビット.

KX Kernel Mode の 64 ビットアドレッシングを指定.

SX Supervisor Mode の 64 ビットアドレッシングを指定.

UX User Mode の 64 ビットアドレッシングを指定.

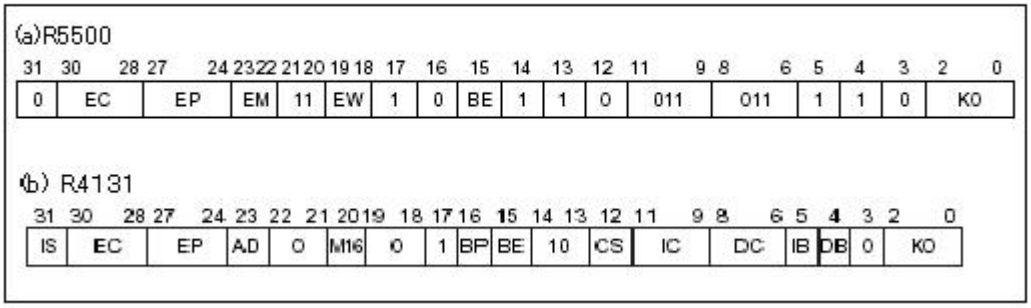
KSU Kernel Mode，Supervisor Mode，User Mode の指定.

ERL Error Level を示す．1 の場合，Kernel Mode と同じ挙動になる.
また，1 の場合は，TLB マップ領域をアドレス変換しない.

EXL Exception Level を示す．1 の場合，Kernel Mode と同じ挙動になる.

IE 割り込み許可を指定.

図9．コンフィグレジスタ（R5000系/R4000系）



主なビットのみ説明.

BE　BigEndian で動作していることを表示.

CS　キャッシュのアドレス計算の単位を表示．1 の場合，2^{10}，0 の場合 2^{12}.

IC　命令キャッシュの容量を表示．CS が 0 の場合は，$2^{(12+IC)}$バイト.

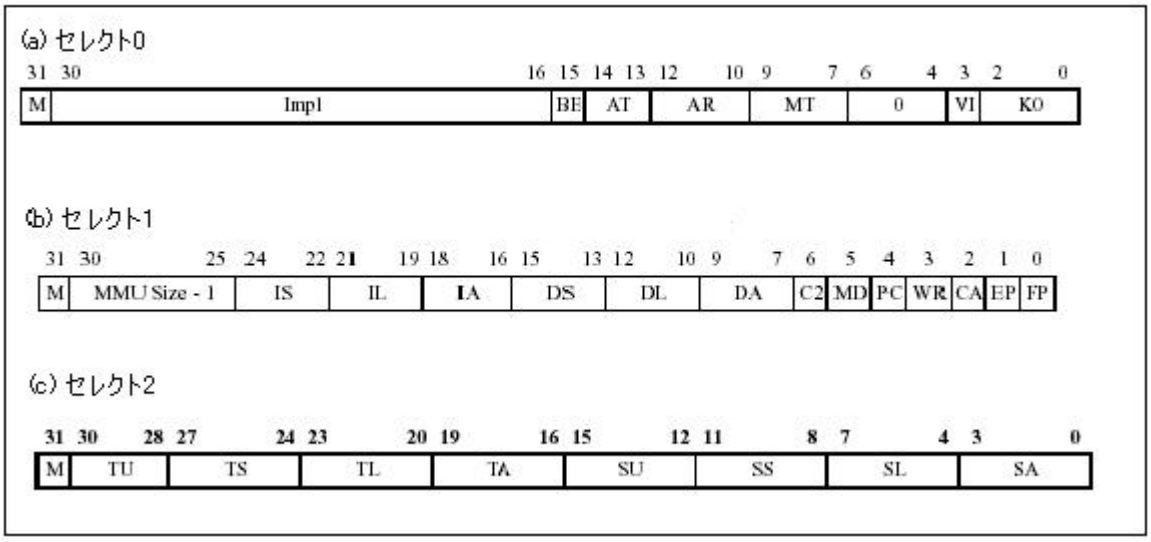
DC　データキャッシュの容量を表示．CS が 0 の場合は，$2^{(12+DC)}$バイト.

IB　命令キャッシュのラインサイズを表示．1 の場合 32 バイト．0 の場合 16 バイト.

DB　データキャッシュのラインサイズを表示. 1 の場合 32 バイト. 0 の場合 16 バイト.

K0　keseg0 のキャッシュアルゴリズムの指定.

図10．コンフィグレジスタ(MIPS32/MIPS64)



主なビットのみ説明.

BE　BigEndian で動作していることを表示.

AT　Architecture type を表示. 0 は MIPS32, 1 は 32 ビットアドレッシングの MIPS64, 2 は MIPS64.

AR　Architecture revision level を表示．0 はリリース 1，1 はリリース 2.

MT　MMU type を表示.

VI　命令キャッシュが仮想アドレスキャッシュかどうかの表示.

K0　keseg0 のキャッシュアルゴリズムの指定.

IS　L1 命令キャッシュの 1 ウェイ当りのセット数．$2^{(IS+6)}$セット.

IL　L1 命令キャッシュのラインサイズ．$2^{(IL+1)}$バイト.

IA　L1 命令キャッシュのウェイ数．(IA+1)ウェイ.

DS　L1 データャッシュの 1 ウェイ当りのセット数．$2^{(DS+6)}$セット.

