SPIM S20: MIPS R2000 シミュレーター*

"タダで 25 分の 1 の性能を実現"

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706, USA
608-262-9519

日本語訳: 角川裕次 (広島大学大学院工学研究科情報工学専攻)

1 SPIM

SPIM S20 は MIPS R2000/R3000 RISC 計算機の動作を模倣し、プログラムの実行を行う[1]。SPIM はアセンブリ言語で書かれているプログラムファイルあるいは MIPS 計算機で生成された a.out 形式の実行可能バイナリファイルを読み込み、すぐに実行を行うことができる。SPIM だけでこれらの形式のプログラムが実行可能であり、デバッガやオペレーティングシステムへのインターフェースも含んでいる。

MIPS 計算機は単純で規則的な内部構造をしているので学習し理解するのは容易である。プロセッサは 32 ビットの汎用レジスタを持ち、コンパイラのコード生成で便利なように注意深く命令集合が定められている。

しかし高速で実物の MIPS ワークステーションを多くの人が所有しているのになぜシミュレーターを使用するのか、という疑問が出るかもしれない。ひとつの理由は、それらのワークステーションはどこでも利用可能とは限らないということである。もうひとつの理由は、高速な計算機が次々と新しく出てくるために実際の MIPS 計算機が長年使われ続ける事はないからである。

さらにシミュレーターでは、実際の計算機よりも低レベルプログラミングのための良い環境を提供できるという理由もある。というのも、様々な誤りを検出したり、実際の計算機よりも多く

*講義で SPIM を使用し、教授の書いたプログラムコードのバグを喜んで見つけてくれたウィスコンシン大学の多くの学生達に感謝する。特に 1990 年春に開講された CS536 を受講し、「デバッグ済み」なシミュレーターの最後のいくつかのバグを忍耐強く見つけ出してくれた学生達に感謝する。私は彼らの忍耐と根気に感謝する。アラン・イエンウイ・シオウ氏は X Window System 版を書いてくれた。

[1]実際の MIPS 計算機の説明については Gerry Kane と Joe Heinrich の著書「MIPS RISC Architecture」(1992 年 Prentice Hall 刊) を参照すること。

の機能を提供することが可能だからである。例えば SPIM は X Window System でのインターフェースを持ち、これは実際の計算機上でのデバッガよりも優れている。

結論を言えば、計算機やその上で動くプログラムを学ぶのにシミュレーターは便利な道具である。なぜならシミュレーターはシリコンではなくソフトウエアで作られているので新たな機能を追加するための変更は簡単であり、新たにマルチプロセッサシステムを構築したり単にデータ収集するように変更するのも簡単である。

1.1 仮想機械の模倣

MIPS 計算機の構造は他の多くの RISC 型計算機と似ており、遅延分岐、遅延ロード、アドレッシングモードの制限等のために直接的にプログラムを書くのは難しい。高級言語でプログラムが書かれることを前提に設計された事を考えればこれらの困難は欠点とはならない。プログラマに対してではなくコンパイラに対してインターフェースが定められているのである。複雑さの多くは遅延命令から生じている。**遅延分岐** (delayed branch) は実行に 2 サイクルを要する。2 番目のサイクルでは分岐命令のすぐ次に位置している命令が実行される。この命令は、分岐の直前に有益な仕事をする命令にしても良いし、単に `nop` (no operation) 命令にしても構わない。同様に**遅延ロード** (delayed loads) は実行に 2 サイクルを要するので、ロード命令のすぐ次に位置する命令ではメモリよりロードされた値を使うことはできない。

MIPS はこの複雑さを隠すためにアセンブラを使って **仮想機械** (virtual machine) を実現する方法を選択した。仮想計算機には遅延分岐や遅延ロードはなく、実際のハードウエアよりも豊富な命令集合を持つ。アセンブラは命令を再配置し遅延スロットを埋める。さらにいくつかの実際の命令の列に変換される疑似命令を追加している。

標準では豊富な機能を持った仮想機械の動作を SPIM は模倣する。実際のハードウエアを模倣することもできる。仮想機械について説明をするが、実際のハードウエアにはない機能については簡単に言及するにとどめる。説明においては、拡張機械の機能を利用している、MIPS アセンブリ言語のプログラマ (とコンパイラ) の慣例に従って説明する。命令のうちでダガー記号 (†) のついたものは疑似命令である。

1.2 SPIM インターフェース

SPIM は単純な文字端末版と X Window System 版がある。共に機能自体は同じだが X Window System 版は利用が簡単であり、様々な情報を表示してくれる。

`spim` は文字端末版のプログラムであり、`xspim` は X Window System 版のプログラムである。コマンドラインオプションは以下の通りである。

`-bare`
: アセンブラが提供する機能のうち、疑似命令や追加アドレッシングモードを無効にし、素の MIPS 計算機の動作を模倣する。このオプションを指定すると `-quiet` も指定したと見做される。

`-asm`
: アセンブラによって提供される機能を有効にし、仮想的な MIPC 計算機を模倣する。通常はこの機能が有効となる。

`-pseudo`
アセンブリコード中に疑似命令を書くことを許可する。

`-nopseudo`
アセンブリコード中に疑似命令を書くことを禁止する。

`-notrap`
標準トラップハンドラを読み込まない。このトラップハンドラには、利用者のプログラムで仮定しないといけない二つの関数がある。最初にトラップハンドラはトラップを処理する。トラップが起ると SPIM は位置 0x80000080 にジャンプする。この位置には例外処理コードを置かないといけない。次に、トラップハンドラのファイルには `main` ルーチンを呼び出す起動コードを含んでいる。トラップハンドラがなければ `__start` とラベルづけされた場所より実行が開始される。

`-trap`
標準トラップハンドラを読み込む。通常はこの機能が有効となる。

`-noquiet`
例外が生じた場合メッセージを表示する。通常はこの機能が有効となる。

`-quiet`
例外が生じてもメッセージは表示しない。

`-nomapped_io`
メモリ割り付け IO 機能を無効にする。(第 5 章を参照せよ。)

`-mapped_io`
メモリ割り付け IO 機能を有効にする。(第 5 章を参照せよ。) 端末から文字入力をするために SPIM システムコール (第 1.5 章参照のこと) を使用するプログラムはメモリ割り付け IO 機能を使うべきでない。

`-file`
指定されたファイルよりアセンブリコードを読み込み、実行を行う。

`-s`*`seg size`* メモリセグメント *seg* の大きさの初期値を *size* バイトに設定する。 各メモリセグメントの名前は `text`, `data`, `stack`, `ktext`, `kdata` である。例えばオプション `-sdata 2000000` により、ユーザーデータセグメントの開始場所を 2,000,000 バイトとする。

`-l`*`seg size`* メモリーセグメント *seg* の大きさの拡大可能な限界値を *size* に設定する。拡大可能なメモリセグメントは `data`, `stack`, `kdata` である。

1.2.1 文字端末インターフェース

文字端末版 (`spim`) には以下のコマンドがある。

`exit`
シミュレーターを終了する。

`read "`*`file`*`"`
: アセンブリ言語で書かれたファイル *file* を SPIM のメモリに読み込む。すでにそのファイルが SPIM に読み込まれている場合は、大局記号が二重定義されるのを防ぐために必ずシステムをクリア (以下の `reinitialize` を参照) しないといけない。

`load "`*`file`*`"`
: `read` と同じ意味である。

`run` *addr*
: プログラムの実行を開始する。アドレス *addr* が指定されればそのアドレスから実行を行う。もし指定されなければ大局記号 `__start` の場所より実行される。なおこの記号は標準トラップハンドラにて定義され、大局記号 `main` で定義されるルーチンを通常の手順に従って呼び出す。

