東海大学の清水です。

99年3月号の原稿になります。よろしくお願いします。
また、先日御提案のXEONとの性能比較ですが、ぜひ取り上げたいと思います。
ですがせっかくなら比較対象を新しい21264のマシンにしたいと思います。
コンパックでちょっと知っている人に聞いたところ代理店経由で協力できると
思うとお答えを頂いておりますので、XEONと264のマシンの貸し出しを手配し
てもらえないでしょうか?

〆切間際になってLinux/Alpha用のApplixwareが出るらしいという
情報が流れてきました。いよいよLinux/Alphaも市場に認知されてきたという
証拠でしょうか? 1月号ではAvalonの話を書いたのですが、その後Avalonはプ
ロセッサを倍に増やしています。例年スーパーコンピュータの国際会議があ
る11月に世界のコンピュータセンターの処理能力を上から500サイト紹介す
るTOP500という発表があってお祭り騒ぎをやるのですが、今年はその中に
Avalonを含めてLinux/Alphaのクラスタが二つ入りました。一つはTOP100に
入っているし、Avalonも大健闘です。商用の並列スーパーコンピュー
タもリストに出ているものの多くはSGI-CrayのT3Eというマシンです。実はこ
れにはアルファ21164Aが使われていますので、実質世界のスーパーコンピュー
タはアルファチップで寡占状態にあるといっても過言ではないかもしれません
。Linux/Alphaに対する世間の関心が高まる中、
Linux/Alphaの性能に関する国際的なメイリングリストもできました。

http://www.alphalinux.org/archives/

にメイリングリストでの議論が出ています。

1. Linuxにおけるスタック使用量の制限

さて、私には忙しくなってくると全く別のことをやりたくなるくせがあって
さらに余計なこと抱え込んでしまうことがよくあります。

先日も
プロセッサの設計を行う演習の為に見本となるプロセッサを作っていたのです
が、ついこの癖を発揮してよせばいいのにデータキャッシュを組み込んだ
プロセッサを作ってヒットアンダーミスやストアバッファ、命令の
フルインターロックなどを入れて8ビットで10命令くらいしかない簡易
アーキテクチャながら本格的なパイプラインプロセッサに仕立てました。
基本のパイプラインプロセッサも分岐予測を組み込んだフルインターロックの
構成でインターバルタイマを入れて割り込みの練習ができるようにしています
。ちょっと構成を欲張ったので演習で使うALTERA社のCPLD EPF10K10に組み込む
ためにゲートを減らすよう試行錯誤を繰り返しているうちにずいぶんと時間が
たつものです。(おおーっ、〆切がー!)

こんな話がどこでLinuxにつながるんだと思っている方もいるかもしれません
が、実はこのプロセッサの設計に使っているPARTHENONという論理CADはLinux
で動作するのです。私のところでは、Linuxで設計とシミュレーションでの動作
確認と論理合成をした後でCPLDへのフィッティングを行うという形で演習を進めてい
ます。
プロセッサ(CPU)の構造をくわしく知りたい読者はソースコードをダウンロー
ドしてじっくり読んでみることをお勧めします。はじめての人は小さい方のプ
ロセッサSP/1が分かりやすいと思います。
このコードは著作権表示さえ
していただければ商用を含み自由に使っていただいてかまいません。
設計に当たっては遅延を減らすためにいろいろな工夫をしているので
ALTERAのCPLDの内部RAMを命令メモリやデータメモリに使えばかなりのスピー
ドで動作すると思います。また、分岐予測も簡単ながら入っているので
単純なループなら1クロック/命令が実現できています。
実は規模の大きなSP/1Cの方が特定のCPLDへの実装を考えていない分クロック
周波数は高くなっています。
これらのソースコードやシミュレーションのスクリプトは次のURLから入手で
きます。

