2001年9月の終わりに東京信濃町の明治記念館で行われた、「Linux Conference 2001」というイベントに参加しました。私が参加したのは最終日のカーネルハックの発表と懇親会だけですが、Linux関連のイベント参加は初めてのことで楽しく過ごしました。このイベントでの私の発表は多粒度TLBの話だったのですが、多粒度TLBといってもピンとこないことが多いので、このたび「Linux Super Page Project」と名前を変更しました。プロジェクトとなっているので、参加者は何人いるのかとたずねられたりしますが、実は私1人だけのプロジェクトです(笑)。Alphaをターゲットにしたプロジェクトから少し範囲を広げて、さまざまなプロセッサアーキテクチャでラージページと同等のTLB効率を目指す、動的なページサイズ可変カーネルを設計するプロジェクトとしていきます。IA-64のような64ビットマシンの方が大規模メモリの要求は大きいと思うので、まずはこういったプロセッサから着手しますが、x86を排除しているわけではなく、他のプロセッサでの可能性も探っていこうと思います。

カンファレンスの発表スライドは、「Linux Super Page Project『Linuxへのページ粒度可変機構の組み込みと性能評価』」(記事末のResource【1】を参照)をご覧ください。

■IBMのPower4について

IBMが発表したPower4マシン「IBM eServer pSeries 690モデル681」には興味が引かれますね。おそらくプロセッサ屋さんなどは、この低誘電率絶縁体やSOI(Sillicon On Insulator)、銅配線などの技術的な面と、オンチップデュアルプロセッサという面に注目しているので、いろいろな雑誌などで解説記事が出るでしょう。そこで、ここではちょっと違う側面から眺めてみます。マシンのそのものの解説はIBMのサイトに行って見てください([2])。

このシステムは、メインフレーム上の業務アプリケーションに対する要求をよく認識している、IBMならではの設計がなされていると思います。LPAR(Logical Partitioning:論理パーティショニング)による論理分割やメモリースループットの大きさは、このマシンで本格的にメインフレーム市場を取ろうという意志が感じられます。LPARで分割したパーティションにLinuxを載せることもできます。Linux Japanの2001年11月号に出ていたナノカーネルも面白い技術ですが、あちらがIBMでいうところのPMA(1980年代のVM/SP HPOの技術)なのに対して、LPARはVM/XAの技術を元に大幅に効率を上げたもので、柔軟性が非常に大きいのが特徴です。論理分割された各パーティションは、比較的簡単なマッピング論理で物理的なリソースのアドレスマッピングを変更して割り当てを受けられ、自分が割り当てられていないリソースはOSから見えなくなります。これにより、仮想マシンのオーバーヘッドは最低限に抑えて高速で柔軟性のあるシステムが構築できます。今まではこのような柔軟性のあるシステムは、同じIBMのメインフレームであるzSeriesくらいしかなかったわけで、(IBMの互換機メーカーである富士通なども同様のシステムを提供していますが)データセンターを集約して効率を上げたい業務用のシステムにはとても良い素材だと思います。

一方、他社はマルチプロセッサマシンの物理分割ができるようにしています。これはPPAR(Physical Partitioning:物理パーティショニング)とIBMは呼んでいます。物理分割では、特に入出力関係の割り当てを設備設置のときに考えておかないと、後からは修正がききません。また、メモリなども物理的に分割されると、パーティションごとに最適なメモリ容量を与えたいという要求に応えることが難しくなります。何しろ物理分割では論理回路として切り離してしまうのですから、切り離しの単位でしか分割できないし、切り離した向こう側にある回路には手出しができなくなるのです。

LPARを実装しているメインフレームでは、サブチャネルの番号をパーティションに割り当ててしまえば入出力のマッピングは比較的容易にできるのに対して、UNIX系のシステムの入出力はメモリマッピングになっていることが多く、普通に考えると一段階余分なアドレス変換をしないと入出力もマッピングできません(メインメモリに関しては、どちらも一段余分なマッピング機能が必要なことには変わりありませんが、メインメモリの割り当ての粒度は比較的粗くて良いのに対して、入出力は小さなメモリをマッピングしなくてはならないのです)。