DL　L1 データキャッシュのラインサイズ．$2^{(DL+1)}$バイト.

DA　L1 データキャッシュのウェイ数．(DA+1)ウェイ.

C2　コプロセッサ 2 が実装されているかどうかを表示.

MD　MDMX 命令が実装されているかどうかを表示.

CA　MIPS16 命令が実装されているかどうかを表示.

FP　FPU が実装されているかどうかを表示.

SS　L2 キャッシュの 1 ウェイ当りのセット数．$2^{(SS+6)}$セット.

SL　L2 データキャッシュのラインサイズ．$2^{(SL+1)}$バイト.

SA　L2 データキャッシュのウェイ数．(SA+1)ウェイ.

TS　L3 キャッシュの 1 ウェイ当りのセット数．$2^{(TS+6)}$セット.

TL　L3 データキャッシュのラインサイズ．$2^{(TL+1)}$バイト.

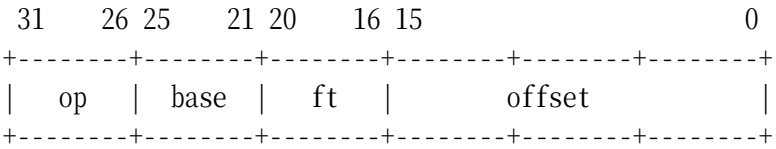
TA　L3 データキャッシュのウェイ数．(TA+1)ウェイ.

表9. キャッシュ命令の機能コード（上位3ビット）

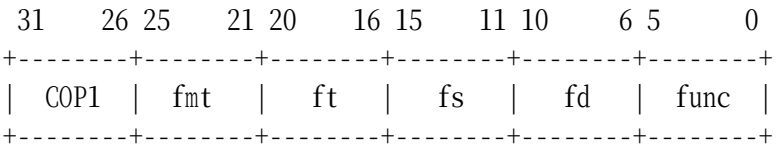
コード	キャッシュ	機能の名称	操作
0	I,SI	Index Invalidate	キャッシュラインの無効化.
0	D,SD	Index Writeback Invalidate	キャッシュラインがダーティならライトバックし無効化.
1	All	Index Load Tag	キャッシュラインのタグの値をタグＬｏレジスタに格納.
2	All	Index Store Tag	タグＬｏレジスタの値をキャッシュラインのタグに設定.
3	D,SD	Create Dirty Exclusive	キャッシュミスのとき，前のラインがダーティならライトバックする．タグを設定し直し状態をダーティにする.
4	All	Hit Invalidate	ヒットするラインの無効化.
5	D,SD	Hit Writeback Invalidate	ヒットするラインがダーティならライトバックし無効化.
5	I	Fill	キャッシュラインに命令を読み込む.
6	I,D,SD	Hit Writeback	命令キャッシュではラインをライトバックする．データキャッシュではラインがダーティならライトバックする.
7	SI,SD	Hit Set Virtual	２次キャッシュの仮想インデクスの変更.

図 11. FPU の命令形式

●I-Type(イミーディエト形式)

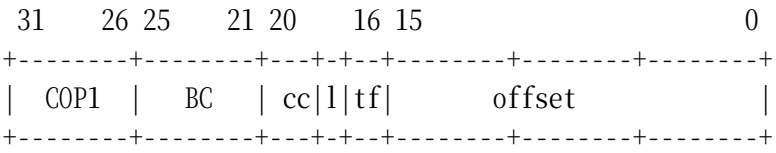


●R-Type(レジスタ形式)

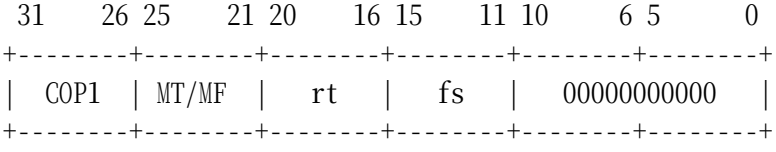


●その他

(1)分岐



(2)CPU, FPU 間データ転送



op：６ビットのオペコード.
COP1：６ビットのオペコード(0x11).
base：５ビットのベースレジスタ.
fmt：５ビットのフォーマット.
　　単精度(0x10)／倍精度(0x11)／
　　32 ビット整数(0x14)／64 ビット整数(0x15).
ft：５ビットのソース(演算，ストア時)，または
　デスティネーション(ロード時)FPU レジスタ.
fs：５ビットのソース FPU レジスタ.
fd：５ビットのデスティネーション FPU レジスタ.
offset：16 ビットの符号付きオフセット.
func：６ビットの関数指定.

BC：５ビットの条件分岐を示すサブオペコード(0x08).
cc：３ビットのコンディションコード番号.
l：１ビットの Branch Likely の指定.
tf：１ビットの true/false の指定.
MT：CPU レジスタ rt の内容を FPU レジスタ fs に転送.

MF：FPU レジスタ fs の内容を CPU レジスタ rt に転送.

