■連載開始前に

連載「Alpha/Linux活用講座」が終了したのにまたしゃしゃり出てきたと思われるかもしれませんが、この新しい連載ではプロセッサについて広く一般の話題を取り上げていきたいと思います。インターネットのスピードに雑誌のような媒体は追い付かないのは自明なので、話題性というか、ただの紹介記事のようなものは避けていきます。私のメインマシンは相変らずAlphaなので、その話題にも言及することもあると思います。ちなみにこの原稿も、UP1100マシンに載せたDebian GNU/Linux（potato）上のvim+skkinputで書いています。

最近「理系離れ」だなんだと言われることも多いのですが、今も昔もいろいろな技術に子供達が興味を持つのは変らないと思っています。私もプラモデルから始まって、自分で何か作ってみたいという思いが強かったのですが、ラジコンのように（当時）高価なおもちゃはとても買えませんでした。そのころ出会ったのが、学研の「科学」という雑誌に付録で付いてきたゲルマニウムラジオと電子ブロックです。特に後者はマニュアルに出ている回路を次々と作ってはいろいろと試してみることができ、子供心に強い影響を与えられました。

それと前後して、アマチュア無線をやっていた従兄弟から電子工作に必要な工具や部品をダンボールを1箱送ってもらい、雑誌を参考にしながらラジオやステレオなどを製作することに喜びを覚えていきました。貧乏なので部品の調達はごみ捨て場が主戦場だったわけですが(笑)、近くのスター精密などでロット不良で捨てられた時計の部品から不良でない部品だけ取り出したりと苦労していました。

もっとも、製作なんていっても雑誌に出ている回路図や、場合によっては実体配線図なんてものを手掛かりに半田付けをしていくだけで、本当の意味での電子回路の理解にはほど遠かったわけですけど、何しろ手持ちの部品だけで作るため雑誌に出ている部品は使えず、それなりに回路の知識が必要になりました。

■作る楽しみを

Linux Japanの新連載で何故こんな昔話を始めるのかとお叱りを受けそうですね。でも、もうしばらくお付き合いください。無料で入手できるソフトウェアの登場で、高価なワークステーションでしかできなかったことが手軽にできるようになった今の時代、自分で作ってもしょうがないとあきらめてしまう方が多くなっているのではないかと危惧しています。またLinuxやFreeBSDのカーネルは規模がそれなりに大きいので、基本はそれほど教科書と違わないとしても、手を付ける気にもならない人が多いのではないでしょうか。

ハードウェアにしても、パソコンを自作したという学生の様子を見ていると、自作自体は実体配線図通りに組み立てるあのころの電子工作と似ているなと思うのですが、違うのは考える余地が少ないことです。動かなければ原因を追及せず「相性」のせいにしてしまうなんてことをやっていたら、技術は身に付きません。もっとも、最近の高速バスの不具合の原因を探すには、高価なロジアナを使って慎重に調べるしか方法がないでしょうから、一概に責められません。また、ドライバとハードの不一致などは、仕様が公開されていない以上追及のしようもありませんしね。

しかし、私が子供のころ、不細工ながらも、感電しつつ作ったラジオから外国の放送が聞こえてくることを喜んだような経験は、現在のソフトウェアの世界でもプロセッサの世界でも味わえるはずです。何も、GCCを作らなくてもLinuxを作らなくてもいいではないですか。もっとシンプルで実用性が低くても、「趣味のソフト製作や趣味のCPU製作があっても面白いのでは？」ということで、実は新しい連載ではLinuxを開発システムとしてCPUを作ったり、コンパイラを作ったりという趣味の電子工作、ソフト設計について「も」話をしていきたいと思います。

また、「なぜ？」に重きを置くことを連載のポイントとにしたいと思っています。与えられるものを無批判に受け入れるのではなく、常に「なぜ？」、「どうして？」と無邪気に疑問に思うことが大切だと思うからです。

ということで、連載第1弾は「初めてのコンパイラ製作」とします。ここでは豪華なコンパイラは作りません。電卓＋アルファ程度の簡単なコンパイラを作っていく中で、併せてCPUのアーキテクチャの話もしていきます。