http:/shimizu-lab.et.u-tokai.ac.jp/pgm/sp1

http:/shimizu-lab.et.u-tokai.ac.jp/pgm/sp1c

ついでに、Java Virtual Machine 命令互換のプロセッサ TRAJA 2.0 を GPL に
従い公開しています。こちらも合わせて御覧ください。

        http:/shimizu-lab.et.u-tokai.ac.jp/pgm/traja2

さて、
SP/1 では問題無いのですが、SP/1C や TRAJA を Linux で論理合成しようとすると
合成プログラム
が SEGV シグナルで止まることがあります。これはメモリ不足の場合に発生する
シグナルで普通はスワップの容量不足を真っ先に疑う性質のものです。
しかし、論理合成では十分なメモリがあっても簡単に発生します。
実は Linux ではメモリの使用量の制限がかかっているのです。FreeBSD などでも
同じように制限がありますが、csh や bash から簡単に外せます。
ところが、Linux ではスタックの容量をルート以外のユーザーが 8MB 以上に拡張でき
ないように設定しています。comp.os.linux.development.system でなんでそう
なっているのか聞いても分かっていないような答えしか帰ってきませんし、カ
ーネルの特定部分の設計はメインテナーが見つからないので Linus 氏にメイル
を出したのですが答えが帰ってきません。(彼はメイルの対応で忙しすぎるの
でしょうね) ということで、勝手にパッチを作って使えるようにしました。
Intel アーキテクチャ用のこのパッチが必要な方は

        http://shimizu-lab.et.u-tokai.ac.jp/pgm/sp1/linux.patch

を/usr/src/linux/include/asm/resource.h に当ててください。
パッチを当てたカーネルはつぎのようにユーザーが自分でスタックサイズを
決められます。
スタックの制限を無くしたい方で
bash を使用している方は

ulimit -s unlimited

csh の方は

limit stacksize unlimited

としてください。

さて、スタックの最大容量をユーザーが自分で決められると何が
まずいのでしょうか? 過去のニュー
スを検索しても私が問い合わせても再帰定義に失敗したプログラムがメモリを
使い尽くしてシステムがダウンするという答えが帰ってきています。しかし、
リアルモードの MINIX ならいざしらず
多重仮想記憶を使っている Linux にとってこの答えはナンセンスですね。
8MB という容量の説明には全くなっていません。敢えて制限をつけるならスタ
ックが拡張してシェアードライブラリのエリアにぶつかるまでの大きさとすべ
きでしょうが、今のユーザーは数 GB のメモリを載せているケースはあまりない
のでこれも心配するほどの問題ではないでしょう。
スタック容量の上限がどうであろうと実際にページが要求されるまではページ
の割当が行われないのが普通のデマンドページのシステムですから、スタック
がどんどん増えて割当可能なページがなくなると要求したプロセスは SEGV のシ
グナルを受けてお亡くなりになります。これはスタックであってもヒープであ
っても全く同じですからデータサイズがデフォルトで unlimited なのにスタッ
クはユーザーからは拡張できないようにしておくという Linux の手法は意味が
ありません。デフォルトのスタックサイズを小さく設定しておくのはプログラ
ムの再帰定義の失敗が早めに分かる程度の効果はありますから、上記のパッチ
ではデフォルトのスタックサイズは触っていません。

大規模な数値計算を行う Linux/Alpha のユーザーの中にもこの問題に悩んでいる方
も多いのですが、対策は簡単です。
対策方法の一つは局所変数に大きな配列を宣言しないという方法があります。
大きなメモリ領域が必要な時には malloc で明示的にメモリ確保をしてあげれば
大きなスタックは不要です。
ですが、すでにプログラムがある場合にどこでエラーが発生するのか調べるの
は案外難しいものです。そこで、スタックサイズの最大値を更新するため
つぎのパッチを