ところが入出力をPCIに限定してしまえば、パーティションのブート時にPCI BIOSがアドレス割り当てをするため、小さな単位でのマッピングの必要がなくなります。IA-32のアーキテクチャに固執する必要のないマシンにとっては、LPARの入出力部分の設計は実は比較的容易なのかもしれません。PCIの本数はたかがしれているので、割り込みもマッピング機構を用意して割り当てパーティションに直接報告するようにすれば、割り込み応答も遅くならなくてすむのでしょう(ただし、私はハードウェアの設計を見ていないので、これらの話は記事を見たかぎりの推測に過ぎません)。

今分かっている範囲では、LPARへの割り当てはCPUは1台単位、メモリは256Mbytes単位、PCIは1スロット単位ということです。

エンタープライズ部門のLinuxの利用を促進するOSDL([3])も活動をしていることだし、これからも従来汎用機しか選択肢がありえない分野にどんどんLinuxをベースとしたシステムが進出すると面白いですね。

■コンパイラの設計

インタプリタの設計に引き続き、今月号からコンパイラを設計していきます。コンパイラを設計するためにはターゲットプロセッサの命令を決定する必要があります。プロセッサに必要な機能については11月号を参照していただくこととして、ここでは必要な機能をどのように実現していくかを考えましょう。

ハードウェアを簡単にするためにロード・ストアアーキテクチャのRISCにするという話を11月号でしました。そこで、メモリをアクセスする命令としては、

- **ロード** LD
- **ストア** ST

を用意します。

高級言語で必要とされる、フレームポインタをベースアドレスとしたインデックスアクセスを実現するために、インデックスアドレス方式を命令形式として用意することにします。

インデックスアドレス方式は、メモリアクセスを行うときにインデックスレジスタを指定して、インデックスレジスタの値と、ディスプレイスメントと呼ばれる命令コード中に示す数値の和を、メモリのアドレスとして利用するものです。この方式の歴史は古く、IBMの汎用コンピュータSystem/360でもすでに使われていて実績のあるアドレスモードです。System/360の系列ではインデックスとベースの2つのレジスタの和を利用していたのに対して、RISCの走りである801プロジェクトではインデックス側があまり使われていないことに注目してベースレジスタのみの方式に変更しています。そこで、インデックスアドレス方式と呼ばずにベースアドレス方式と呼ぶ場合もありますが、意味は同じです。元々System/360(現在のzSeries)などでは、ベースレジスタはスタックフレームやユーザーページの基底(ベース)アドレスを保持するもので、インデックスレジスタは配列の添字(インデックス)を与えるものとして作られています。しかし、配列の参照のために大部分のレジスタ・メモリタイプの命令にインデックスを指定するようなアーキテクチャにしたことは、やはり無駄が多いと思います。配列参照は一般プログラムでそれほど頻出するものではありませんからね。コンパイラが生成するコードを解析して、ソフトウェアが使う命令だけを用意するというRISCの思想に沿った801プロジェクトでは、インデックスレジスタを廃止してベースレジスタのみでメモリの参照を行うように変更しています。

ディスプレイスメントの指定できる範囲は、ソフトウェアから見たアーキテクチャの使い勝手に影響します。1960年代であればSystem/360の12ビットのディスプレイスメント(変位ともいう。12ビットは4Kbytes相当)も容認できたのですが、ソフトウェアが大型化した現在では使い勝手はあまり良いとはいえません。SPARCやMIPSなどのRISCプロセッサは大抵16ビットのディスプレイスメントが使えるようになっています。32ビットの定数が必要な場合には、上位16ビット値を格納した後下位16ビットとマージするなど手間をかけています。ただ、この方法ではアドレス定数などの情報が複数の命令に分散してしまい、リンカなどの実現が多少トリッキーになります。MIPSなどでは、擬似命令を用意してアドレス定数が勝手な命令間で分割されないようにすることで対応しているようです。完全なアドレスを定数で生成できない以上、12ビットだろうが16ビットだろうが大差ありません。