図 12. FPU（CP1）のオペコードマップ

COP1									
	bit23:21	0	1	2	3	4	5	6	7
bit25:24		000	001	010	011	100	101	110	111
0	000	MFC1	DMFC1	CFC1	MFHC1	MTC1	DMTC1	CTC1	MTHC1
1	001	BC1	BC1ANY2	BC1ANY4					
2	010	S	D			W	L	PS	
3	011								

S

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L	TRUNC.L	CEIL.L	FOOLR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010		MOVCF	MOVZ	MOVN		RECIP	RSQRT	
3	011					RECIP2	RECIP1	RSQRT1	RSQRT2
4	100		CVT.D			CVT.W	CVT.L	CVT.PS	
5	101								
6	110	C.F CABS.F	C.UN CABS.UN	C.EQ CABS.EQ	C.UEQ CABS.UEQ	C.OLT CABS.OLT	C.ULT CABS.ULT	C.OLE CABS.OLE	C.ULE CABS.ULE
7	111	C.SF CABS.SF	C.NGLE CABS.NGLE	C.SEQ CABS.SEQ	C.NGL CABS.NGL	C.LT CABS.LT	C.NGE CABS.NGE	C.LE CABS.LE	C.NGT CABS.NGT

D

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	ROUND.L	TRUNC.L	CEIL.L	FOOLR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010		MOVCF	MOVZ	MOVN		RECIP	RSQRT	
3	011					RECIP2	RECIP1	RSQRT1	RSQRT2
4	100	CVT.S				CVT.W	CVT.L		
5	101								
6	110	C.F CABS.F	C.UN CABS.UN	C.EQ CABS.EQ	C.UEQ CABS.UEQ	C.OLT CABS.OLT	C.ULT CABS.ULT	C.OLE CABS.OLE	C.ULE CABS.ULE
7	111	C.SF CABS.SF	C.NGLE CABS.NGLE	C.SEQ CABS.SEQ	C.NGL CABS.NGL	C.LT CABS.LT	C.NGE CABS.NGE	C.LE CABS.LE	C.NGT CABS.NGT

W/L

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010								
3	011								
4	100	CVT.S	CVT.D					CVT.PS.PW	
5	101								
6	110								
7	111								

PS

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	ADD	SUB	MUL			ABS	MOV	NEG
1	001								
2	010		MOVCF	MOVZ	MOVN				
3	011	ADDR		MULR		RECIP2	RECIP1	RSQRT1	RSQRT2
4	100	CVT.S.PU				CVT.PW.PS			
5	101	CVT.S.PL				PLL.PS	PLU.PS	PUL..PS	PUU.PS
6	110	C.F CABS.F	C.UN CABS.UN	C.EQ CABS.EQ	C.UEQ CABS.UEQ	C.OLT CABS.OLT	C.ULT CABS.ULT	C.OLE CABS.OLE	C.ULE CABS.ULE
7	111	C.SF CABS.SF	C.NGLE CABS.NGLE	C.SEQ CABS.SEQ	C.NGL CABS.NGL	C.LT CABS.LT	C.NGE CABS.NGE	C.LE CABS.LE	C.NGT CABS.NGT

BC1	bit16	
bit17	0	1
0	BC1F	BC1T
1	BC1FL	BC1TL

BC1ANY2	bit16	
bit17	0	1
0	BC1ANY2F	BC1ANY2T
1		

BC1ANY4	bit16	
bit17	0	1
0	BC1ANY4F	BC1ANY4T
1		

MOVCF	bit16	
	0	1
	MOVF.fmt	MOVT.fmt

COP1X

	bit2:0	0	1	2	3	4	5	6	7
bit5:3		000	001	010	011	100	101	110	111
0	000	LWXC1	LDXC1				LUXC1		
1	001	SWXC1	SDXC1				SUXC1		PREFX
2	010								
3	011							ALNV.PS	
4	100	MADD.S	MADD.D					MADD.PS	
5	101	MSUB.S	MSUB.D					MSUB.PS	
6	110	NMADD.S	NMADD.D					NMADD.PS	
7	111	NMSUB.S	NMSUB.D					NMSUB.PS	

表 10. FPU 演算命令と動作

表 10-1. FPU ロード／ストア命令

命令	形式と説明	op	base	ft	offset	
Load Word to Floating Point	LWC1 ft,offset(base) 符号拡張した offset を CPU レジスタ base の内容に加算しアドレスを生成する. アドレスから 32 ビットデータをリードし FPU レジスタ ft に格納する.					(MIPS I)
Load Double-word to Floating Point	LDC1 ft,offset(base) 符号拡張した offset を CPU レジスタ base の内容に加算しアドレスを生成する. アドレスから 64 ビットデータをリードし FPU レジスタ ft に格納する.					(MIPS II)
Store Word to Floating Point	SWC1 ft,offset(base) 符号拡張した offset を CPU レジスタ base の内容に加算しアドレスを生成する. FPU レジスタ ft の 32 ビットデータをアドレスにライトする.					(MIPS I)
Store Double-word to Floating Point	SDC1 ft,offset(base) 符号拡張した offset を CPU レジスタ base の内容に加算しアドレスを生成する. FPU レジスタ ft の 64 ビットデータをアドレスにライトする.					(MIPS II)

命令	形式と説明	COP1X	base	index	fs	fd	func	
Load Word Indexed to Floating Point	LWXC1 fd,index(base) CPU レジスタ index と base の内容を加算しアドレスを生成する. アドレスから 32 ビットデータをリードし FPU レジスタ ft に格納する.							(MIPS IV)

Load Double-word Indexed to Floating Point	LDXC1 fd,index(base)	(MIPS IV)
	CPU レジスタ index と base の内容を加算しアドレスを生成する. アドレスから 64 ビットデータをリードし FPU レジスタ ft に格納する.	
Store Word Indexed to Floating Point	SWXC1 fs,index(base)	(MIPS IV)
	CPU レジスタ index と base の内容を加算しアドレスを生成する. FPU レジスタ fs の 32 ビットデータをアドレスにライトする.	
Store Double-word Indexed to Floating Point	SDXC1 fs,index(base)	(MIPS IV)
	CPU レジスタ index と base の内容を加算しアドレスを生成する. FPU レジスタ fs の 64 ビットデータをアドレスにライトする.	