/usr/src/linux/include/asm-alpha/resource.h

に当てカーネルを作り直してください。

```
*** resource.h.orig     Thu Dec  3 11:15:46 1998
--- resource.h  Thu Dec  3 11:16:00 1998
***************
*** 25,31 ****
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_CPU */ \
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_FSIZE */ \
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_DATA */ \
!      {_STK_LIM, _STK_LIM},                    /* RLIMIT_STACK */ \
       {       0, LONG_MAX},                    /* RLIMIT_CORE */ \
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_RSS */ \
       { NR_OPEN,  NR_OPEN},                    /* RLIMIT_NOFILE */ \
--- 25,31 ----
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_CPU */ \
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_FSIZE */ \
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_DATA */ \
!      {_STK_LIM, LONG_MAX},                    /* RLIMIT_STACK */ \
       {       0, LONG_MAX},                    /* RLIMIT_CORE */ \
       {LONG_MAX, LONG_MAX},                    /* RLIMIT_RSS */ \
       { NR_OPEN,  NR_OPEN},                    /* RLIMIT_NOFILE */ \
```

この構造はユーザプロセスのデフォルトと最大の各種制限値を決定しています。変更点は最大のスタックサイズを拡張したことです。管理者によっては最大データサイズやCPU時間などいろいろと制限しておきたい人もいるかもしれませんが、この構造をいじるといろいろと設定できます。

2. 新プロセッサ21264/21364の概要

アルファの新しい世代のプロセッサ21264が出荷されいろいろなところでベンチマークの結果が出つつありますが、その興奮も醒めやまないうちにマイクロプロセッサフォーラムで新しいプロセッサ21364がアナウンスされました。出荷はまだまだ先ですが、なかなか面白いプロセッサに仕上がりそうです。
21264はハードウェアリファレンスマニュアルが出版されていないので
正確なところが不明ですが今分かっている概要をお伝えします。

2.1 21264

21264の構成等いろいろな情報は各種雑誌に紹介されているし、WEBからも入手できるのでここでは新しい機能を整理して数値計算での有用性について考えて見たいと思います。

・一次キャッシュが128KBになった(データ/命令それぞれ64KB)。
　アクセスにかかる時間は伸びたのですが、アウトオブオーダ実行によって遅れた分の時間は他の命令が動作することによって容量の増加に見合った性能が得られます。キャッシュミスの頻度を低減するこの拡張は一般のアプリケーションには大きな効果があります。数値計算の時にもブロッキング単位をこの範囲に収めれば高い性能が見込めます。ただし、21164の二次キャッシュより容量が小さいので注意が必要です。

・外部キャッシュのデータ/アドレスがメモリバスと分離された。
　従来はメモリのデータを外部キャッシュに格納している間は新たなメモリデータの転送ができなかったので、バススピードは実質半分しか使えなかったのですが(ストアの時にはもっと悪くなります)、キャッシュバスとメモリバスを分離したことによって十分な性能が得られるようになりました。数値計算ユーザーは素直によろこびましょう。:-)

・バスインターフェイスを改良しメモリ転送スループットが向上した。
　これも上の事項と同様転送スループットの向上は多くの場合直接数値計算の性能向上につながります。また、メモリバス上のプロトコルとしては最大16個のメモリ要求が発行可能なように構成されています。

・キャッシュミス時の外部へのメモリ要求を8個まで発行可能にした。
　21164では二つまでだったのですが、一気に4倍にしました。もっともその大半は二次キャッシュに吸収されるので21164に比べてどのくらい上がったことになるのかはハードウェアマニュアルの登場まではよく分かりません。もしか

したら従来の6個のミスアドレスファイル(MAF)が8個になっただけという可能性もあります。

・アドレス変換バッファが128エントリに増やされた。
　これはまだまだ全然足りないです。一気に1024エントリくらいに増やして欲しいところです。

・アウトオブオーダ実行可能となった。
　上記のような改良点をフルに引き出すプロセッサの構成上の要です。命令の依存性や命令のメモリ上の配置に関係なく次々に命令の発行ができ、発行された命令はプロセッサのキューにおいて依存関係の解消やデータの待ち合わせをすることになります。プログラムの並びが性能に与える影響が低くなるので特にg77やgccなどで利用しているユーザーには朗報です。メモリの読みだし遅延を考えたプログラムを作りさえすれば命令の並びを気にしなくても十分高性能なプログラムが作成できるようになります。