この問題に対する比較的スマートな方法は、System/360系列と同様に定数自体はメモリ中に書いておき、必要な場合にメモリから読み出す方法です。汎用機のアセンブラをご存知の方なら

```
L 1,=A(label_name)
```

のような記述式と思っていただければ分りやすいのではないでしょうか?

知らない方のために一応解説しておきます。zSeriesではL命令はロード命令を表します。初めの1はロード命令でデータを読み出すレジスタの番号です。=A(label_name)は、label_nameというラベルのアドレスへのポインタをアセンブラが管理するデータ領域に作成して、そのポインタのデータを示すアドレスを表します*1。このように、アセンブラを記述する人(もしくはコンパイラ)は明示的にポインタを意識することなく変位部分では指定できない数値をアセンブラ中で利用することを可能にしています。

もちろん、プログラムの各セクションがどのベースアドレスを利用するかの管理が必要になってきます。それでも複数命令に分断されたアドレス定数にリンカが無理矢理リンク情報の書き換えを行うよりは、ポインタのデータをリンカが書き換えるように設計した方がよりスマートだと思います。

このように、必要な即値データをポインタを仲介して生成するようにすることで、我々のプロセッサでも問題を簡単にすることができます。といっても、ベースレジスタをアセンブラで管理する必要があるので、アセンブラの設計に注意が必要になります。この方式は長期的にこのような対策をとることができるという可能性を示唆するだけで、(私が)これを実現するアセンブラを設計するわけではないことにご注意ください。

さて、前置きが長くなりましたが、ロード命令とストア命令は次のような表記を使うことにします。

```
LD $1, 100($2)
ST $3, 3($0)
```

レジスタ番号を表す数値は$記号を付けます。オペランドの記述は

変位(ベースレジスタ)

の形を取ることにします。命令長は16ビットとしたので、変位の範囲はその半分の8ビットとします。スタックフレームの前をアクセスすることはないので、変位は通常正の値だけでいいのですが、我々は命令数を低減することを狙っているので、変位に2の補数による符号付き2進数を使うことにします。そこで、変位として利用できる範囲は

```
-128〜127
```

となります。

2つ目のオペランドである「変位(ベースレジスタ)」の部分は、アドレスを表すのでアドレスのラベルを記述しても構わないこととします。ただし、この連載で設計するアセンブラではベースレジスタの自動管理をしていないので、ラベルはベースレジスタが$0であることを前提とすることにします。System/360と同じくベースレジスタに$0を指定した場合には、$0はアドレスの加算には用いないことにします。従って、この場合には変位が直接アドレスを与えます。

アドレスを計算する機構を用意したのだから、その結果をプログラムで使うための命令も追加しておきます。それはSystem/360などにもありますが、ロードアドレスという命令です。この命令ではアドレス加算の結果をレジスタに格納するようにします。

ロードアドレス LDA

オペランドはロードアドレスと同じように変位とベースレジスタを指定します。変位が正負両方の値を取れることからこの命令をレジスタに小さな整数を加算するために利用することもできます。例えば、

```
LDA  $3, -4($3)
```

という命令は$3レジスタの内容から4を減算する命令になります。また、ベースレジスタに$0を指定することによってレジスタに小さな整数を設定する命令としても利用できます。

```
LDA  $0, 1($0)
```

という命令は$0レジスタに1を設定する命令になります。

他の命令の組み合わせで実現できる命令を作らないことを前提に、演算器の構成を考えると、表1のようなものがあればよさそうです。演算

【表1】演算器の構成

加算	ADD
論理積	AND
否定	NOT
右シフト	Shift Right(SR)
比較	Set Less Than(SLT)