命令	形式と説明	COP1X	base	index	hint	0	func
Prefetch Indexed	PREFX hint,index(base)						(MIPS IV)
	CPU レジスタ index と base の内容を加算しアドレスを生成する. hint が示す情報に従い, アドレスで指定されたデータをキャッシュにプリフェッチする.						

表 10-2. FPU 演算命令

命令	形式と説明	COP1	fmt	ft/cc	fs	fd/cc	func
Add	ADD.fmt fd,fs,ft						(MIPS I)
	FPU レジスタ fs と ft の内容を加算し結果を FPU レジスタ fd に格納する.						

	SUB.fmt　fd,fs,ft	(MIPS I)
Subtract	FPU レジスタ fs の内容から ft の内容を減算し結果を FPU レジスタ fd に格納する.	
	MUL.fmt　fd,fs,ft	(MIPS I)
Multiply	FPU レジスタ fs と ft の内容を乗算し結果を FPU レジスタ fd に格納する.	
	DIV.fmt　fd,fs,ft	(MIPS I)
Divide	FPU レジスタ fs の内容を ft の内容で除算し結果を FPU レジスタ fd に格納する.	
	SQRT.fmt　fd,fs	(MIPS II)
Square Root	FPU レジスタ fs の内容の平方根を求め結果を FPU レジスタ fd に格納する.	
	ABS.fmt　fd,fs	(MIPS I)
Absolute Value	FPU レジスタ fs の内容の絶対値を求め結果を FPU レジスタ fd に格納する.	
	MOV.fmt　fd,fs	(MIPS I)
Move	FPU レジスタ fs の内容を FPU レジスタ fd に格納する.	
	NEG.fmt　fd,fs	(MIPS I)
Negate	FPU レジスタ fs の内容を符号反転して結果を FPU レジスタ fd に格納する.	
	MOVT.fmt　fd,fs,cc	(MIPS IV)
※1	浮動小数点条件 cc が 1 なら, FPU レジスタ fs の内容を FPU レジスタ fd に格納する.	
	MOVF.fmt　fd,fs,cc	(MIPS IV)
※2	浮動小数点条件 cc が 0 なら, FPU レジスタ fs の内容を FPU レジスタ fd に格納する.	
	MOVZ.fmt　fd,fs,rt	(MIPS IV)
※3	CPU レジスタ rt の値が 0 なら, FPU レジスタ fs の内容を FPU レジスタ fd に格納する.	
	MOVN.fmt　fd,fs,rt	(MIPS IV)
※4	CPU レジスタ rt の値が 0 以外なら, FPU レジスタ fs の内容を FPU レジスタ fd に格納する.	
	RECIP.fmt　fd,fs	(MIPS IV)

※5	FPU レジスタ fs の逆数を FPU レジスタ fd に格納する.	
※6	RSQRT.fmt fd,fs (MIPS IV) FPU レジスタ fs の平方根を計算し，その逆数を FPU レジスタ fd に格納する.	
※7	ROUND.L.fmt fd,fs (MIPS III) FPU レジスタ fs の内容をもっとも近い 64 ビット整数に変換し，結果を fd に格納する.	
※8	TRUNC.L.fmt fd,fs (MIPS III) FPU レジスタ fs の内容を小数点以下を無視して 64 ビット整数に変換し，結果を fd に格納する.	
※9	CEIL.L.fmt fd,fs (MIPS III) FPU レジスタ fs の内容を+∞方向に丸め，64 ビット整数に変換し，結果を fd に格納する.	
※10	FLOOR.L.fmt fd,fs (MIPS III) FPU レジスタ fs の内容を-∞方向に丸め，64 ビット整数に変換し，結果を fd に格納する.	
※11	ROUND.W.fmt fd,fs (MIPS II) FPU レジスタ fs の内容をもっとも近い 32 ビット整数に変換し，結果を fd に格納する.	
※12	TRUNC.W.fmt fd,fs (MIPS II) FPU レジスタ fs の内容を小数点以下を無視して 32 ビット整数に変換し，結果を fd に格納する.	
※13	CEIL.W.fmt fd,fs (MIPS II) FPU レジスタ fs の内容を+∞方向に丸め，32 ビット整数に変換し，結果を fd に格納する.	
※14	FLOOR.W.fmt fd,fs (MIPS II) FPU レジスタ fs の内容を-∞方向に丸め，32 ビット整数に変換し，結果を fd に格納する.	
※15	CVT.S.fmt fd,fs (MIPS I,III) FPU レジスタ fs の内容を単精度浮動小数点データに変換し，結果を fd に格納する.	
※16	CVT.D.fmt fd,fs (MIPS I,III) FPU レジスタ fs の内容を倍精度浮動小数点データ	