・平方根の命令が追加された。
　平方根を多用するユーザーには朗報ですが、実はあまり早い演算にはなっていません。

・外部チップセットはプロセッサと一対一の接続となった。
　これはプロセッサ本体のはなしではありませんが、外部のチップセットTSUNAMIはクロスバーをサポートするようになりました。構成によってメモリを128ビットから512ビットまでのバンクの構成で使うことができるので、広い性能レンジの製品が提供されることと思います。

2.2 21364

マイクロプロセッサフォーラムにおいて21364が発表されました。
このプロセッサは21264に各種の機能を追加したようなプロセッサになっています。
追加された項目は次の通りです。

・1.5MBもの大容量の二次キャッシュがオンチップに乗ります。
　12nSのレイテンシとなっているので今世の中にある外付け用のキャッシュに比べてあまり早くはないのですが、バンド幅はさすがに16GB/Sと優れています。また、外付けのチップを使わずにシステムを構成できるので量産でプロセッサが安くなるとシステム価格が非常に安くなる可能性があります。

・ダイレクトRAMBUSのメモリコントローラがチップに統合されます。
　とうとうきたかという感じですね。直接RAMBUSに要求を発行することでメモリのレイテンシを低く押さえることができます。メモリ転送性能は6GB/Sとされています。

・プロセッサ間通信のためのネットワークインターフェイスが追加されます。
　このインターフェイスは10GB/Sのスループットでデータ転送ができるとなっています。プロセッサ当たり4本のプロセッサポートと1本のI/Oポートがあります。ディレクトリベースのコヒーレンシ制御機構を持っているし、プロセッサ間で非同期転送を行うとなっているので、SCIのプロトコルに近いインターフェイスの可能性があります。

・可用性向上の為のロックステップ機能がつけられます。
　(何でしょうね?)

・一次キャッシュのミスバッファが16エントリになります。
　21264ではミスキューが8個だったのですが、コアを変えずに外部に16個つけてもしかたがないのでおそらくミスキューが倍になるのだと思います。

・一次キャッシュと二次キャッシュのビクティムバッファがそれぞれ16エントリになります。
　ビクティムって16エントリも必要なのかはよく分かりませんが、キャッシュの競合による性能低下を押さえる働きをします。

このプロセッサの構成を見ているとSGI-CrayのT3Eの後継機種にぴったりはまりそうな気がひしひしとしてきます。SGIではT3Eの後継機種の選定作業に入っているでしょうから今21364を発表したというのはそれを狙っているのでしょうか? もっとも、SGIが作らなくても21364なら誰だってT3E相当のマシンを作

ることができるので、残る問題はコンパイラやアプリケーションだけです。
これらのソフトウェアはハードウェアが十分安く提供されればそれを使いたい
人達のあいだで(Linux のように)改良されリリースされる可能性があります。

駆け足で新プロセッサの概要を紹介しました。
いくらアウトオブオーダといってもメモリの読みだし速度はそんなもので吸
収できるほど小さくありませんし、分岐を正しく予測できなければほとんど捨
てるための無駄な命令ばかりを実行していることにもなりかねません。
例えば、ロード命令の間に 80 命令の命令を先に実行できる考えてください。
分岐の頻度が 20%位として 8 個の分岐をまたいで命令が発行されることになるの
は分かりますね。では、各分岐の予測成功率を 21264 のように 95%と考えてみま
しょう。すると、80 命令後には 16 命令の分岐を実行してきたはずですね。(あ
くまでも確率の問題です。)すると 80 命令後に実行するべき命令を実行してい
る確率は 19%になります。これが、予測成功率が 90%にまで低下すると正しい命
令を実行している確率は限りなく 0 に近付いてしまいます。このように分岐予
測はアウトオブオーダの実行を行う時には大変重要な問題になります。

3. Alpha のアセンブリ言語入門(その三)

さて、前回のアセンブリ言語入門はいかがでしたか? ちょっとここまではでき
ないけど C や FORTRAN ではできないことを少しだけしてみたいという方も中には
いるかと思います。今回はそんなあなたのインラインアセンブラの使い方を簡
単に説明します。
インラインアセンブラは C のコード中に直接アセンブラを表記できるものです。
次のように表記します。

```
 asm volatile ("addt %1,%2,%0" :"r="(result): "r"(source1),"r"(source2))
```