`step` *N*
: プログラムの *N* 個 (数が未指定の場合は 1 つ) の命令を段階的に実行する。命令が実行される毎に命令を表示する。

`continue`
: 段階的に実行をせずにプログラムの実行を継続する。

`print $`*N*
: レジスタ *N* の内容を表示する。

`print $`*fN*
: 浮動小数レジスタ *N* の内容を表示する。

`print` *addr*
: アドレス *addr* のメモリ内容を表示する。

`print_sym`
: 記号表の内容を表示する。つまり (局所的ではない) 大局記号のアドレスが表示される。

`reinitialize`
: メモリとレジスタの内容を消す。

`breakpoint addr`
: アドレス *addr* にブレイクポイントを設定する。*addr* はメモリアドレスでも記号ラベルでもよい。

`delete addr`
: アドレス *addr* でのブレイクポイントを全て削除する。

`list`
: ブレイクポイント全てを表示する。

`.`
: 行の残りの部分をアセンブリ命令としてメモリに格納する。

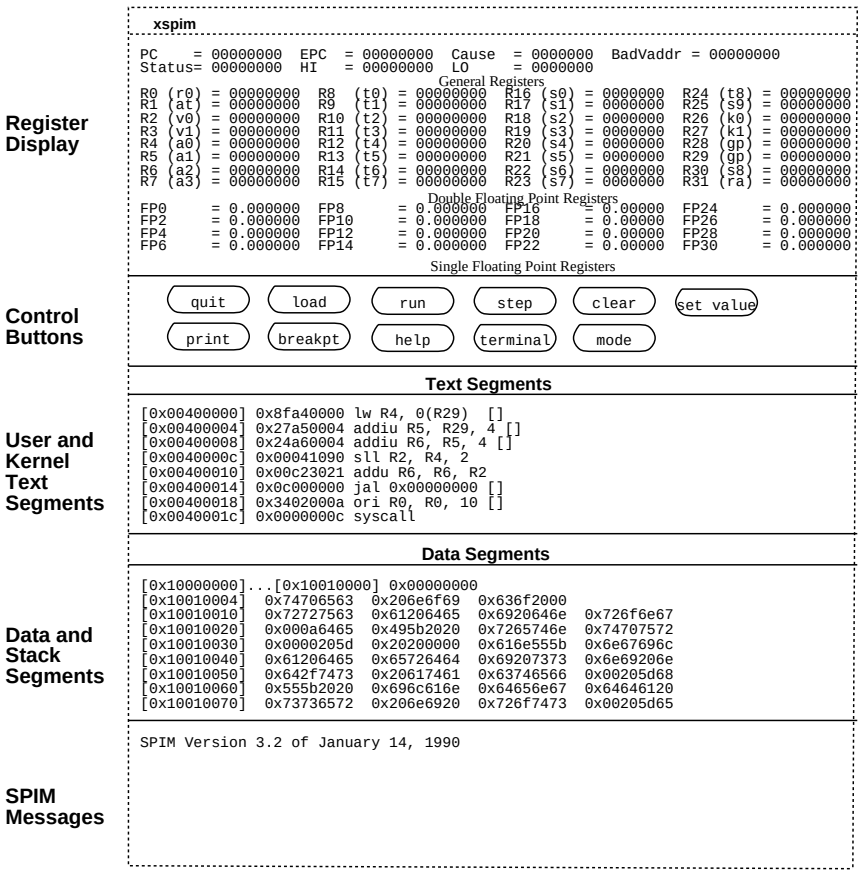

図 1: SPIM の X Window System 版

`<nl>`
: 前回のコマンドを再度実行する。

`?`
: 使用法を表示する。

ほとんどのコマンド名は、例えば `ex`, `re`, `l`, `ru`, `s`, `p` のように、省略して最初の数文字で指定できる。なお `reinitialize` のように危険なコマンドに関してはより長い入力をしないといけない。

1.2.2 X Window System 版

X 版の SPIM のプログラム名は `xspim` であり、見かけは異なるものの文字端末版の `spim` と同じ働きをする。ウインドウは 5 つの表示域より構成される (図 1 参照)。最上部の表示域ではレジスタの内容を表示している。プログラムを実行している間以外は逐次表示が更新される。

その下の表示域ではシミュレーターを制御するボタンが並んでいる。

quit
: シミュレーターを終了する。

load

ソースファイルをメモリーに読み込む。

run
プログラムの実行を開始する。

step
プログラムの実行を 1 段階進める。

clear
レジスタとメモリを再初期化する。

set value
レジスタまたはメモリ位置の値を設定する。

print
レジスタまたはメモリ位置の値を表示する。

breakpoint
ブレイクポイントの設定と解除、ならびにブレイクポイント全ての表示を行う。

help
使用法の表示をする。

terminal
コンソールウインドウの表示と非表示を選択する。

mode
SPIM の動作モードを選択する。

さらに下にある 2 つの表示域にはメモリ内容が表示される。これらの内の上側の表示域にはユーザーセグメントおよびカーネルテキストセグメントでの命令が表示される[2]。テキストセグメントでの最初のいくつかの命令はスタートアップコード (`__start`) で、`argc` と `argv` をレジスタにロードし `main` ルーチンを呼び出す。

下側の表示域ではデータセグメントとスタックセグメントを表示する。プログラムが実行されるに従いこれら 2 つの表示は更新される。

一番下の表示域ではシミュレーターからのメッセージが表示される。ここには実行中のプログラムからの出力は表示されない。プログラムが読み込みや書き込みを行なう場合はコンソールと呼ばれる別のウインドウで入出力が表示される。

1.3 実物と異なる機能

SPIM は忠実に MIPS 計算機の動作を模倣するが SPIM はあくまでシミュレーターであり、いくつかの点で実物の計算機とは同一ではない。SPIM はキャッシュやメモリ遅延は模倣せず、浮動小数点演算や乗算と除算に要する時間も模倣しない。

[2]これらの命令は疑似命令ではなく、実際の MIPS 計算機命令である。SPIM はプログラムでのアセンブラ疑似命令を、1 つから 3 つの MIPS 計算機命令に変換してメモリに格納する。元となる疑似命令それぞれに対し、変換された最初の命令の位置にコメントとして表示される。

他の機能として疑似命令をいくつかの機械命令に展開するが、これは現物の計算機でもそのようになっている。デバッガでの 1 命令実行やメモリ参照では、表示される命令はソースプログラムとは少し異なる。SPIM は遅延スロットを埋めるために命令の配置を変更しないので、疑似命令を含んだ命令集合と実際の命令集合との対応付けはとても単純である。

1.4 アセンブラ構文

アセンブラファイルでのコメントはシャープ記号 (#) で始まる。シャープ記号よりその行の最後まですべてがコメントとして無視される。

識別子 (identifiers) はアルファベット文字、数字、アンダーバー (_)、ドット (.) の並びで、最初が数字でないものをいう。命令としてのオペコード (opcode) は予約語で、正しい識別子ではない。ラベル (label) は行の最初に置き、続けてコロンを置くことで宣言できる。例えば次の通りである。

```
        .data
item:   .word 1
        .text
        .globl main             # 大局的（global）でないといけない
main:   lw $t0, item
```

文字列はダブルクオート (") で囲む。文字列での特殊文字は C 言語での慣習に従う。

改行	\n
タブ	\t
引用符	\"

SPIM では MIPS アセンブラで用意されているアセンブラ指示子の一部が利用可能である。

`.align n`
: 次のデータ (1 つ) を 2^n バイト境界に整合して配置する。例えば `.align 2` により、次のデータ値をワードの境界に整合して配置する。`.align 0` により、指示子 `.data` または `.kdata` が現れるまでの間、指示子 `.half`, `.word`, `.float`, `.double` での自動整合を機能を使わずにデータを配置する。

`.ascii str`
: 文字列をメモリに格納する。なお文字列の最後にはナル文字を置かない。

`.asciiz str`
: 文字列をメモリに格納する。なお文字列の最後にナル文字を置く。

`.byte b1, ..., bn`
: 連続したメモリ・バイトに n 個の値を格納する。

`.data <addr>`
: これ以降の項目をデータセグメントに置くことを指示する。省略可能な引数 $addr$ が与えられると、以降のデータ項目はアドレス $addr$ より始まるアドレスに格納される。

`.double d1, ..., dn`
n 個の倍精度浮動小数点数を連続したメモリ位置に格納する。

`.float f1, ..., fn`
n 個の単精度浮動小数点数を連続したメモリ位置に格納する。

`.globl sym`
記号 `sym` は大局的と宣言する。これにより他のファイルでこの記号の参照が可能となる。

`.half h1, ..., hn`
n 個の 16 ビットの値を連続したメモリ位置に格納する。

`.kdata` *addr*
これ以降の項目をカーネルデータセグメントに置くことを指示する。省略可能な引数 *addr* が与えられると、以降のデータ項目はアドレス *addr* より始まるアドレスに格納される。

`.ktext` *addr*
これ以降の項目をカーネルテキストセグメントに置くことを指示する。SPIM において、これ以降の項目には命令またはワード (以下の `.word` 指示子を参照せよ) だけが許されている。 省略可能な引数 *addr* が与えられると、以降のデータ項目はアドレス *addr* より始まるアドレスに格納される。

`.space n`
現在のセグメントにおいて n バイトの領域を確保する。(なお SPIM では現在のセグメントはデータセグメントでないといけない。)

`.text` *addr*
これ以降の項目をユーザーテキストセグメントに置くことを指示する。SPIM ではこれ以降の項目には、命令またはワード (以下の `.word` 指示子を参照せよ) だけが許されている。 省略可能な引数 *addr* が与えられると以降のデータ項目はアドレス *addr* より始まるアドレスに格納される。