	に変換し，結果を fd に格納する．	
	CVT.L.fmt　fd,fs　(MIPS III)	
※17	FPU レジスタ fs の内容を 64 ビット整数データ	
	に変換し，結果を fd に格納する．	
	CVT.W.fmt　fd,fs　(MIPS I)	
※18	FPU レジスタ fs の内容を 32 ビット整数データ	
	に変換し，結果を fd に格納する．	
	C.cond.fmt　fs,ft　(MIPS I)	
Compare	FPU レジスタ fs の内容と FPU レジスタ ft の内容を比較し	
	結果を cc0 に設定する．	
	C.cond.fmt　cc,fs,ft　(MIPS IV)	
	FPU レジスタ fs の内容と FPU レジスタ ft の内容を比較し	
	結果を指定された cc に設定する．	

命令	形式と説明	COP1X	fr	ft	fs	fd	func
Multiply Add	MADD.fmt　fd,fr,fs,ft　(MIPS IV) FPU レジスタ fs と ft の内容を乗算し，結果に FPU レジスタ fr の内容を加算し，結果を FPU レジスタ fd に格納する．						
Multiply Subtract	MSUB.fmt　fd,fr,fs,ft　(MIPS IV) FPU レジスタ fs と ft の内容を乗算し，結果から FPU レジスタ fr の内容を減算し，結果を FPU レジスタ fd に格納する．						
Negative Multiply Add	NMADD.fmt　fd,fr,fs,ft　(MIPS IV) FPU レジスタ fs と ft の内容を乗算し，結果に FPU レジスタ fr の内容を加算し，符号反転の結果を FPU レジスタ fd に格納する．						
Negative Multiply Subtract	NMSUB.fmt　fd,fr,fs,ft　(MIPS IV) FPU レジスタ fs と ft の内容を乗算し，結果から FPU レジスタ fr の内容を減算し，符号反転の結果を FPU レジスタ fd に格納する．						

※１　Move Conditional on Floating Point True
※２　Move Conditional on Floationg Point False
※３　Move Conditional on Zero
※４　Move Conditional on Not Zero
※５　Reciprocal Approximation
※６　Reciprocal Square Root Approximation

※7　Round to Long Fixed Point
※8　Truncate to Long Fixed Point
※9　Ceiling Convert to Long Fixed Point
※10　Flooring Convert to Long Fixed Point
※11　Round to Word Fixed Point
※12　Truncate to Word Fixed Point
※13　Ceiling Convert to Word Fixed Point
※14　Flooring Convert to Word Fixed Point
※15　Convert to Single Floating Point
※16　Convert to Double Floating Point
※17　Convert to Long Fixed Point
※18　Convert to Word Fixed Point

表 10-3. FPU 分岐命令

命令	形式と説明	COP1	BC1	cc	offset	
Branch on FP True	BC1T offset 浮動小数点条件ビット cc0 が１なら１命令遅れて分岐先へ分岐する.					(MIPS I)
	BC1T cc,offset 指定された浮動小数点条件ビット cc が１なら１命令遅れて分岐先へ分岐する.					(MIPS IV)
Branch on FP True Likely	BC1TL offset 浮動小数点条件ビット cc0 が１なら１命令遅れて分岐先へ分岐する. cc0 が０なら遅延スロットを実行しない.					(MIPS I)
	BC1TL cc,offset 指定された浮動小数点条件ビット cc が１なら１命令遅れて分岐先へ分岐する. cc が０なら遅延スロットを実行しない.					(MIPS IV)
Branch on FP False	BC1F offset 浮動小数点条件ビット cc0 が０なら１命令遅れて分岐先へ分岐する.					(MIPS I)
	BC1F cc,offset 指定された浮動小数点条件ビット cc が０なら１命令遅れて分岐先へ分岐する.					(MIPS IV)
Branch on FP	BC1FL offset 浮動小数点条件ビット cc0 が０なら１命令遅れて分岐先へ					(MIPS I)

|False | 分岐する．cc0 が１なら遅延スロットを実行しない． |
|Likely +--+
	BC1FL cc,offset (MIPS IV)
	指定された浮動小数点条件ビット cc が０なら１命令遅れて
	分岐先へ分岐する．cc が１なら遅延スロットを実行しない．
+--------+--+

表 10-4. FPU 転送命令

命令	形式と説明	COP1	func	rt	fs	0
Move Word From Floating Point	MFC1 rt,fs (MIPS I) 32 ビットデータを FPU レジスタ fs から CPU レジスタ rt に転送する．					
Move Word To Floating Point	MTC1 rt,fs (MIPS I) 32 ビットデータを CPU レジスタ rt から FPU レジスタ fs に転送する．					
Move Double- Word From Floating Point	DMFC1 rt,fs (MIPS III) 64 ビットデータを FPU レジスタ fs から CPU レジスタ rt に転送する．					
Move Double- Word To Floating Point	DMTC1 rt,fs (MIPS III) 64 ビットデータを CPU レジスタ rt から FPU レジスタ fs に転送する．					
Move Control Word From Floating Point	CFC1 rt,fs (MIPS III) FPU 制御レジスタ fs の内容を CPU レジスタ rt に転送する．					

Move Control Word To Floating Point	CTC1 rt,fs	(MIPS III)
	CPU レジスタ rt の内容を FPU 制御レジスタ fs に転送する.	