ここで、最初のオペランドは命令列の文字列を表しています。
命令は一つでも複数でもかまいません。複数命令の命令列を指定したい時には
命令間を \n\t で区切っておくと分かりやすいと思います。
：で区切られたところにそれぞれ結果の指示と呼び出しオペランドを書きます。
プログラムの中で引数を指定する時には出現順に %0, %1, %2 の用に指定しま
す。
"r="となっているのは結果をこの変数(レジスタを介して)result に
書き込むということ表しています。命令は二つのレジスタを使ってそれぞれ
source1,source2 という変数から渡されます。メモリの内容を使いたい時には
"r"でなく"m"とします。
例題を以下示しますが、もっといろいろなケースを知りたいという方は

/usr/src/linux/arch/alpha

のディレクトリもしくは

/usr/src/linux/include/asm-alpha

において

```
grep asm *{,/*.c} | grep volatile
```

として探してみてください。

説明は例題から入った方が分かりやすいのでまずは次のプログラムを御覧ください。

```
#define N 100000

double b[N+8][3];

double foo(double b[][3]) {
int i;
double t0,t1,t2,t3,t4,t5;
double a1,a2;

        i = 0;
        a1 = a2  = 0.0;
        t0 = b[i][0];
        t1 = b[i][1];
```

```
        t2 = b[i][2];
        t3 = b[i+1][0];
        t4 = b[i+1][1];
        t5 = b[i+1][2];

        for(i=0; i<N-2; i+=2)
         {
           a1 += 1/(1+(t0*t0) + (t1*t1) + (t2*t2));
           a2 += 1/(1+(t3*t3) + (t4*t4) + (t5*t5));
           t0 = b[i+2][0];
           t1 = b[i+2][1];
           t2 = b[i+2][2];
           t3 = b[i+3][0];
           t4 = b[i+3][1];
           t5 = b[i+3][2];
         }
        a1 += a2;
        for(i=i; i<N; i++)
           a1 += 1/(1+b[i][0]*b[i][0]+b[i][1]*b[i][1]+b[i][2]*b[i][2]);
        return(a1);
}
```

このプログラムは特に何かの意味があるわけではありませんが、
比較的単純な演算のループをアンローリングとソフトウェアパイプラインによって
すこしだけ高性能化したものです。
ところが、このループはせっかく苦労してコーディングしてもあまり早くなり
ません。その理由を次に説明します。
一般に大きな配列に対するロード命令はキ
ャッシュブロックの境界でほぼ確実にキャッシュミスを起こしてしまいます。
代表的なメモリのアクセス時間は60nS程度ですが、実際に外部のチップセット
を通ってプロセッサに到着するまでにはもっとずっと長い時間(例えば160nS)
がかかります。500MHzのクロックのマシンでは1クロックは2nSになります。
すると80クロックの間データの待ち合わせが発生することになります。
ソフトウェアパイプラインを使っているとはいってもデータが到着しないこと
には除算の計算ははじめられません。しかも時間のかかる除算を待つ間に
待ち合わせができるならよいのですが、除算で使うデータそのものを待ってい
る以上データがこない限り除算の演算もできないわけです。
そこで、長い時間かけてデータを待った挙げ句除算で演算の時間を待ち合わせ
る必要がそのままかかってしまいループの性能を低下させています。