■yacc/lexとは

コンパイラをゼロから作るのは大変ですが、Thomas Niemann氏のサイト（記事末のResource［1］を参照）の例題を元に、プログラム言語風に拡張してみたいと思います。

yacc/lex（yet another compiler compiler/lexical analyzer）というツールをご存知の方は多いと思いますが、常用のツールとしてこれらを使っているケースは少ないのではと思います。compiler compiler（コンパイラ・コンパイラ）という名前に魅かれるものがありますよね。では、コンパイラの機能のうち、yacc/lexはどれだけ自動化してくれるのでしょうか？

実は、あんまり大したことをしてくれないというのが正解です。もちろん、yaccを使った方がずっと楽になるのですが、過大な期待をすると思わぬ作業量の大きさに「こんなはずでは」となります。

コンパイラの仕事は

1 変数や関数の名前の管理とメモリ上の記憶域の割り付け
2 命令列生成
3 命令列最適化

の3つが大きなものになりますが、最後の最適化はどちらかというと力技に近く、「初めての」製作には生成コードが分かりにくくなるなどデメリットもあるので、上の2つだけを取り上げます。

名前の管理についてはyaccは何もしてくれません。lexがトークン（構文の要素）として切り出してくれれば、トークンの番号として把

握しますが、それだけです。これには普通、ハッシュ(名前に演算を施して検索を高速化する技法)などを用いて、名前と記憶域を対応させるシンボルテーブルを作るなどの手法がとられます。実は先程のThomas Niemann氏の例題は、この点で大胆な手法によって名前の管理を大幅に簡略化しています。コンパイラの基本としては十分だと思うので、私もこれに従います。そしてその方法とは、「変数はa-zの26個が大域変数として固定的に与えられる」というものです。大域変数はどの関数からも見える変数で、相対するものとして局所変数という関数内でしか見えない変数があります。

とにかく、最初からプログラムはaからzの26個の変数を持っていて、定義も何もなしで使えると思っていてください。また、Thomas Niemann氏の例題には関数もないので、関数名も不要です(逆に言えば関数がないのだから局所変数などは初めからないわけです)。でも、これはさすがに困るので、この記事では1つだけ関数を許すことにします。関数呼び出しと再帰呼び出しなどの例題も示したいので、関数なしではすまないからです。でも、例題としては1つあれば十分ですね。

そこで、関数名は「foo」に固定し、引数がある場合には引数も1つだけで名前は「arg」に固定することにします。これらの仮定によって、名前を管理しなくてもコード生成が可能になります。もっとも名前を普通に使って、アセンブラにその解決をお願いするという方法もあるのですが、ハンドアセンブル(プログラムではなく人間がアセンブラを機械語に変換すること)を考えると、なるべくシンプルにしてアセンブラが複雑になるのを避けたいところです。もう少し多くの関数を使いたいという方は、例えばf0～f9までの10個の関数が使えるようにするなどの拡張は簡単にできますからぜひ試してみてください。変数名と同じようにlexでパターンマッチングできる範囲ならば比較的簡単に処理ができますし、たかだか10個と分かっていれば、コンパイラにそれだけの配列を始めから用意すればいいので処理は難しくはありません。lexでは

```
"f"[0-9]
```

と書けばいいだけです。もちろん、yaccとコード生成の処理ではこの0～9に相当する関数のバッファを用意して、呼び出し側に応じてバッファを切り分けるように記述する必要がありますが、大して難しい処理ではありません。これは、読者のみなさんへの宿題といたしましょう。