表 11. 比較演算の条件と条件ビットにセットされる値

条件(cond)	関係				Unordered のとき無効演算例外を発生するか
	＞	＜	＝	Unordered	
F	0	0	0	0	
UN	0	0	0	1	
EQ	0	0	1	0	
UEQ	0	0	1	1	しない.
OLT	0	1	0	0	
ULT	0	1	0	1	
OLE	0	1	1	0	
ULE	0	1	1	1	
SF	0	0	0	0	
NGLE	0	0	0	1	
SEQ	0	0	1	0	
NGL	0	0	1	1	する.
LT	0	1	0	0	
NGE	0	1	0	1	
LE	0	1	1	0	
NGT	0	1	1	1	

表 12. MIPS16 のレジスタ

MIPS16 でのエンコード	MIPS32 でのエンコード	記号	説明
0	16	s0	汎用レジスタ.
1	17	s1	汎用レジスタ.
2	2	v0	汎用レジスタ.
3	3	v1	汎用レジスタ.
4	4	a0	汎用レジスタ.
5	5	a1	汎用レジスタ.
6	6	a2	汎用レジスタ.
7	7	a3	汎用レジスタ.
---	24	t8	条件コードレジスタ. BTEQZ, BTNEZ, CMP, CMPI, SLT, SLTU, SLTI,SLTIU 命令で暗黙的に参照される.
---	29	sp	スタックポインタ.
---	31	ra	リターンアドレスレジスタ.
---	---	PC	プログラムカウンタ.
---	---	HI	乗除算結果の格納用. 上位, 剰余.
---	---	LO	乗除算結果の格納用. 下位, 商.