これに対処するために21164以降のアルファチップにはプリフェッチの命令が
用意されています。といっても特別なことはなく常に0である
レジスタにロード命令を発行するときちんとメモリまでデータを取りに行くと
いうだけです。
21164以前のプロセッサではキャッシュミスするとその場でプロセッサが止ま
ってしまうのであらかじめ読み出しを発行しても早めに止まるだけで効果はあ
りません。
21164ではキャッシュにヒットしなかった場合でもミスアドレスファイルに登録して
そのまま後続の命令を実行するし、二つまでのメモリ読み出しが
外部に発行されるのでこの仕組みをうまく使うとSDRAMをかなりの利用率で駆
動することができます。
面白いのは浮動小数点のレジスタを読み出す場合と整数レジス
タを読み出す場合で(プリフェッチだけでなく)キャッシュへのデータの登録の
方法が少し異なることです。 その結果, 浮動小数点のレジスタを対象にプリフ
ェッチを行うと一次キャッシュが使われていて忙しい場合には二次キャッシュ
まで持ってきたところで動作が止まりますので、プリフェッチによる余計な性
能低下の可能性が低くなります。
この命令は結果を生じないのでいつ発行してもかまわないわけで、
これによりあらかじめ必要なデータをキャッシュまで取って来ることが
できるのです。そこで、プリフェッチによって先に使うべきデ
ータをあらかじめ読みだしておけば実際に使う時にはデータがすでに到着して
いるという具合です。

ところが、CやFORTRANにはプリフェッチなどという概念は無いですからこれを
使いたければアセンブラで最適化したプログラムを作らなくてはならないと思
っている人が多いわけです。
そんなときにgccに備わっているインラインアセンブラの機能を使うとスマー
トに解決できます。

プリフェッチ付のプログラムは次のようになります。

```
#if defined(__alpha__)
#define prefetch(x) asm volatile ("ldt $f31,%0" :: "m"((x)))
#else
#define prefetch(x) asm volatile ("nop" :: "m"((x)))
#endif
#define N 100000

double b[N+8][3];

double foo(double b[][3]) {
int i;
double t0,t1,t2,t3,t4,t5;
double a1,a2;

        i = 0;
        a1 = a2  = 0.0;
        t0 = b[i][0];
        t1 = b[i][1];
        t2 = b[i][2];
        t3 = b[i+1][0];
        t4 = b[i+1][1];
        t5 = b[i+1][2];

        for(i=0; i<N-2; i+=2)
         {
            prefetch(b[i+3][2]);
           a1 += 1/(1+(t0*t0) + (t1*t1) + (t2*t2));
           a2 += 1/(1+(t3*t3) + (t4*t4) + (t5*t5));
           t0 = b[i+2][0];
           t1 = b[i+2][1];
           t2 = b[i+2][2];
           t3 = b[i+3][0];
           t4 = b[i+3][1];
           t5 = b[i+3][2];
         }
        a1 += a2;
        for(i=i; i<N; i++)
           a1 += 1/(1+b[i][0]*b[i][0]+b[i][1]*b[i][1]+b[i][2]*b[i][2]);
        return(a1);
}
```

prefetchでは2個先のループのデータを要求しています。
その後、除算の演算をしている間にこの要求はメモリまで到着してデータをキャッシュ(二次キャッシュ)に持ってきます。その段階では1次キャッシュにはデータは到着していませんが、ソフトウェアパイプラインにより次の除算までにはレジスタまで読み込みがなされます。
21164の二次キャッシュはパイプライン動作が可能になっていてソフトウェアパイプラインで次々とデータが要求されてもプロセッサを止めること無く二次キャッシュから一次キャッシュにデータの転送をすることが可能です。

さて、このように細かくインラインアセンブラでプリフェッチを出したりするチューニングをする場合にはその効果を常に把握しておきたいものです。
その場合にはプロセッサのクロックサイクルをカウントするrpccという命令を使うのが便利です。PCCというレジスタは全てのアルファのアーキテクチャに定義されているので全てのアルファで利用可能です。ただし、このレジスタの1カウントは1から16まで実装依存で変わる可能性があります。今出荷されている21164までは1カウントが1クロックになっていますが、将来高速なプロセッサが出てきた時には変わる可能性があることに注意してください。このレジスタは32ビット(!)なので少し長い時間をはかりたい場合には使えません。
また、ちょうどタイミング的にカウンタが一周するところに引っかかると単に差を取っただけではうまくないので注意が必要です。
サイクルカウンタを読み出すためのインライン関数は次のように定義します。
(インラインにしないとせっかく計測するのに関数呼び出しの余計な時間が必要になるので必ずインライン関数にしてください。)

```
static inline int rpcc(void)
{
    int result;

    asm volatile ("rpcc %0" : "r="(result));
    return result;
}
```