*1 実際に機械語に展開するときには、変位とベースレジスタとインデックスレジスタ(これは通常0)に変換されます。インデックスレジスタやベースレジスタに0を指定すると、0番のレジスタの内容は無視して0を指定されたインデクスもしくはベースはアドレス加算に使用されません。

のソースとディスティネーションのレジスタを独立して設定できるようにすると、2項演算ではレジスタを3つ、単項演算ではレジスタを2つ設定する命令が必要になります。

減算は引く側の数値を否定演算でビット反転して加算した後、定数の1を加算することで代用できます。定数加算には前述のLDAが利用できるので

```
NOT $2,$2
LDA $2,1($2)
ADD $1,$1,$2
```

のような命令列で減算も実行できます。

論理和その他の論理演算は論理積と否定の組み合わせで実現できます(**リスト1**、**リスト2**)。多少冗長でコードは長くなりますが、このように組み合わせ論理演算はすべて実現できます。

左シフトは加算で代用できます。

```
ADD $1,$1,$1
```

左シフトは2進数で考えれば2倍しているだけなので、加算命令を使って2倍数を作ってあげることで代用できるのです。

次に、減算が命令の組み合わせでできるのに、用意する命令の中に比較が入っているのはおかしいと思うかもしれません。しかし、16ビット整数の数値で表現できる範囲の減算でオーバーフローを起こすと、減算結果で比較を行うのは困難になります(実は、元の数値を保存して、元の数値の符号を併せて判定すれば可能ですが、比較のための命令列は結構厄介になります)。比較命令として用意する命令はSet Less Than という命令です。この命令はMIPSの命令セットにも入っているものですが、2つの数値を比較して第1の数値が第2の数値より小さい場合に、命令の結果として1をディスティネーションレジスタに書き込みます。この条件を満たさない場合にはディスティネーションレジスタには0が書き込まれます。

一般に大小の比較なら引いて負になればよいと簡単に考えるのですが、数値の範囲によってはオーバーフローで符号が変わり、この前提が成り立たなくなります。話だけでは分かりにくいので例題で示してみます。16ビットの符号付整数は2の補数という表現形式で表します。これは表したい数をVとすると、**リスト3**のような式で表される数値を用います。ここで、d0、d1……d15は、それぞれ0ビット目、1ビット目……15ビット目の1/0の数値を表しています。例えば、-1という数値をこの式に当てはめるとd0 = d1 =……= d15 = 1となります(1111111111111111(2))。この形式で表せる最小の数は**リスト4**になります。逆にこの形式で表せる最大の数は**リスト5**になります。比較の演算として32767と-1を比較してみましょう。大小関係は10進で見ると一目瞭然です。比較のために減算を行って結果がどうなるか見てみると**リスト6**のようになってしまいます。C言語ではオーバーフローを無視する仕様になっているので、この結果は間違いではありません。では、**リスト7**の結果はどうなるでしょうか?

減算した結果の符合だけで判定すると、(a-b < 0だから) a<bという結果になるはずです。ところが、比較計算の結果は正しく比較できています。これはCの言語仕様では計算時のオーバーフローを無視するようになっていても、比較演算ではオーバーフローを考えて演算しているからです。

intが32ビットのLinuxで実験するときには、32767の代わりに2147483647を用いてください。このように比較の演算は通常の演算と異なる処理が必要となるので、命令としても別に用意します。

C言語の条件制御文の仕様は条件文の実行結果として、0か0以外かで条件判定を行うようになっています。C言語を実現するためには条件制御文を実現する必要があるのはもちろんです。そこで、比較した結果を使って条件分岐を行う命令が必要になります。条件分岐命令は比較演算の結果が0か0以外になるかによって分岐する命令とします。命令仕様をこのようにすることでC言語との整合性がよくなります。すると、命令数を節約するとすれば、0か0以外で後続命令を切り替える条件分岐命令を1つ用意すればよいことになります。