■プロセッサの検討

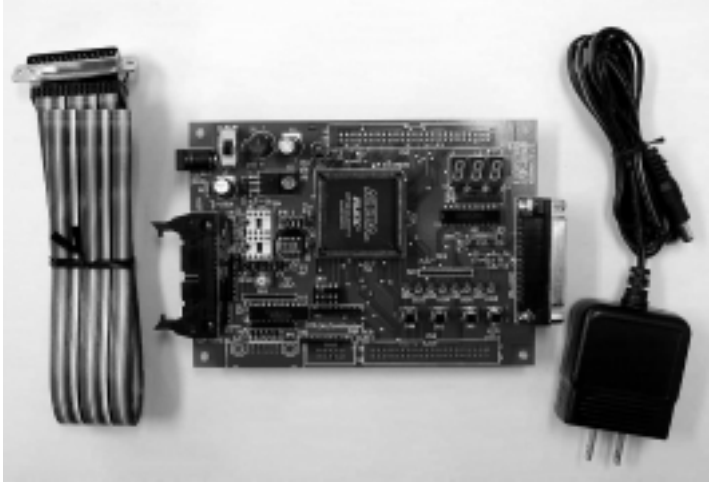
まずはターゲットとなるプロセッサを検討しましょう。アマチュアが実験できる程度のものと考えると、あまり機能を詰め込まないほうがよさそうです。プロセッサはCPLD(Complex Programmable Logic Device)というLSIに論理回路を作成することにします。CPLDは論理回路を書き込めるLSIなので、自分で作成したCPUをその上に実現できるのです。FPGA(Field Programmable Gate Array)も同じように論理回路を書き込めるLSIです。これらのLSIメーカーとしては米国のALTERA社([2])とXILINX社([3])が有名で、どちらもWebサイト上に書き込みや合成に使える無料ソフトウェアを置いています。どちらの製品を使ってもいいのですが、ここでは私が使い慣れているALTERA社のCPLDで話を進めさせてもらいます。小さなCPLDから大規模なものまでいろいろとデバイスはあるのですが、ALTERA社のCPLDでいくとEPF6016Aあたりが安くて使いやすそうです。

ところが、これはTQFPというピン間隔が狭いパッケージを使っているので半田付けが大変そうです。安価で製作しやすいパッケージというと、EPF10K10のPLCC 84pinというのがあります。このPLCCパッケージはソケットが市販されているので、アマチュアの製作用途には優れものの一品です。しかも、PLCCのソケットは10mil(2.54mm)ピッチなので、半田付けも楽勝です。TQFPは0.65mmピッチなので半田付けもプリント基板の作成も難しくなります。

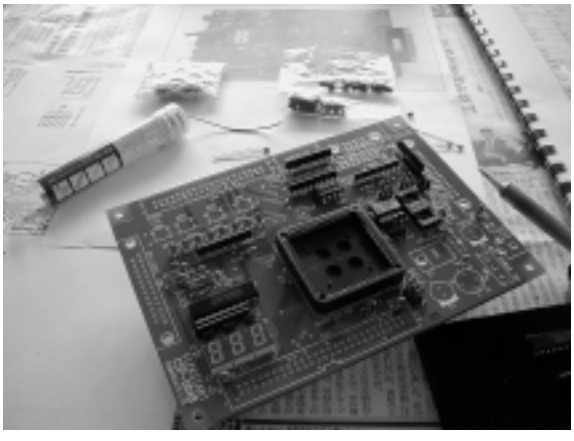
ということで、EPF10K10に載る論理構造を持つCPUをターゲットとします。このCPLDは約1万ゲート相当ということになっていますので、あまり凝った回路は組めません。

ところで、EPF10K10というCPLDにはEAB(Embedded Array Block)という小さなRAMが3個搭載されています。これは、それぞれは256×8のRAMなので、データメモリとしてこれらのRAMを組み合せて用いることで、CPLDの外にメモリを付けなくてもよくなり配線を減らせます。16ビットCPUなので、データに2個使うことにしましょう。命令メモリにも16ビット必要ですが、残りのEABは1個なので8ビットのデータを2回に分けて読み出すことにします。実はデータを2回に分けるようにした方が性能上は有利なのですが、大きなプログラムを組むときに外付けROMを使えるようにしたいという下心もあってこのようにします。外付けROMを使う場合にはジャンクなどで用意に入手できる27C***シリーズを使いたいのですが、これらのEPROMはデータバスが8ビットになっています。16ビットの命令だからと16ビットの命令バスとするとROMを2個使わなくてはならず、コスト的にも負担が増え

【写真1】FLEX10K10ブレッドボードキット(完成品)



【写真2】キット制作中の写真



ますが、それにもまして自由に利用可能なピンの数が減ってしまうのが実験用としてはつらいところです。

なお、余談になりますが、84pin PLCCを持つのはEPF10Kシリーズの中でも1番古いチップだけで、低電圧になった新しいチップはQFP以上のパッケージしかありません。パッケージが大きいので値段の問題があるのだと思いますが、これがなくなるとアマチュアにはちょっと痛いですね。