`.word w1, ..., wn`
n 個の 32 ビットの値を連続したメモリ位置に格納する。

1.5 システムコール

オペレーティングシステムに似た幾つかのサービスを SPIM は提供しており、システムコール命令 (`syscall`) を通じて利用可能である。あるサービスを要求するにはシステムコール番号 (表 1 参照) をレジスタ `$v0` に設定し、引数を レジスタ `$a0`...`$a3` (浮動小数点数の場合はレジスタ `$f12`) に設定する。実行結果を返すシステムコールでは、レジスタ `$v0` (浮動小数点数の場合はレジスタ `$f0`) に結果が入れられる。例えば「`the answer = 5`」を表示するには以下のようにする。

```
        .data
str:    .asciiz "the answer = "
        .text
```

表 1: システムサービス

サービス	システムコール番号	引数	結果
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer ($v0)
read_float	6		float ($f0)
read_double	7		double ($f0)
read_string	8	$a0 = バッファ, $a1 = 長さ	
sbrk	9	$a0 = メモリ量	アドレス ($v0)
exit	10		

```
li $v0, 4          # print_str のシステムコール番号
la $a0, str        # 表示する文字列のアドレス
syscall            # 文字列を表示する

li $v0, 1          # print_int のシステムコール番号
li $a0, 5          # 表示する整数値
syscall            # 表示する
```

print_int は引数として 1 つの整数を持ち、コンソールにその値を表示する。print_float は単精度の浮動小数点数を表示する。print_double は倍精度の浮動小数点数を表示する。print_string は引数としてナル文字で終端されている文字列へのポインタが与えられ、その文字列をコンソールに表示する。

read_int, read_float, read_double は、入力の一行を全体を改行文字まで読み込み (改行文字も含めて読まれる) んで数を返す。数より後ろにある文字は無視される。read_string は Unix のライブラリ関数 fgets と同じ振る舞いをする。$n-1$ 個までの文字を読み込んでバッファに入れ、文字列の最後にナル文字を置いて文字列を終端する。もし読み込んだ行の文字数が少なければ改行文字までを読み込み、ナル文字で文字列を終端する。**注意:** 端末から読み込みを行うシステムコールを使用するプログラムでは、メモリ割り付け IO を使ってはならない。(第 5 章参照)

sbrk によりメモリ領域の割当てを要求する。n バイト以上のメモリ領域へのポインタが返される。exit によりプログラムの実行を終了する。

2 MIPS R2000 の説明

MIPS プロセッサは整数処理ユニット (CPU) といくつかのコプロセッサより構成され (図 2 参照)、コプロセッサは補助的仕事や浮動小数点数のような他のデータ型の演算を行う。SPIM は 2 つのコプロセッサの動作を模倣する。コプロセッサ 0 はトラップや例外の処理と、仮想記憶システムを取り扱う。SPIM ではトラップや例外機能の多くを模倣するが、メモリシステムについては全く省略している。コプロセッサ 1 は浮動小数点ユニットである。SPIM はこのユニットの機能のほとんどを模倣する。

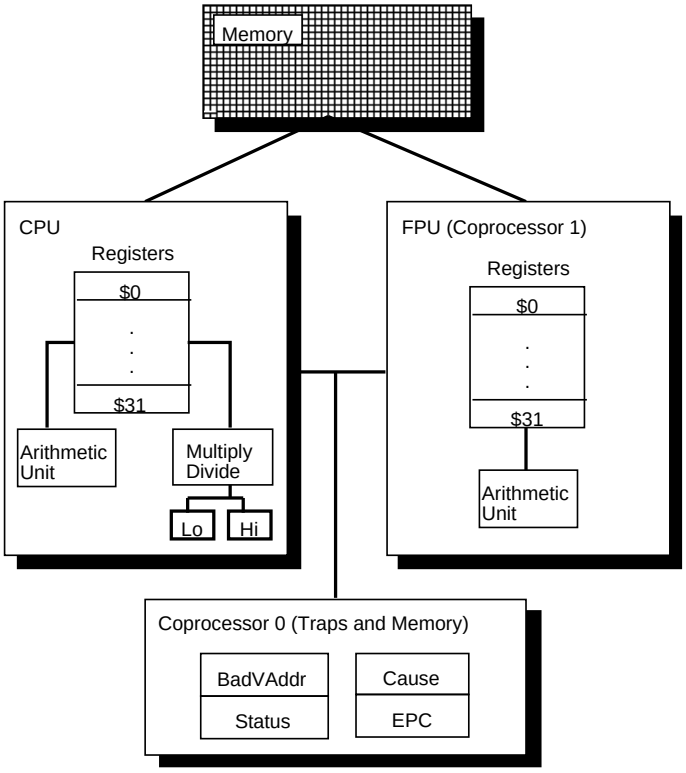

図 2: MIPS R2000 CPU と FPU

2.1 CPU レジスタ

MIPS (および SPIM) CPU には 32 個の汎用 32 ビットレジスタがあり、0–31 まで番号づけされている。 n 番レジスタは `$n` と表記される。レジスタ `$0` は常に値 0 となるようにハードウエアが構成されている。MIPS ではレジスタをどのように使用するかという慣例を定めている。この慣例はあくまでも指針であって、ハードウエアによって強制されている訳ではない。しかしこの慣習を守らないプログラムは、他のソフトウエアと組み合わせて使うことはできない。表 2 にレジスタとその想定された使い方を示す。

レジスタ `$at` (1), `$k0` (26), `$k1` (27) はアセンブラとカーネルだけしか利用してはならない事になっている。

レジスタ `$a0`–`$a3` (4–7) はルーチンへの第一から第四引数を渡すのに使われる。(なお残りの引数はスタックに積んで渡される。) レジスタ `$v0` と `$v1` (2, 3) は手続きの結果を返すのに使われる。レジスタ `$t0`–`$t9` (8–15, 24, 25) は一時的な値を保持するのに使われる。手続きを呼び出すとこれらの値は保存されないので、手続きの呼び出し側が必要に応じて保存しないといけない。レジスタ `$s0`–`$s7` (16–23) は呼び出された手続き側で値の保存を行うレジスタで、手続きを呼び出しても値は変らない。

レジスタ `$sp` (29) はスタックポインタで、現在のスタックの位置を示す[3]。レジスタ`$fp` (30)

[3]初期の SPIM の説明書では `$sp` は (スタックフレームの最後のワードではなく) スタックでの未使用領域を指し示すと書かれていた。最近の SPIM では それは誤りであると明記している。どちらの方法を採用しても同じように問題なく動作するが、実際のシステムの方を選ぶことにした。

レジスタ名	番号	使用法
zero	0	定数 0
at	1	アセンブラ専用
v0	2	式の評価結果と
v1	3	関数の戻り値
a0	4	引数 1
a1	5	引数 2
a2	6	引数 3
a3	7	引数 4
t0	8	一時的データ (関数呼出しで保存されない)
t1	9	一時的データ (関数呼出しで保存されない)
t2	10	一時的データ (関数呼出しで保存されない)
t3	11	一時的データ (関数呼出しで保存されない)
t4	12	一時的データ (関数呼出しで保存されない)
t5	13	一時的データ (関数呼出しで保存されない)
t6	14	一時的データ (関数呼出しで保存されない)
t7	15	一時的データ (関数呼出しで保存されない)
s0	16	一時的なデータ保持 (関数呼出しで保存される)
s1	17	一時的なデータ保持 (関数呼出しで保存される)
s2	18	一時的なデータ保持 (関数呼出しで保存される)
s3	19	一時的なデータ保持 (関数呼出しで保存される)
s4	20	一時的なデータ保持 (関数呼出しで保存される)
s5	21	一時的なデータ保持 (関数呼出しで保存される)
s6	22	一時的なデータ保持 (関数呼出しで保存される)
s7	23	一時的なデータ保持 (関数呼出しで保存される)
t8	24	一時的データ (関数呼出しで保存されない)
t9	25	一時的データ (関数呼出しで保存されない)
k0	26	OS カーネル専用
k1	27	OS カーネル専用
gp	28	大域的データへのポインタ
sp	29	スタックポインタ
fp	30	フレームポインタ
ra	31	戻りアドレス (関数呼出しで使用)