図13．MIPS16の命令形式

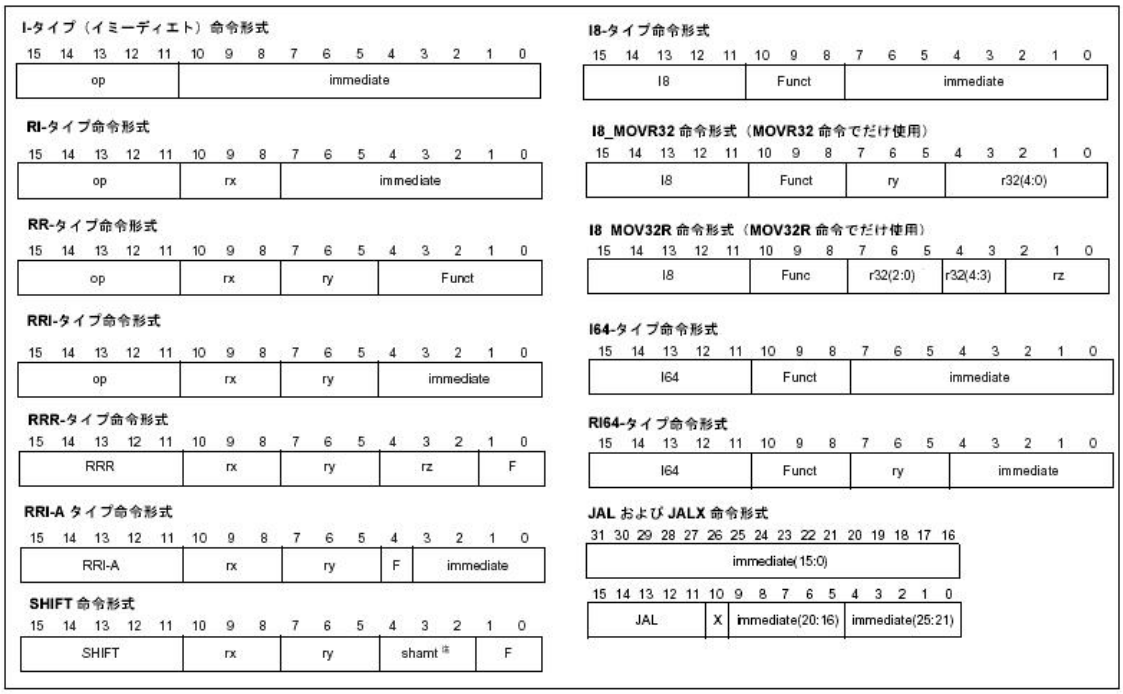


図14．EXTENDされた命令形式

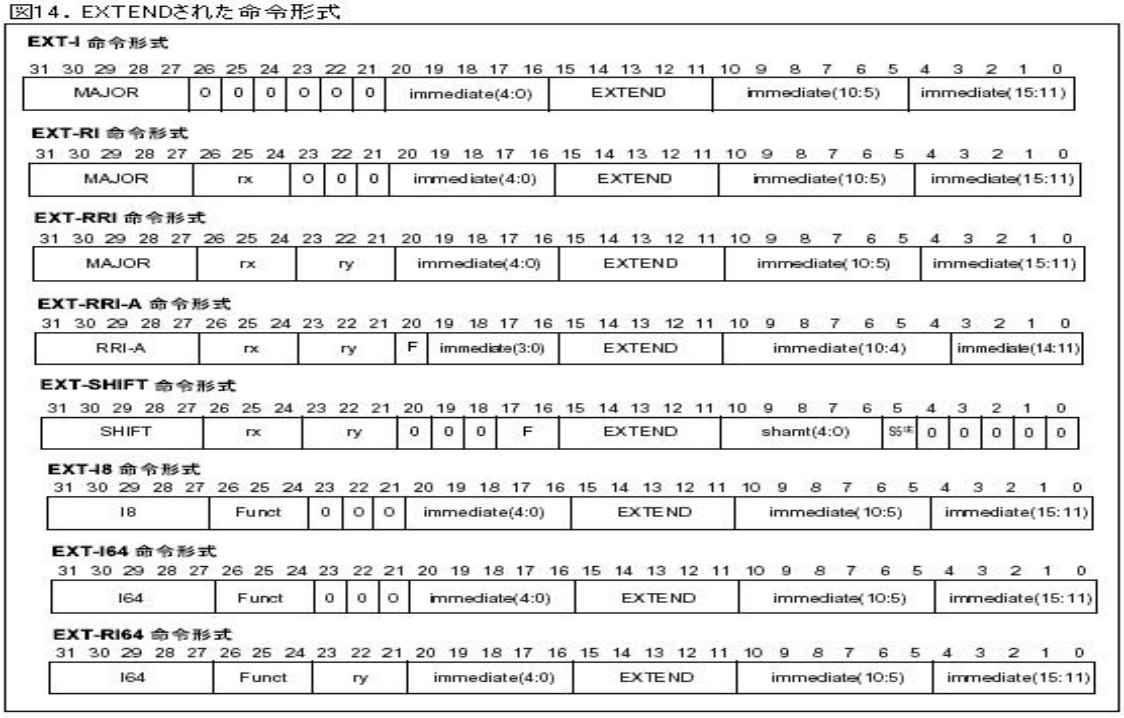


表 13-1. ロード／ストア命令

命令	形式
Load Byte	LB ry,offset(rx)
Load Byte Unsigned	LBU ry,offset(rx)
Load Halfword	LH ry,offset(rx)
Load Halfword Unsigned	LHU ry,offset(rx)
Load Word	LW ry,offset(rx)
	LW rx,offset(pc)
	LW rx,offset(sp)
Load Word Unsigned	LWU ry,offset(rx)
Load Doubleword	LD ry,offset(rx)
	LD ry,offset(pc)
	LD ry,offset(sp)
Store Byte	SB ry,offset(rx)
Store Halfword	SH ry,offset(rx)
Store Word	SW ry,offset(rx)
	SW rx,offset(sp)
	SW ra,offset(sp)
Store Doubleword	SD ry,offset(rx)
	SD ry,offset(sp)
	SD ra,offset(sp)

表 13-2. ALU イミーディエト命令

命令	形式
Load Immediate	LI rx,immediate
Add Immediate Unsigned	ADDIU ry,rx,immediate
	ADDIU rx,immediate
	ADDIU sp,immediate
	ADDIU rx,pc,immediate
	ADDIU rx,sp,immediate
Doubleword Add Immediate Unsigned	DADDIU ry,rx,immediate
	DADDIU ry,immediate
	DADDIU sp,immediate
	DADDIU ry,pc,immediate
	DADDIU ry,sp,immediate
Set on Less Than Immediate	SLTI rx,immediate
Set on Less Than Immediate Unsigned	SLTIU rx,immediate
Compare Immediate	CMPI rx,immediate

表 13-3. 2, 3 オペランド命令

命令	形式
Add Unsigned	ADDU rz,rx,ry
Subtract Unsigned	SUBU rz,rx,ry
Doubleword Add Unsigned	DADDU rz,rx,ry
Doubleword Subtract Unsigned	DSUBU rz,rx,ry
Set on Less Than	SLT rx,ry
Set on Less Than Unsigned	SLTU rx,ry
Compare	CMP rx,ry
Negate	NEG rx,ry
AND	AND rx,ry
OR	OR rx,ry
Exclusive OR	XOR rx,ry
NOT	NOT rx,ry
Move	MOVE ry,r32

表 13-4. シフト命令

命令	形式
Shift Left Logical	SLL rx,ry,immediate
Shift Right Logical	SRL rx,ry,immediate
Shift Right Arithmetic	SRA rx,ry,immediate
Shift Left Logical Variable	SLLV ry,rx
Shift Right Logical Variable	SRLV ry,rx
Shift Right Arithmetic Variable	SRAV ry,rx
Doubleword Shift Left Logical	DSLL rx,ry,immediate
Doubleword Shift Right Logical	DSRL ry,immediate
Doubleword Shift Right Arithmetic	DSRA ry,immediate
Doubleword Shift Left Logical Variable	DSLLV ry,rx
Doubleword Shift Right Logical Variable	DSRLV ry,rx
Doubleword Shift Right Arithmetic Variable	DSRAV ry,rx

表 13-5. 乗除算命令

命令	形式
Multiply	MULT rx,ry
Multiply Unsigned	MULTU rx,ry
Divide	DIV rx,ry
Divide Unsigned	DIVU rx,ry
Move From HI	MFHI rx
Move From LO	MFLO rx
Doubleword Multiply	DMULT rx,ry
Doubleword Multiply Unsigned	DMULTU rx,ry
Doubleword Divide	DDIV rx,ry
Doubleword Divide Unsigned	DDIVU rx,ry

表 13-6. ジャンプ／分岐命令

命令	形式
Jump And Link	JAL target
Jump And Link Exchange	JALX target
Jump Register	JR rx
	JR ra
Jump And Link Register	JALR rx
Branch on Equal to Zero	BEQZ rx,immediate
Branch on Not Equal to Zero	BNEZ rx,immediate
Branch on T Equal to Zero	BTEQZ rx,immediate
Branch on T Not Equal to Zero	BTNEZ rx,immediate
Branch Unconditional	B immediate

表 13-7. 特殊命令

命令	形式
Breakpoint	BREAK immediate
Extend	EXTEND immediate

表 14. リリース 2 で特徴的な命令

命令	形式と説明	COP0	MFMC0	rt	012	0	sc	0
Disable Interrupts	DI rt ステータスレジスタの値をレジスタ rt に格納し，IE ビットを 0(割り込み禁止)にする.							
Enable Interrupts	EI rt ステータスレジスタの値をレジスタ rt に格納し，IE ビットを 1(割り込み許可)にする.							