先程提示した関数 foo とその変形を使って
次のようにプリフェッチの効果を調べてみましょう。
prefetchi は整数レジスタを使ったプリフェッチです。これはデータを一次キャッシュまで運びます。nop はその名の通り何もしない命令です。

```
#define prefetch(x) asm volatile ("ldt $f31,%0" :: "m"((x)))
#define prefetchi(x) asm volatile ("ldq $31,%0" :: "m"((x)))
#define nop(x) asm volatile ("nop" :: "m"((x)))
#define N 500000

static inline int rpcc(void)
{
    int result;

    asm volatile ("rpcc %0" : "r="(result));
    return result;
}

double foo(double b[][3]) {
int i;
double a;
        a=0;
       for(i=0; i<N; i++) {
           prefetch(b[i+1][2];
          a += 1/(1+b[i][0]*b[i][0]+b[i][1]*b[i][1]+b[i][2]*b[i][2]);
           }
       return(a);
}

double foo1(double b[][3]) {
int i;
double a;
        a=0;
       for(i=0; i<N; i++) {
           prefetchi(b[i+1][2];
          a += 1/(1+b[i][0]*b[i][0]+b[i][1]*b[i][1]+b[i][2]*b[i][2]);
           }
       return(a);
}

double foo1(double b[][3]) {
int i;
double a;
        a=0;
       for(i=0; i<N; i++) {
           nop(b[i+1][2];
          a += 1/(1+b[i][0]*b[i][0]+b[i][1]*b[i][1]+b[i][2]*b[i][2]);
           }
       return(a);
}

double vect[N][3];
main() {
int i,j,j1;
double a,b;

 b=0;
 printf("start\n");
```

```
 j = rpcc();
 a += foo(vect);
 j1 = rpcc();
 b = (double)(j1-j);
 printf("foo with prefetch  need %d cycles\n",(int)b/N);
 j = rpcc();
 a += foo1(vect);
 j1 = rpcc();
 b = (double)(j1-j);
 printf("foo with prefetchi need %d cycles\n",(int)b/N);
 j = rpcc();
 a += foo2(vect);
 j1 = rpcc();
 b = (double)(j1-j);
 printf("foo with nop       need %d cycles\n",(int)b/N);
}
```

私のところには21164以降のマシンはないので結果は読者におまかせします。m_o_m
この手の性能の問題を扱う時に注意しなくてはならないのはキャッシュのミス
の効果とDTB(アドレス変換バッファ)のミスの効果が両方入ってくることです
。これを注意深く区別してあげないと得られたデータはつじつまが合わないよ
うに見えたりします。

3. おわりに

新しいプロセッサは原稿執筆時点の予価として$20,000程度の値段を提示して
いる店があります。是非とも評価したいプロセッサの一つですが、まだまだ個
人で買える値段になるには時間がかかりそうです。

余談ですが、忙しくなると全く別のことをやりたくなるという癖をいかんなく
発揮して(?)最近Fenderの JAZZ BASSを買いました。
購入にあたっていろいろな会社のカタログを見比べて検討したのですが、
(安いパッシブなフレットレスが欲しいという要求仕様がそもそも選択肢を
狭めているのですが)
どうもどの会社も個性的なものを出していません。Fenderならオ
リジナルかと思うと'60年代の製品のコピーだったり有名演奏者の楽器のコピ
ーだったりちっともオリジナリティを感じません。
その反動からか雑誌の中古の広告にはTEISCOやGUYATONEなどの大昔のギターが
並んでいたりします。(ちなみに私はTEISCOのWG-4Lなんてものも持っ
ています。)

コンピュータでもギターでも個性が強烈に出ている方が使っていて面白いので
、新しいプロセッサは良い設計ではあるけれど21164の強烈な個性を見慣れた
(ただし持っていない)身には新鮮さに欠けるように思えてしまいます。