表 2: MIPS レジスタとその利用の慣例

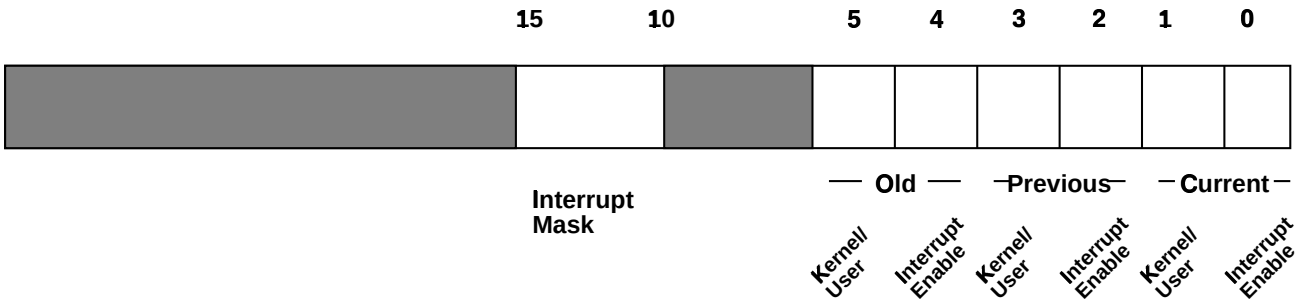

図 3: Status レジスタ

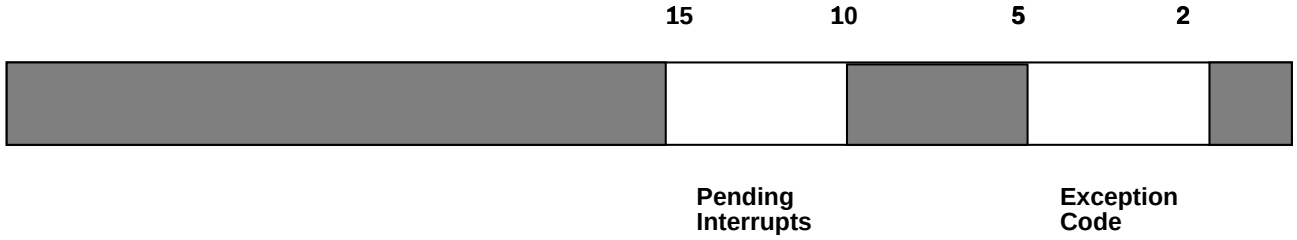

図 4: Cause レジスタ

はフレームポインタである[4]。レジスタ $ra (31) は jal 命令によって手続きを呼び出すときに戻りアドレスが書き込まれる。

レジスタ $gp (28) は大局ポインタで、定数や大局変数が置かれているヒープ中での 64 キロバイトのメモリブロックの真中を指し示している。1 つのロードまたはストア命令だけで、このヒープの中にあるデータを高速に読み書きできる。

さらにコプロセッサ 0 は例外を取り扱うのに便利なレジスタがある。SPIM ではこれらのレジスタのほとんどを実現しているが、シミュレーターではあまり使わないものや実現していないメモリシステムでのレジスタは省略している。実現しているものは以下の通りである。

レジスタ名	番号	使用法
BadVAddr	8	アドレス例外が生じたメモリアドレス
Status	12	割り込み禁止と許可を表すビット
Cause	13	例外の型と未処理割り込みを表すビット
EPC	14	例外を発生させた命令のアドレス

これらのレジスタはコプロセッサ 0 の一部であり、lwc0, mfc0, mtc0, swc0 命令によって設定や読み出しが行なわれる。

図 3 は SPIM で実現されている Status レジスタの各ビットの説明である。interrupt mask には 5 つの各割り込みレベルに対するビットがある。もしビットの値が 1 ならばそのレベルの割り込みが許可される。もし 0 ならばそのレベルの割り込みは禁止される。Status レジスタの下位 6 ビットは kernel/user と interrupt enable ビットに対する 3 段階のスタックを実現している。もし割り込みが生じた時点で実行していたプログラムがカーネルならば kernel/user ビットは 0 になる。実行していたプログラムがユーザープログラムならばそのビットは 1 になる。もし

[4]MIPS コンパイラではフレームポインタは使わないので、このレジスタは呼び出された手続き側で値の保存をするレジスタ $s8 として使われている。

`interrupt enable` ビットが 1 ならば割り込みが許可される。もしこれが 0 ならば割り込みは禁止される。割り込みが発生する時にこれらの 6 ビットの値は 2 ビット左にずらされて現在のビットは前回のビットとなり、前回のビットは古いビットとなる。そして現在のビットはともに 0、つまりカーネルモードでかつ割り込み禁止となる。

図 4 は `Cause` レジスタの各ビットの説明である。`pending interrupt` での 5 ビットは、5 つの割り込みレベルそれぞれに対応している。ビットの値が 1 となるのは、そのビットに対応したレベルの割り込みが発生したにも関わらずまだ割り込み処理がされていない時である。`exception code` レジスタには例外の理由を表す値を保持する。例外とこのレジスタの値の関係は以下の通りである。

番号	名前	説明
0	INT	外部割り込み
4	ADDRL	アドレス誤り例外 (ロードまたは命令フェッチ)
5	ADDRS	アドレス誤り例外 (ストア)
6	IBUS	命令フェッチ時でのバスエラー
7	DBUS	ロード又はストア時でのバスエラー
8	SYSCALL	システムコール例外
9	BKPT	ブレイクポイント例外
10	RI	禁止命令例外
12	OVF	算術演算オーバーフロー例外

2.2 バイト・オーダー

プロセッサによってはワード内の最下位のバイトを一番右のバイトにするものもあれば一番左のバイトにするものもある。それぞれの機械によって慣例的に使われるバイトの並びを**バイトオーダー**と呼ぶ。MIPS プロセッサは以下に示す**ビッグ・エンディアン** (big-endian) バイトオーダー

バイト #			
0	1	2	3

あるいは以下に示す **リトル・エンディアン** (little-endian) バイトオーダーどちらとも使用できる。

バイト #			
3	2	1	0

SPIM はどちらのバイトオーダーも使用できる。SPIM のバイトオーダーは、シミュレーターを実行しているハードウエアのバイトオーダーで決定される。DECstation 3100 での SPIM はリトルエンディアンであるが、HP Bobcat, Sun 4, PC/RT での SPIM はビッグエンディアンである。

2.3 アドレッシング・モデル

MIPS はロード /ストア構造をしている。つまりロード命令とストア命令だけがメモリを参照する。計算命令はレジスタの値に対してのみ演算を行なう。素の機械では単一のアドレッシングモード `c(rx)` のみが用意されている。これはイミディエト (整数) `c` とアドレスとしてのレジスタ `rx` の値の和を表す。仮想機械ではロードとストア命令のために、以下のアドレッシングモードが用意されている。

書式	アドレス計算
(register)	レジスタの内容
imm	イミディエト (即値)
imm (register)	イミディエト + レジスタの内容
symbol	記号のアドレス
symbol ± imm	記号のアドレス + または − イミディエト
symbol ± imm (register)	記号のアドレス + または − (イミディエト + レジスタの内容)

ほとんどのロード命令とストア命令は整合されて配置されたデータに対してのみ動作する。データオブジェクトが**整合** (aligned) して配置されているとは、データオブジェクトが置かれているアドレスが (バイトで数えた) データの大きさの倍数になっているときをいう。例えば大きさが半ワードであるデータオブジェクトは偶数のアドレスに置かないといけない。また大きさがワードであるデータオブジェクトは 4 の倍数のアドレスに置かないといけない。しかし MIPS には、整合されていないデータを取り扱う命令がいくつか用意されている。

2.4 算術ならびに論理演算命令

以下に示すいずれの命令において `Src2` はレジスタまたは (16 ビット整数での) イミディエト値のどちらでもよい。イミディエト形式の命令の場合、参考の目的のみでここに含めている。アセンブラは、例えば `add` のようなより一般的な命令の形式を、二番目の引数が定数の場合は `addi` のような直接形式に変換する。

`abs Rdest, Rsrc` *Absolute Value* †
レジスタ `Rsrc` の整数の絶対値を求め、結果をレジスタ `Rdest` に入れる。

`add Rdest, Rsrc1, Src2` *Addition (*オーバーフローあり*)*
`addi Rdest, Rsrc1, Imm` *Addition Immediate (*オーバーフローあり*)*
`addu Rdest, Rsrc1, Src2` *Addition (*オーバーフローなし*)*
`addiu Rdest, Rsrc1, Imm` *Addition Immediate (*オーバーフローなし*)*
レジスタ `Rsrc1` の整数と `Src2` (または `Imm`) で表される整数との間の和を計算し、結果をレジスタ `Rdest` に入れる。

`and Rdest, Rsrc1, Src2` *AND*
`andi Rdest, Rsrc1, Imm` *AND Immediate*
レジスタ `Rsrc1` の整数と `Src2` (または `Imm`) で表される整数との間の論理積 (AND) を計算し、結果をレジスタ `Rdest` に入れる。

`div Rsrc1, Rsrc2` *Divide (*符合付き*)*
`divu Rsrc1, Rsrc2` *Divide (*符合なし*)*
2 つのレジスタの値の間で除算をする。`divu` はオペランド (operand, 被演算数) が符合なしの値として計算する。商はレジスタ `lo` に、剰余はレジスタ `hi` に入れられる。もしオペランドのどれかが負の場合、剰余の値がどうなるかは MIPS 計算機は規定していない。SPIM では使用している計算機に依存して決まる。

`div Rdest, Rsrc1, Src2` *Divide (*符合付き、オーバーフローあり*)* †
`divu Rdest, Rsrc1, Src2` *Divide (*符合なし、オーバーフローなし*)* †