プリント基板は作りたくはないけれど、試しにCPUを作ってみたいという方は（市販ボードの価格が気になると思いますが）、有限会社ヒューマンデータ（[4]）の「FLEX10K10ブレッドボードキット（CSP-004KIT）」（写真1、写真2、表1）であれば、1万9900円と気軽に工作に利用できる程度の金額で入手できます。PICなどに比べたら当然高いのですが、CPUを丸ごと実現するのに加え、空いた回路を使って周辺装置もこの1個のLSIで実現できるのと、SRAMタイプなのでPCだけあれば書き込み器は不要というのもおいしいメリットだと思います（書き込みケーブルは付属しています）。なお、電源はプラグが2.1 φの7Vから14Vの出力電圧の普通のＡＣアダプターがあれば利用可能です。ダイオードブリッジが入っていて極性を自動調整しているためアダプタの電極の極性はどれでも使えます。私はジャンク箱のアダプターを適当に使っています。

このボードは発光ダイオードやスイッチ、シリアルポートなどがあるのでボードだけで動作確認ができるのもうれしいですね。CPLDの空きピンが少ないので外付けのROMをつけるのは難しいのですが、お手軽に実験ができます。周辺回路をすべて自作する自信のある方は、同じ会社のCSP012kitというボードがもう少し値段も安くいいかもしれません。

なお、FLEX10K10なんかでCPUが作れるのかと思っている方もいるかもしれないので、私のところで実際に動かした写真を示します（写真3）。これは

```
i=0;
while(1) {
 io[0] = i;
 i=i+1;
}
```

を連載中で紹介するコンパイラとアセンブラで作成したファイルを命令メモリに入れて動かしているところです。写真では動きがないので面白くないのですが、LEDが点滅している様子がお分かりでしょうか？　10MHz程度では動くのですが、人間の目には動作が判断できなくなるのでクロック周波数はデバッグ用に非常に低く（300Hz程度）抑えてあります。

ということで、とりあえず、遊びに使える程度のCPUというところを考えて概略スペックを次のように決めます。

1 Load/Store型RISC命令セット
2 命令／データは16bit
3 割り込みレベルは4レベル
4 16bit汎用レジスタ4本
5 命令数10命令程度

さて、命令の詳細や命令フォーマットはコンパイラを検討してから逆に決定していくことにしますので、ここでいったんプロセッサの話を棚上げします。

■コンパイラの仕様

C言語などで用いられる基本的な制御構造としてループと条件判定があります。ループはいくつかのバリエーションがあって、よく用いられるのは`for`文ですが、`do`～`while`や`while`文なども利用されます。また、忘れてはならないものに、再帰呼び出しもループとしての役割を持っていたりします。このようにさまざまなバリエーションをサポートすることも可能ですが、実は「ループする」ことの本質を考えると同じような処理を行う構文であったりします。特に、`for`文と`while`文はループ条件をループ本体開始前にチェックする同一のループ構造を定義しています。であれば、ホビーのためのコンパイラはとりあえず片方があればプログラムを組むのに困らないと言えます。また、`do`～`while`文は利用頻度が低いのと、必要なら`while`文の前に1回実行する文を置けばすむので、ここでは`while`文だけをターゲットにします。再帰呼び出しについては関数を素直に実装すれば自動的にサポートできるので、構文としては特別なことはしません。