例えば、if文を実現するには

```
条件計算
条件＝0ならばELSE/NEXTへ
THEN文実行
ELSE/NEXT:
```

のような流れで命令を組み立てます。また、while文では

```
LOOP:
条件計算
条件＝0ならENDへ
WHILE本文実行
GOTO LOOP
END:
```

のように命令を組み立てます。ここで「リテラル:」の形の行はその場所の命令アドレスを示すラベルを表します。これらの例を見て分かることは、条件分岐命令としては、条件計算の結果が0の場合に分岐するものだけでよさそうと

【リスト1】論理和

```
NOT $1,$1
NOT $2,$2
AND $1,$1,$2
NOT $1,$1
```

【リスト2】排他的論理和

```
NOT $0,$1
AND $0,$1,$2
NOT $0,$0
NOT $2,$2
AND $1,$1,$2
NOT $1,$1
AND $1,$0,$1
NOT $1,$1
```

【リスト3】

```
V = -d15*2^(15) + Σ di*2^i
```

【リスト4】

```
1000000000000000(2) = -32768
```

【リスト5】

```
0111111111111111(2) = 32767
```

【リスト6】

```
32767 - (-1) = 32767 + 1 => 1000000000000000(2) == -32768
```

【リスト7】

```
#include<stdio.h>
void main() {
int a,b;

  a = 32767;
  b = -1;
  printf("%d - %d = %d, %d > %d = %d\n", a,b,a-b,
a,b,a>b);
}
```

【表2】条件文の演算結果を加工

1 a<b --> slt
2 a>b --> slt(オペランド順序を変更)
3 a==b --> a-bの結果を0と比較
4 a<=b --> !(a>b)
5 a=>b --> !(a<b)
6 a!=b --> !(a==b)

いうことです。それによって条件分岐命令を1つだけにすることができます。

多くのプロセッサには多種類の比較命令が用意されています。しかし、ここでは比較命令としてはあえて1つだけに絞ります。実際には条件分岐命令で0か0以外を切り分けるようにするので、比較命令と条件分岐命令を組み合わせて多種類の比較演算を行えます。

表2のように条件文の演算結果を加工することによって、よく使う条件文は構成できます。そこで、条件を計算する命令は少なくてすみます。このように比較命令sltがあれば通常の演算命令と併用してC言語に必要となる程度の条件演算は作ることができます。

ここで示したようにslt命令を用意しておけば命令の組み合わせで多くの条件を設定できることが分かります。それでは、条件によりプログラムの動作を変更するにはどうしたら良いでしょうか？　一般に、プロセッサでは条件分岐命令がこの目的に用意されます。我々のプロセッサも条件分岐命令を用意しましょう。

前出のアルゴリズムを観測すれば、必要な条件分岐命令は実は演算結果が0のときに分岐する命令だけであろうということが分かります。そこで、分岐命令として0の場合に分岐する「Branch on Zero」(BZ)命令を用意することにします。分岐先命令のあるアドレスを命令で指定するので、この部分はメモリアクセスの命令と同様に「変位＋ベースレジスタ」で表し、ラベルを使った表記を認めるとします。

```
BZ $1, label
```

とすることで、$1レジスタが0のときにはlabelの示すアドレスに分岐し、0以外のときにはBZ命令の次の命令に実行が移るものとします。

関数の実現のためには、分岐したところに制御が戻ってくる仕組みが必要になります。分岐する先はBZ命令と同様に変位とベースレジスタで指定しますが、戻ってくるためには戻りアドレスをどこかに記憶する必要があります。x86のプロセッサでは戻りアドレスはスタックに記憶することになっていますが、スタック操作を行う命令は多少複雑な動作が必要となります。我々は、他の操作で代用できるものはなるべく代用して命令数を低減することを考えているので、できれば、命令からスタックという専用のメモリ管理に依存するものを避けたいところです。