レジスタ Rsrc1 の整数と Src2 で表される整数の商を計算し、レジスタ Rdest に入れる。 divu はオペランド全てが符合なしの値として計算する。

`mul Rdest, Rsrc1, Src2` *Multiply (*オーバーフローなし*)* †
`mulo Rdest, Rsrc1, Src2` *Multiply (*オーバーフローあり*)* †

`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply (*オーバーフローあり*)* †
レジスタ Rsrc1 の整数と Src2 で表される整数との間の積を計算し、結果をレジスタ Rdest に入れる。

`mult Rsrc1, Rsrc2` *Multiply*
`multu Rsrc1, Rsrc2` *Unsigned Multiply*
2 つのレジスタの値の積を計算する。 積の結果の下位のワードをレジスタ lo に入れ、 上位のワードを hi に入れる。

`neg Rdest, Rsrc` *Negate Value (*オーバーフローあり*)* †
`negu Rdest, Rsrc` *Negate Value (*オーバーフローなし*)* †
レジスタ Rsrc の整数を符合を反転し、結果をレジスタ Rdest に入れる。

`nor Rdest, Rsrc1, Src2` *NOR*
レジスタ Rsrc1 の整数と Src2 で表される整数との間の否定論理和 (NOR) を計算し、結果をレジスタ Rdest に入れる。

`not Rdest, Rsrc` *NOT* †
レジスタ Rsrc の整数値のビットごとの論理否定 (NOT) を計算し、結果をレジスタ Rdest に入れる。

`or Rdest, Rsrc1, Src2` *OR*
`ori Rdest, Rsrc1, Imm` *OR Immediate*
レジスタ Rsrc1 の整数と Src2 (または Imm) で表される整数との間の論理和を計算し、結果をレジスタ Rdest に入れる。

`rem Rdest, Rsrc1, Src2` *Remainder* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †
レジスタ Rsrc1 の整数値を Src2 で示される整数で割ったときの剰余を計算し、レジスタ Rdest に入れる。もしオペランドが負の場合、MIPS 計算機では剰余の値は不定であり、SPIM を実行している計算機に依存する。

`rol Rdest, Rsrc1, Src2` *Rotate Left* †
`ror Rdest, Rsrc1, Src2` *Rotate Right* †
レジスタ Rsrc1 の内容を Src2 で示される数だけ左 (あるいは右) にローテートし、その結果をレジスタ Rdest に入れる。

`sll Rdest, Rsrc1, Src2` *Shift Left Logical*
`sllv Rdest, Rsrc1, Rsrc2` *Shift Left Logical Variable*

`sra Rdest, Rsrc1, Src2` *Shift Right Arithmetic*
`srav Rdest, Rsrc1, Rsrc2` *Shift Right Arithmetic Variable*
`srl Rdest, Rsrc1, Src2` *Shift Right Logical*
`srlv Rdest, Rsrc1, Rsrc2` *Shift Right Logical Variable*
レジスタ `Rsrc1` の内容を `Src2` (`Rsrc2`) で示される数だけ左 (あるいは右) に論理シフトあるいは算術シフトをした結果をレジスタ `Rdest` に入れる。

`sub Rdest, Rsrc1, Src2` *Subtract (*オーバーフローあり*)*
`subu Rdest, Rsrc1, Src2` *Subtract (*オーバーフローなし*)*
レジスタ `Rsrc1` の整数から `Src2` で表される整数との間で減算をし結果をレジスタ `Rdest` に入れる。

`xor Rdest, Rsrc1, Src2` *XOR*
`xori Rdest, Rsrc1, Imm` *XOR Immediate*
レジスタ `Rsrc1` の整数値と `Src2` (または `Imm`) で表される整数との間の排他的論理和 (XOR) を計算し結果をレジスタ `Rdest` に入れる。

2.5 定数操作命令

`li Rdest, imm` *Load Immediate* †
レジスタ `Rdest` にイミディエト値 `imm` を入れる。

`lui Rdest, imm` *Load Upper Immediate*
イミディエト値 `imm` の下位の半ワードをレジスタ `Rdest` の上位の半ワードに入れる。 レジスタの下位ビットは 0 に設定される。

2.6 比較命令

以下に説明する命令では、`Src2` はレジスタでも (16 ビット整数の) イミディエト値のどちらでも良い。

`seq Rdest, Rsrc1, Src2` *Set Equal* †
レジスタ `Rsrc1` の値が `Src2` と等しければレジスタ `Rdest` の値を 1 に設定する。そうでなければ、0 に設定する。

`sge Rdest, Rsrc1, Src2` *Set Greater Than Equal* †
`sgeu Rdest, Rsrc1, Src2` *Set Greater Than Equal Unsigned* †
レジスタ `Rsrc1` の値が `Src2` 以上ならばレジスタ `Rdest` の値を 1 に設定する。そうでなければ、0 に設定する。

`sgt Rdest, Rsrc1, Src2` *Set Greater Than* †
`sgtu Rdest, Rsrc1, Src2` *Set Greater Than Unsigned* †

レジスタ Rsrc1 の値が Src2 より大きければレジスタ Rdest の値を 1 に設定する。そうでなければ 0 に設定する。

`sle Rdest, Rsrc1, Src2` *Set Less Than Equal* †
`sleu Rdest, Rsrc1, Src2` *Set Less Than Equal Unsigned* †
レジスタ Rsrc1 の値が Src2 以下ならばレジスタ Rdest の値を 1 に設定する。そうでなければ 0 に設定する。

`slt Rdest, Rsrc1, Src2` *Set Less Than*
`slti Rdest, Rsrc1, Imm` *Set Less Than Immediate*
`sltu Rdest, Rsrc1, Src2` *Set Less Than Unsigned*
`sltiu Rdest, Rsrc1, Imm` *Set Less Than Unsigned Immediate*
レジスタ Rsrc1 の値が Src2 より小さければレジスタ Rdest の値を 1 に設定する。そうでなければ 0 に設定する。
レジスタ Rsrc1 の値が Src2 と異なればレジスタ Rdest の値を 1 に設定する。そうでなければ 0 に設定する。

2.7 分岐ならびにジャンプ命令

以下に説明する命令では、Src2 はレジスタでも (16 ビット整数の) イミディエト値のどちらでも良い。分岐命令には符合付きの 16 ビットオフセット欄がある。つまり分岐命令では前方 $2^{15}-1$ (バイトではなく) 命令から、後方 2^{15} 命令までの間へのジャンプができる。**ジャンプ**命令には 26 ビットのアドレス欄がある。

`b label` *Branch instruction* †
ラベル label へ無条件に分岐する。

`bczt label` *Branch Coprocessor z True*
`bczf label` *Branch Coprocessor z False*
コプロセッサ z の条件フラグが真 (偽) の場合にラベル label へ分岐する。

`beq Rsrc1, Src2, label` *Branch on Equal*
レジスタ Rsrc1 の値と Src2 が等しいときラベル label へ分岐する。

`beqz Rsrc, label` *Branch on Equal Zero* †
レジスタ Rsrc の値が 0 と等しいときラベル label へ分岐する。

`bge Rsrc1, Src2, label` *Branch on Greater Than Equal* †
`bgeu Rsrc1, Src2, label` *Branch on GTE Unsigned* †
レジスタ Rsrc1 の値が Src2 以上ならばラベル label へ分岐する。

`bgez Rsrc, label` *Branch on Greater Than Equal Zero*
レジスタ Rsrc の値が 0 以上ならばラベル label へ分岐する。

`bgezal Rsrc, label` *Branch on Greater Than Equal Zero And Link*
レジスタ `Rsrc` の値が 0 以上ならば分岐命令の次の命令のアドレスをレジスタ 31 に保存して、ラベル label へ分岐する。

`bgt Rsrc1, Src2, label` *Branch on Greater Than* †
`bgtu Rsrc1, Src2, label` *Branch on Greater Than Unsigned* †
レジスタ `Rsrc1` の値が `Src2` より大きければラベル label へ分岐する。

`bgtz Rsrc, label` *Branch on Greater Than Zero*
レジスタ `Rsrc` の値が 0 より大きければラベル label へ分岐する。

`ble Rsrc1, Src2, label` *Branch on Less Than Equal* †
`bleu Rsrc1, Src2, label` *Branch on LTE Unsigned* †
レジスタ `Rsrc1` の値が `Src2` 以下ならばラベル label へ分岐する。

`blez Rsrc, label` *Branch on Less Than Equal Zero*
レジスタ `Rsrc` の値が 0 以下ならばラベル label へ分岐する。

`bgezal Rsrc, label` *Branch on Greater Than Equal Zero And Link*
`bltzal Rsrc, label` *Branch on Less Than And Link*
レジスタ `Rsrc1` の値が 0 以上あるいは 0 未満ならば分岐命令の次の命令のアドレスをレジスタ 31 に保存し、ラベル label へ分岐する。