では、`while`文はCPUではどのように処理されるのでしょうか？

```
while（条件） ループ本体
```

が、Cの`while`文の構文になります。この文は

1 条件を評価
2 条件が0のときにはループ本体をスキップしてループ脱出
3 ループ本体を実行
4 1に戻る

のように処理します。

CPUの命令としては条件を評価する演算命令と、条件が0のときに命令実行順序を変更する条件分岐命令と、無条件でループ先頭に命令

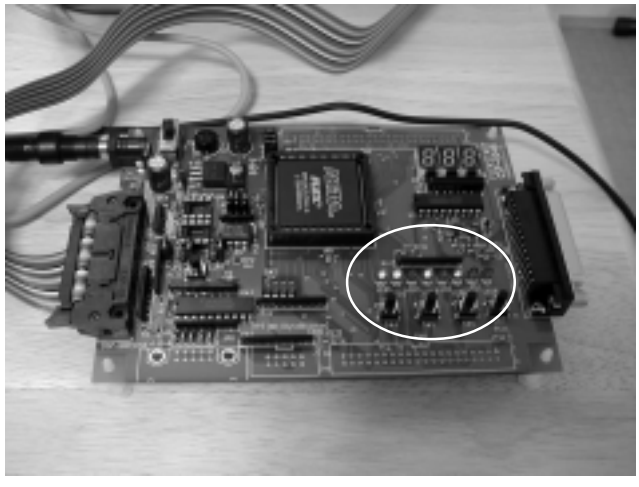
【表1】FLEX10K10ブレッドボードキット（CSP-004KIT）製品構成

内容	・ブレッドボードＣＳＰ－００４ＫＩＴ（EPF10K10LC84-4付属） ・ダウンロードケーブル（ケーブルは組み立て済み） ・取り扱い説明書（回路図含む）
販売価格	1万9900円（税別）
専用ACアダプタ（別売）	6000円（税別）

【リスト1】if文

```
if（条件） 条件実行文 [else else 実行文]
```

【写真3】LEDを点滅させている様子



実行順序を変更するための無条件分岐命令が必要になります。もちろん、その他にループ本体の実行に必要な命令もありますが、構文として必要となるのはこれだけです。

次に条件文であるif文について考えましょう。if文は**リスト1**の形を取ります。「[]」の文はなくてもかまいません。この文は

1 条件を評価
2 条件が0のときは条件実行文をスキップして、elseがあればelse実行文に。なければif文の次に分岐
3 条件実行文を実行
4 elseがあればelse実行文の後に分岐

のように処理します。

CPUの命令としてはwhileと同じように条件を評価する演算命令と、条件が0のときに分岐する条件分岐命令と無条件分岐命令があれば実行できます。

関数呼出しは

```
x = foo([arg]);
```

のように呼出します。引数はなくてもかまいません。呼び出しには呼び出し手順(コーリングシーケンス)を決めておきますが、決めなくてはならないことは

1 引数の渡しかた
2 返り値の受取りかた
3 戻り命令アドレスの渡しかた

などです。レジスタが多いマシンでは、関数内で退避するレジスタと呼び出し側が退避すべきレジスタなど、もう少しいろいろ決めていますが、4つしかレジスタを持たないつもりのターゲットプロセッサではレジスタの余裕はないので、退避が必要なレジスタはすべて呼出側で退避することにします。そこで、

1 引数があれば引数を評価し、引数レジスタに設定
2 関数呼び出し後に使うレジスタ変数があれば退避
3 戻り番地を戻り番地レジスタに設定
4 関数アドレスに分岐
5 返り値レジスタから値を読み出す

のような流れになります。また、呼び出された関数側では

1 退避領域と局所変数があれば局所変数領域を確保
2 戻り番地レジスタを退避
3 引数があれば引数レジスタを退避
4 関数実行
5 戻り番地レジスタを回復
6 局所変数領域と退避領域解放
7 戻り番地へレジスタ相対分岐

のような流れになります。

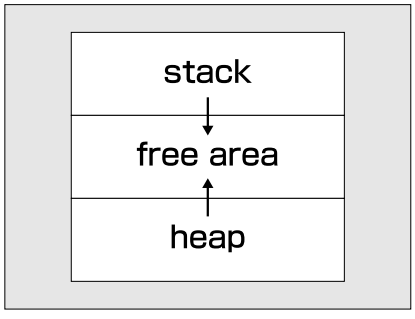
関数の呼び出しでは局所変数や退避領域の確保や解放といったメモリ領域の管理が必要になります。スタックを持つマシンでは通常この領域はスタック上にとって、フレームと呼ぶその関数だけで利用するオブジェクトとして扱います。関数の中で局所変数や引数はフレームのトップからの相対アドレスで参照します。スタックは、メモリ上位から取ることが多いのですが、我々も同じようにメモリ上位から低位へ拡張するスタックを使うことにします(**図1**)。可変長となるスタック領域はあらかじめ使う領域を判定するのは難しく(Javaなどではclassファイルを見れば決まりますが)、空いているメモリ領域の上位から下位に向かって領域の確保を行ない、逆にヒープと呼ぶプログラムから明示的に要求するメモリ領域は下位から上位に向かって領域の確保を行うことで、利用可能なメモリをできるだけ有効に利用できます。