では、制御を戻すために必要な情報とはなんでしょうか？　実は戻ったときに読み出す命令のアドレスさえ分かっていれば制御を戻すことはそれほど難しくないのです。そこで、関数呼び出しのための命令には戻り番地をどこかに記憶するための機構を用意します。どこにって？

一番簡単に使えるのはレジスタです。呼び出しから戻る情報をリンクと呼び、リンク情報を格納しながら分岐する命令として「Branch and Link」(BAL)命令を用意します。

```
BAL $2, func
```

とすることで、この命令の次の命令アドレス(リンク)を$2レジスタに格納した後、funcの示す命令アドレスに分岐します。

さて、以上をまとめるとコンパイラが命令を生成するために必要な命令セットは表3のようなものになります。

前に書いたように命令コードは16ビット長とします。作成しているコンパイラ言語の整数型は16ビットであり、I形式命令のLDA命令で直接作成できる定数値は8ビットであるため、コンパイラが定数を生成するためにはシフトと加算を組み合わせる必要があります。例えば、238という数値をコンパイラが生成しようとする場合にはI形式で直接表すことのできる-128～127の数値を用いてリスト8のような計算を行います。

MIPSアーキテクチャなどのように、レジスタの上位に命令からの即値を格納する命令(Load High命令)を有するプロセッサもありますが、なるべくハードウェアを小さくするという当初の狙いがあるので敢えて命令の組み合わせで実現させています。

さて、コンパイラのコード生成で利用する命令はほぼ揃いました。これら以外にも入出力の命令である、INやOUT命令などがありますが、命令の説明ばかりしていてもつまらないので一旦説明を終えます。

これらの命令を使ってコンパイラのコード生成を行っていきますが、少し長くなったのでコンパイラのソースコード（今月号の付録CD-ROMに収録）だけ示して、コード生成部分の説明は次号のお楽しみとしたいと思います。それまで掲載したリストを見ながらコンパイラの動作を体験していていただけたら幸いです。

最後に、2月号ではコンパイラの詳細な設計の話とアセンブラの設計について述べていきます。なかなかハードウェアの話にならないと首を長くしている方もいらっしゃると思いますが、しばらくお待ちください。

Linuxで論理回路を作成する場合に利用可能なツールはそれほど多くはないのですが、その中でこの連載ではPARTHENON（[4]）というツールを利用していきます。このツールの記述言語はSFLという言語で、企業などで利用が多いVerilogやVHDLとは異なりますが、言語の違いはあまり問題ではなく、どれか1つでも習得すれば他の言語の理解は容易です。早めにハードウェアで楽しみたい方は、PARTHENONのwebサイトの「その他の情報」というリンクをたどるとツールのダウンロードが可能になっていますので、ご利用ください。

【リスト8】

```
LDA $1,1($0)
ADD $1,$1,$1
ADD $1,$1,$1
ADD $1,$1,$1
ADD $1,$1,$1
ADD $1,$1,$1
ADD $1,$1,$1
ADD $1,$1,$1
LDA $1,110($1)
```

【表3】必要な命令セット

メモリ、即値などを扱う命令群（I形式命令）	LD、ST、LDA、BZ、BAL
レジスタ演算命令群（R形式命令）	ADD、AND、NOT、SR、SLT
汎用レジスタ数	4（$0～$3） ただし、$0はI形式のベースレジスタとして用いるときには値0として扱う。
I形式ディスプレイスメント	8ビット符号付（－128～127）

Resource

[1] Linux Super Page Project「Linuxへのページ粒度可変機構の組み込みと性能評価」
http://shimizu-lab.et.u-tokai.ac.jp/lsp/lc2001/index.htm

[2] IBM新製品ニュース
http://www-6.ibm.com/jp/Products/news/011004/pseries/

[3] OSDL
http://www.osdl.org

[4] PARTHENON
http://www.kecl.ntt.co.jp/parthenon/index_j.htm