`blt Rsrc1, Src2, label` *Branch on Less Than* †
`bltu Rsrc1, Src2, label` *Branch on Less Than Unsigned* †
レジスタ `Rsrc1` の値が `Src2` 未満ならばラベル label へ分岐する。

`bltz Rsrc, label` *Branch on Less Than Zero*
レジスタ `Rsrc` の値が 0 未満ならばラベル label へ分岐する。

`bne Rsrc1, Src2, label` *Branch on Not Equal*
レジスタ `Rsrc1` の値が `Src2` と等しくないときラベル label へ分岐する。

`bnez Rsrc, label` *Branch on Not Equal Zero* †
レジスタ `Rsrc` の値が 0 でないときラベル label へ分岐する。

`j label` *Jump*
ラベル label へ無条件にジャンプする。

`jal label` *Jump and Link*
`jalr Rsrc` *Jump and Link Register*
ラベル label へまたはレジスタ `Rsrc` の値のアドレスへ無条件にジャンプする。ジャンプ命令の次の命令のアドレスをレジスタ 31 に保存する。

`jr Rsrc` *Jump Register*
レジスタ `Rsrc` の値のアドレスへ無条件にジャンプする。

2.8 ロード命令

`la Rdest, address` *Load Address* †
レジスタ `Rdest` に *address* の計算結果をロードする。(注意: アドレスのメモリ内容ではない。)

`lb Rdest, address` *Load Byte*
`lbu Rdest, address` *Load Unsigned Byte*
レジスタ `Rdest` にアドレス *address* の 1 バイトの内容をロードする。`lb` 命令では読み込まれたバイトの符合拡張を行うが、`lbu` 命令では符合拡張はしない。

`ld Rdest, address` *Load Double-Word* †
レジスタ `Rdest` と `Rdest+1` にアドレス *address* の 64 ビットをロードする。

`lh Rdest, address` *Load Halfword*
`lhu Rdest, address` *Load Unsigned Halfword*
レジスタ `Rdest` にアドレス *address* の 16 ビット (半ワード) をロードする。`lh` 命令では読み込まれた半ワードの符合拡張を行うが、`lhu` 命令では符合拡張はしない。

`lw Rdest, address` *Load Word*
レジスタ `Rdest` にアドレス *address* の 32 ビット (ワード) をロードする。

`lwcz Rdest, address` *Load Word Coprocessor*
コプロセッサ z (ただし z は 0 から 3 まで) のレジスタ `Rdest` にアドレス *address* のワードをロードする。

`lwl Rdest, address` *Load Word Left*
`lwr Rdest, address` *Load Word Right*
レジスタ `Rdest` にアドレス *address* より始まる 1 ワードの中の左 (あるいは右) 側のバイトをロードする。(アドレス *address* は整合されていなくてもよい。)

`ulh Rdest, address` *Unaligned Load Halfword* †
`ulhu Rdest, address` *Unaligned Load Halfword Unsigned* †
レジスタ `Rdest` にアドレス *address* より始まる 1 ワードの中の左側 (あるいは右側) の半ワードをロードする。(アドレス *address* は整合されていなくてもよい。) `ulh` 命令は半ワードを符合拡張をするが `ulhu` 命令は符合拡張をしない。

`ulw Rdest, address` *Unaligned Load Word* †
レジスタ `Rdest` にアドレス *address* より始まる 32 ビット (1 ワード) をロードする。(アドレス *address* は整合されていなくてもよい。)

2.9 ストア命令

`sb Rsrc, address` *Store Byte*
レジスタ `Rsrc` の下位 1 バイトをアドレス *address* にストアする。

`sd Rsrc, address` *Store Double-Word* †
レジスタ `Rsrc` と `Rsrc + 1` の 64 ビットをアドレス *address* にストアする。

`sh Rsrc, address` *Store Halfword*
レジスタ `Rsrc` の下位半ワードをアドレス *address* にストアする。

`sw Rsrc, address` *Store Word*
レジスタ `Rsrc` の 1 ワードをアドレス *address* にストアする。

`swc`*z* `Rsrc, address` *Store Word Coprocessor*
コプロセッサ *z* のレジスタの 1 ワードをアドレス *address* にストアする。

`swl Rsrc, address` *Store Word Left*
`swr Rsrc, address` *Store Word Right*
レジスタ `Rsrc` の左 (あるいは右) のバイトをアドレス *address* にストアする。(アドレス *address* は整合されていなくてもよい。)

`ush Rsrc, address` *Unaligned Store Halfword* †
レジスタ `Rsrc` の下位半ワードをアドレス *address* にストアする。(アドレス *address* は整合されていなくてもよい。)

`usw Rsrc, address` *Unaligned Store Word* †
レジスタ `Rsrc` の 1 ワードをアドレス *address* にストアする。(アドレス *address* は整合されていなくてもよい。)

2.10 データ転送命令

`move Rdest, Rsrc` *Move* †
`Rsrc` の内容を `Rdest` に入れる。

乗除算ユニットは、演算結果を 2 つのレジスタ hi と lo に書き込む。転送命令はこれらのレジスタとの間で値を転送する。上で説明した乗算、除算、剰余命令はいずれも、乗除算ユニットが汎用レジスタに対して動作するかのように見せる疑似命令であり、0 による除算やオーバーフローなどのエラーの条件を検出する。

`mfhi Rdest` *Move From hi*
`mflo Rdest` *Move From lo*
レジスタ hi (または lo) の内容を `Rdest` に転送する。

`mthi Rdest` *Move To hi*
`mtlo Rdest` *Move To lo*
レジスタ `Rdest` の内容をレジスタ hi (または lo) に転送する。

コプロセッサには独自のレジスタ群がある。以下の命令はコプロセッサのレジスタと CPU のレジスタの間で値を転送する。

`mfcz Rdest, CPsrc` *Move From Coprocessor z*
コプロセッサ z のレジスタ `CPsrc` の内容を CPU のレジスタ `Rdest` に転送する。

`mfc1.d Rdest, FRsrc1` *Move Double From Coprocessor 1* †
浮動小数点レジスタ `FRsrc1` と `FRsrc1 + 1` の内容を CPU レジスタ `Rdest` と `Rdest + 1` に転送する。

`mtcz Rsrc, CPdest` *Move To Coprocessor z*
CPU レジスタ `Rsrc` の内容をコプロセッサ z のレジスタ `CPdest` に転送する。

2.11 浮動小数点命令

MPIS には単精度 (32 ビット) と倍精度 (64 ビット) の浮動小数点数の演算を行う浮動小数コプロセッサ (番号 1) がある。このコプロセッサは独自のレジスタ群を持ち、`$f0`–`$f31` と番号づけられている。これらのレジスタは 32 ビット幅なので、倍精度の値を保持するためには 2 つのレジスタが必要となる。単精度の演算命令であっても、単純化のために、浮動小数点演算は偶数番号のレジスタのみを使用する。

上で説明した `lwc1`, `swc1`, `mtc1`, `mfc1` 命令あるいは以下に説明する疑似命令の `l.s`, `l.d`, `s.s`, `s.d` により、浮動小数点レジスタは一度につき 1 ワード (32 ビット) の書き込みか読み出しが行われる。CPU は `bc1t` と `bc1f` 命令を使って浮動小数点数の比較によって設定されるフラグを読み出す。

以下に示す命令すべてにおいて `FRdest`, `FRsrc1`, `FRsrc2`, `FRsrc` は、浮動小数点レジスタ (例えば `$f2`) を表す。

`abs.d FRdest, FRsrc` *Floating Point Absolute Value Double*
`abs.s FRdest, FRsrc` *Floating Point Absolute Value Single*
レジスタ `FRsrc` に入っている倍精度 (あるいは単精度) 浮動小数点数の絶対値を計算し、その結果をレジスタ `FRdest` に入れる。

`add.d FRdest, FRsrc1, FRsrc2` *Floating Point Addition Double*
`add.s FRdest, FRsrc1, FRsrc2` *Floating Point Addition Single*
レジスタ `FRsrc1` と `FRsrc2` に入っている倍精度 (あるいは単精度) 浮動小数点数の和を計算し、その結果をレジスタ `FRdest` に入れる。

`c.eq.d FRsrc1, FRsrc2` *Compare Equal Double*
`c.eq.s FRsrc1, FRsrc2` *Compare Equal Single*
レジスタ `FRsrc1` と `FRsrc2` に入っている倍精度 (あるいは単精度) 浮動小数点数を比較し、それらが等しければ浮動小数点条件フラグの値を真に設定し、等しくなければ偽に設定する。