命令	形式と説明	SPEC.3	rs	rt	msb	lsb	func
Extract Bit Field	EXT rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する.						
Double-word Extract Bit Field	DEXT rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する.						
Double-word Extract Bit Field Middle	DEXTM rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する．size が 32 ビットより大きい場合に使用する.						
Double-word Extract Bit Field Upper	DEXTU rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する．pos が 32 ビットより大きい場合に使用する.						
Insert	INS rt,rs,pos,size						

Bit Field	レジスタ rs の内容の下位 size ビットの領域を，レジスタ rt の pos ビットから size ビットの領域に挿入する．
Double-word Insert Bit Field	DINS rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する．
Double-word Insert Bit Field Middle	DINSM rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する．pos が 32 ビットより小さく，(pos+size)が 32 ビットより大きい場合に使用する．
Double-word Insert Bit Field Upper	DINSU rt,rs,pos,size レジスタ rs の内容のビット pos から size ビットの領域を取り出し，レジスタ rt に格納する．pos が 32 ビット以上の場合に使用する．

命令	形式と説明	SPEC.3	0	rt	rd	sub	func
Word Swap Bytes within Halfwords	WSBH rd,rt レジスタ rt の内容の下位 32 ビットを上位 16 ビット，下位 16 ビットの範囲でバイトスワップを行い，ワードで符号拡張してレジスタ rd に格納する．						
Double-word Swap Bytes within Halfwords	DSBH rd,rt レジスタ rt の内容に対して，ハーフワードごとにバイトスワップを行い，レジスタ rd に格納する．						
Double-word Swap Halfwords	DSHD rd,rt レジスタ rt の内容に対して，ハーフワードスワップを行い，レジスタ rd に格納する．						

within Double-words		
Sign-Extend Byte	SEB rd,rt	レジスタ rt の内容をバイト位置で符号拡張を行い，結果をレジスタ rd に格納する．
Sign-Extend Halfword	SEH rd,rt	レジスタ rt の内容をハーフワード位置で符号拡張を行い，結果をレジスタ rd に格納する．

命令	形式と説明	op	rs	rt	rd	sa	func
Rotate Word Right	ROTR rd,rt,sa						
	レジスタ rt の内容の下位 32 ビットを sa ビットだけ右にローテートする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．						
Rotate Word Right Variable	ROTRV rd,rt,rs						
	レジスタ rt の内容の下位 32 ビットをレジスタ rs の内容の下位 5 ビットで示される値だけ右にローテートする．シフト結果をワードで符号拡張して，レジスタ rd に格納する．						
Double-word Rotate Right	DROTR rd,rt,sa						
	レジスタ rt の内容を sa ビットだけ右にローテートする．シフト結果をレジスタ rd に格納する．						
Double-word Rotate Right+32	DROTR32 rd,rt,sa						
	レジスタ rt の内容を(sa+32)ビットだけ右にローテートする．シフト結果をレジスタ rd に格納する．						
Double-word Rotate Right Variable	DROTRV rd,rt,rs						
	レジスタ rt の内容をレジスタ rs の内容の下位 6 ビットで示される値だけ右にローテートする．シフト結果をレジスタ rd に格納する．						

命令	形式と説明	op	rs	rt	rd	sa	func

命令	形式と説明	op	base	synci	offset	
Sync ICache to Make Write Effective	SYNCI offset(base)					
	16 ビットの offset を符号拡張し，レジスタ base と加算して仮想アドレスを計算する．そのアドレスに位置する命令を書き換えた場合に，書き換え結果を有効にする．					

表 15. MIPS-3D で拡張された命令

ニーモニック	処理
ADDR	浮動小数点リダクション加算. 組どうしの加算.
MULR	浮動小数点リダクション乗算. 組どうしの乗算.
RECIP1	逆数. 高速近似. 精度的には劣る.
RECIP2	逆数. 第２ステップ. 精度を上げる処理.
RSQRT1	平方根の逆数. 高速近似. 精度的には劣る.
RSQRT2	平方根の逆数. 第２ステップ. 精度を上げる処理.
CVT.PS.PW	ペアドワードからペアドシングルへの型変換.
CVT.PW.PS	ペアドシングルからペアドワードへの型変換.
CABS	浮動小数点絶対値比較.
BC1ANY2F	2 つの条件コードのどれかが偽なら分岐.
BC1ANY2T	2 つの条件コードのどれかが真なら分岐.
BC1ANY4F	4 つの条件コードのどれかが偽なら分岐.
BC1ANY4T	4 つの条件コードのどれかが真なら分岐.

表 16. MIPS16e で追加された命令

ニーモニック	処理
JRC	遅延スロットを実行しないレジスタ間接ジャンプ.
JALRC	遅延スロットを実行しないレジスタ間接関数コール.
SEB	バイト位置での符号拡張.
SEH	ハーフワード位置での符号拡張.
ZEB	バイト位置でのゼロ拡張.
ZEH	ハーフワード位置でのゼロ拡張.
RESTORE	ra,s0-s7,a0-a4(全部, 一部)をスタックに退避し, スタックポインを補正する.
SAVE	ra,s0-s7,a0-a4(全部, 一部)をスタックから回復し, スタックポインを補正する.
ASMACRO	実装依存のマクロ命令を実行する.