`c.le.d FRsrc1, FRsrc2` *Compare Less Than Equal Double*
`c.le.s FRsrc1, FRsrc2` *Compare Less Than Equal Single*
レジスタ `FRsrc1` と `FRsrc2` に入っている倍精度 (あるいは単精度) 浮動小数点数を比較し、`FRsrc1` が `FRsrc2` 以下であれば浮動小数点条件フラグの値を真に設定し、等しくなければ偽に設定する。

`c.lt.d FRsrc1, FRsrc2` *Compare Less Than Double*
`c.lt.s FRsrc1, FRsrc2` *Compare Less Than Single*
レジスタ `FRsrc1` と `FRsrc2` に入っている倍精度 (あるいは単精度) 浮動小数点数を比較し、`FRsrc1` が `FRsrc2` 未満であれば浮動小数点条件フラグの値を真に設定し、等しくなければ偽に設定する。

`cvt.d.s FRdest, FRsrc` *Convert Single to Double*
`cvt.d.w FRdest, FRsrc` *Convert Integer to Double*
レジスタ `FRsrc` に入っている単精度の浮動小数点数あるいは整数を倍精度の浮動小数点数に変換しその結果をレジスタ `FRdest` に入れる。

`cvt.s.d FRdest, FRsrc` *Convert Double to Single*
`cvt.s.w FRdest, FRsrc` *Convert Integer to Single*
レジスタ `FRsrc` に入っている倍精度の浮動小数点数あるいは整数を単精度の浮動小数点数に変換しその結果をレジスタ `FRdest` に入れる。

`cvt.w.d FRdest, FRsrc` *Convert Double to Integer*
`cvt.w.s FRdest, FRsrc` *Convert Single to Integer*
レジスタ `FRsrc` に入っている単精度の浮動小数点数あるいは倍精度の浮動小数点数を整数に変換しその結果をレジスタ `FRdest` に入れる。

`div.d FRdest, FRsrc1, FRsrc2` *Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2` *Floating Point Divide Single*
レジスタ `FRsrc1` と `FRsrc2` に入っている単精度あるいは倍精度の浮動小数点数の商を計算しその結果をレジスタ `FRdest` に入れる。

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †
アドレス `address` にある倍精度 (あるいは単精度) 浮動小数点数をレジスタ `FRdest` にロードする。

`mov.d FRdest, FRsrc` *Move Floating Point Double*
`mov.s FRdest, FRsrc` *Move Floating Point Single*
レジスタ `FRsrc` の倍精度 (あるいは単精度) 浮動小数点数をレジスタ `FRdest` に転送する。

`mul.d FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Single*
レジスタ `FRsrc1` と `FRsrc2` に入っている単精度 (あるいは倍精度) の浮動小数点数の積を計算しその結果をレジスタ `FRdest` に入れる。

`neg.d FRdest, FRsrc` *Negate Double*
`neg.s FRdest, FRsrc` *Negate Single*
レジスタ `FRsrc` に入っている単精度 (あるいは倍精度の) 浮動小数点数の符合を反転しその結果をレジスタ `FRdest` に入れる。

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †

レジスタ FRdest に入っている倍精度の (あるいは単精度) 浮動小数点数をアドレス address に格納する。

`sub.d FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Double*
`sub.s FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Single*
レジスタ FRsrc1 と FRsrc2 に入っている単精度 (あるいは倍精度) の浮動小数点数の差を計算しその結果をレジスタ FRdest に入れる。

2.12 例外ならびにトラップ命令

`rfe` *Return From Exception*
ステータスレジスタを復元し例外処理ハンドラの実行を終了する。

`syscall` *System Call*
システムコールを呼び出す。レジスタ $v0 に SPIM が提供するシステムコール番号 (表 1 参照) を入れる。

`break n` *Break*
例外番号 n の例外を発生させる。なお例外番号 1 はデバッガ専用である。

`nop` *No operation*
何もしない。

3 使用するメモリ

MIPS システムでのメモリのメモリの構成は従来手法に従っている。プログラムのアドレス空間は 3 つの部分より構成される。図 5 を参照せよ。

ユーザーアドレス空間の基底部 (0x400000) はテキストセグメント (text segment) で、プログラムを構成する一連の命令が配置される。

テキストセグメントの上にはデータセグメント (data segment) があり、これは 0x10000000 より始まる。データセグメントはさらに 2 つに分割されている。静的データ部にはコンパイラとリンカがあらかじめ大きさとアドレスを決定できるようなデータオブジェクトが配置される。これらのデータオブジェクトのすぐ上に動的データが配置される。プログラムの実行中に (例えば malloc 関数などにより) 動的にメモリ領域を割当てるごとに sbrk システムコールは高位のアドレスに向かってデータセグメントの上限を伸して行く。

アドレス空間の最高位部 (0x7fffffff) にはプログラムスタックが配置される。スタックはデータセグメントに向かって下方に広がってゆく。

4 手続き呼び出しの慣習

この章で説明する手続き呼び出しの手順は *gcc* で用いられている方法である。より複雑な代わりに僅かに高速な MIPS コンパイラで用いられている方法とこの方法は異なっている。

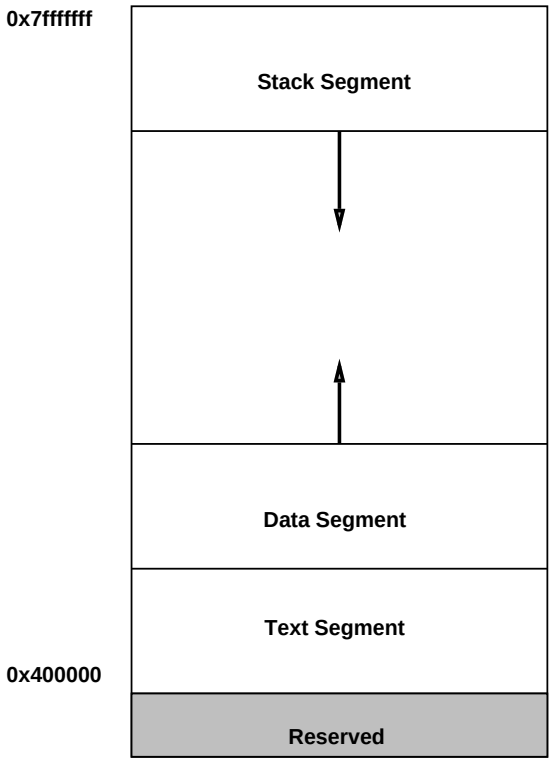

図 5: メモリ配置

図 6 はスタックフレーム (stack frame) の構造を図示している。フレームポインタ (frame pointer, `$fp`) はスタック上に積まれている最後の引数のワードのすぐ後を指し、スタックポインタ (stack pointer, `$sp`) はフレーム内の最後のワードを指している。これらの間のメモリ領域がフレーム (frame) を構成する。典型的な Unix システムでは高位のメモリアドレスから低位のメモリアドレスに向かってスタックは拡大して行く。つまりフレームポインタはスタックポインタより高位のアドレスにある。

手続きを呼び出すには以下の手順が必要である。

1. 引数を渡す。 最初の 4 つの引数はレジスタ `$a0`–`$a3` に入れて渡すのが慣例である。(この慣例に従わずに引数のすべてをスタックを使って渡すような単純なコンパイラもあるかも知れない。) 残りの引数はスタックにプッシュされる。
2. 呼び出し側保存のレジスタの値を保存する。呼び出し側においてレジスタ `$t0`–`$t9` の値が手続き呼び出し後にも必要ならばこれらを保存しないといけない。
3. `jal` 命令を実行する。

呼び出された手続きでは以下の手順が必要である。

1. スタックポインタからフレームサイズ分を引くことでスタックフレームを作る。
2. このフレームにおいて手続き側保存のレジスタの値を保存する。レジスタ `$fp` は常に保存を行う。さらに手続きを呼び出す場合はレジスタ `$ra` を保存する必要がある。レジスタ `$s0`–`$s7` のうち呼び出された手続内で使用するものは値を保存する必要がある。
3. レジスタ `$sp` の値にスタックフレームの大きさから 4 引いた値を加え、その結果をフレーム

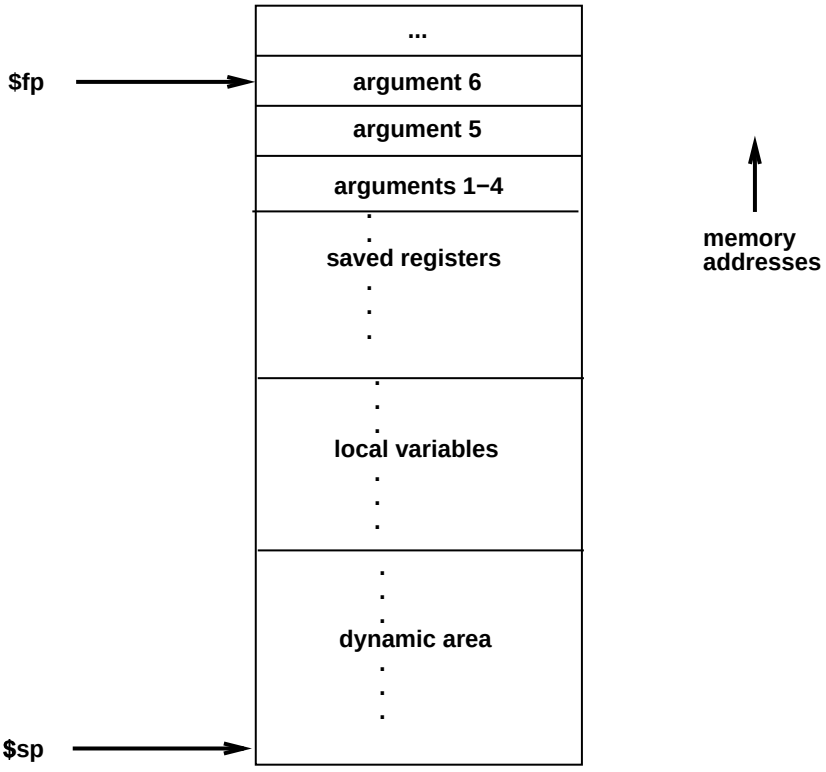

図 6: スタックフレームの配置。フレームポインタはスタック上に積まれている最後の引数のすぐ下を指している。スタックポインタはフレーム内の最後のワード を指している。

ポインタに設定する。

手続呼び出しから戻るときは戻り値をレジスタ $v0 に入れ、以下の手順を実行する。

1. 手続きの実行開始時に保存された手続き側保存のレジスタの値を復元する。(フレームポインタ $fp も含む。)
2. フレームの大きさを $sp に加えることでスタックフレームを破棄する。
3. レジスタ $ra の内容のアドレスにジャンプすることで呼び出し側に戻る。

5 入出力

CPU とオペレーティングシステムの基本的な動作を模倣するのに加えて SPIM は計算機に接続された端末機を模倣する機能を持つ。この端末機はメモリ割り付け型の端末である。プログラムの実行中は SPIM は独自の端末機 (xspim の場合は別のコンソールウインドウ) に接続する。プログラムは実行中にタイプされた文字を読み取ることができる。SPIM が端末機に文字を表示する命令を実行すると、文字が SPIM の端末機またはコンソールウインドウに表示される。例えばコントロール C が入力されたりブレイクポイントに到達するなどの理由でプロセッサが停止した場合、端末機は SPIM に再接続され、SPIM のコマンドが入力できるようになる。メモリ割り付け IO を使用するには、コマンドラインオプションに -mapped_io を加えて spim または xspim を起動しないといけない。

端末デバイスは受信部 (receiver) と送信部 (transmitter) の 2 つの独立した部分より構成されている。受信部はキーボードよりキーが押される度に文字を読み取る。送信部は端末の画面に文字を描く。2 つの部分は完全に独立している。このため例えば、キーボードで押された文字は自動的に画面に表示されない。押された文字を表示するにはプロセッサは入力された文字を受信部より受け取り、それを端末機に送り返さないといけない。

プロセッサは図 7に示された 4 つのレジスタを使って端末機にアクセスする。「メモリ割り付け」とは各レジスタが特定のメモリ位置に置かれている事を意味する。受信制御レジスタ (Receiver Control Register) は位置 0xffff0000 に置かれており、全ビットの内の 2 ビットだけが実際に使われる。ビット 0 は ready と呼ばれている。このビットが 1 の時はキーボードで入力され、まだ受信されていない文字が受信部のデータレジスタにあることを表す。ready ビットは読み出し専用であり、書き込みを行っても無視される。ready ビットはキーボードでキーが押されたときに自動的に 0 から 1 に変化し、受信部のデータレジスタよりデータが読まれたときに自動的に 1 から 0 に変化する。

受信制御レジスタのビット 1 は割り込みの許可を制御する。このビットはプロセッサによって読み出しも書き込みもできる。このビットの初期値は 0 である。もしプロセッサがこのビットの値が 1 に設定すると、ready ビットが 1 の時はいつも端末機によるレベル 0 の割り込み要求が発生する。プロセッサが実際に割り込みを受け付けるためには、コプロセッサのステータスレジスタで割り込みを許可しないといけない。(第 2 章参照のこと。)

受信制御レジスタの他のビットは使用されていない。読み出しの時は 0 が返され、書き込みをしても無視される。

二番目の端末装置のレジスタはアドレス 0xffff0004 に置かれている受信データレジスタ (Receiver Data Register) である。このレジスタの下位 8 ビットは最後にキーボードで押された文字を保持し、他のビットは全て 0 である。このレジスタは読み出し専用でありキーボードでキーが押された

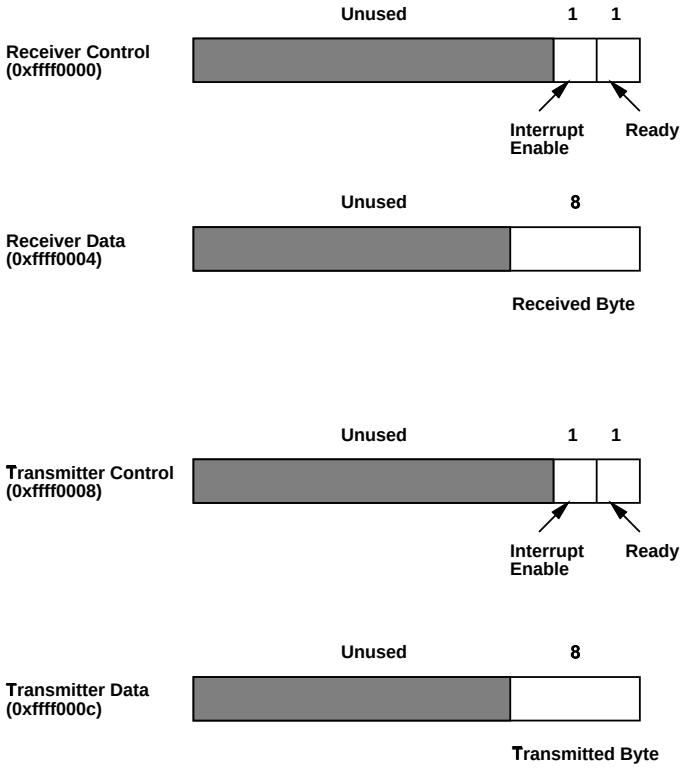

図 7: 端末機は 4 つのレジスタによって制御される。それぞれはある決まったアドレスのメモリ位置に配置されている。これらのレジスタの内、一部のビットのみが使用される。その他のビットは読み出しの時は 0 が返され、書き込みをしても無視される。

ときのみに値が変更される。受信データレジスタを読み出すと、受信制御レジスタの ready ビットは 0 に変化する。

三番目の端末装置レジスタは送信制御レジスタ (Transmitter Control Register) で、アドレス 0xffff0008 に置かれている。このレジスタの下位 2 ビットだけが使用され、それらは受信制御レジスタでの対応したビットと同様な働きをする。ビット 0 は ready ビットで、読み出し専用である。もしこのビットが 1 の場合、送信部が別の文字の送信の準備ができていることを表す。もしこのビットが 0 の場合は、前回に与えられた文字が現在送信中であることを表す。ビット 1 は、割り込み許可ビットである。このビットは読み出しも書き込みも可能である。このビットが 1 の時、ready ビットが 1 の時はいつでも、レベル 1 の割り込み要求を発生する。

最後の端末装置レジスタは、転送データレジスタ (Transmitter Data Register) で、アドレス 0xffff000c に置かれている。このレジスタにデータが書かれると下位 8 ビットが ASCII 文字コードとして取出されて画面に表示される。送信データレジスタに書き込みがあると、転送制御レジスタの ready ビットは 0 に設定される。 文字が端末機に送られるまでの間、このビットは 0 のままである。そして ready ビットは再び 1 になる。転送データレジスタへの書き込みは転送制御レジスタの ready ビットが 1 の時のみに行うべきである。転送の準備ができていないときは転送データレジスタへの書き込みは無視される。つまり書き込みはできても文字は表示されないことになる。

実際の計算機では、端末機と計算機を結ぶシリアル線を通じて文字を送るのにある程度時間がかかる。 SPIM は転送に要する時間も模倣している。例えば転送部が文字を転送開始した後に、転送部の ready ビットはしばらくの間 0 になる。 SPIM では実際の時間ではなく、実行する命令数でこの時間を計測している。つまりプロセッサがある一定数の命令を実行しない限り、転送部は次の転送の準備ができないということである。もし機械を止めて SPIM を使って ready ビットの値を観測すると、いつまでも変化しない。しかし機械を実行させるといつかは必ず 1 に戻る。