この記事のターゲットプロセッサでもスタックを使うようにします。といっても、スタック専用の命令を用意するのではなく、スタックを実現できるような一般的な命令群を用意するだけです。

フレームの参照、フレームの確保と解放が通常の命令でできるとすると、関数の実現に必要な命令のうち既出でないものは、戻り番地を計算する命令とレジスタ相対分岐になります。レジスタ相対分岐は後で示しますが、通常の無条件分岐の形式を工夫することで同等の命令が実現できます。

戻り番地はこれらに比べてやっかいです。実行中のプログラムカウンタを参照しなければ戻り番地は分からないわけですが、プログラムカウンタは汎用レジスタとは別のレジスタであり、通常の命令では参照できないからです。

【図1】



余談ですが、旧DECのPDP11などは汎用レジスタの1個がプログラムカウンタとなっていて通常命令でプログラムカウンタの操作が可能となっていましたが、プログラムカウンタのために扱える汎用レジスタの数をわざわざ少なくするのは、トランジスタやメモリがふんだんに使える現在ではかえって使いにくくするだけのような気がしますね。

とりあえず、ここでは特殊な命令としてプログラムカウンタを汎用レジスタにコピーしてから、分岐する命令を追加します。

■特殊機能

コンパイラにもう少し特殊な機能を追加しておきましょう。

・メモリ配列
普通の配列は用意しませんが、メモリを直接アクセスする配列を作ります。

```
mem[アドレス]
```

の形でCの一般的な配列のように利用します。

・I/O配列
普通の配列は用意しませんが、I/Oを直接アクセスする配列を作ります。

```
io[アドレス]
```

の形でCの一般的な配列のように利用します。

・割り込み許可
CPUの割り込みマスクを設定する手続きです。

```
intflag_1();  割り込み許可
intflag_0();  割り込み禁止
```

・割り込み復帰
割り込み復帰のための手続きです。0から3の割り込みレベルに対応して4つの手続きを用意します。

```
reti_0();
reti_1();
reti_2();
reti_3();
```

■lexコードの作成

大体のコンパイラの論理構造が決まったところで、lexのコードを作りましょう。前述のEpaperpress社のサンプルを用いて必要な部分を改造していきます。

整数はそれ自身で値を持つので、入力文字列を整数に変換して値を取り出します。変数は変数領域のメモリアドレスに変換するのですが、

aからzまでの変数があるので、変数の文字コードから「a」の文字コードを引くことで、変数領域のメモリアドレスからのオフセットを計算しています。intflagとretiの組み込み関数は複数の関数を1つのトークンで済ませているので、関数の後に付く数値を値としています。これらの関数については、別々に作らなくても1つの関数の引数で処理を切り分ければいいと思う方もいるかもしれませんので、少し解説します。

割り込みフラグは、実はどちらでも構わないのです。しかし引数にするということは、プロセッサの命令から考えると値渡しにすることになるため、レジスタに値を作ってから命令を呼ぶことになります。その方が自由度は高いように見えますが、割り込みを許可するか禁止するかはプログラム作成者は当然分っているわけで、レジスタなど使わずに直接フラグを操作する方が自然なインターフェイスです。また、割り込み復帰の方は割り込んだプログラムへ復帰するのですから、割り込みプログラムの中でレジスタが変更されて戻ると、一般のプログラムからみるとレジスタの値がいつ変更されるか予測できないことになって困ります。そのため、レジスタを1つも使わずに復帰する必要があります。これらの事情から命令中に直接割り込み番号を書いておく命令とします。もちろん、引数にしても定数以外を禁止するという方法もありますが、制限を作るなら元々コンパイラの仕様でできないようにしておいた方が親切です。

これら以外の言語要素は、ただトークンのパターンを検出して、あるラベルで定義された値を返しているだけです。

インクルードファイルの中身はyaccのファイルで使うもので、lexには直接関係ありません。

次回は、yaccで書いたコンパイラの構文を紹介するとともに構文の確認とプログラムのデバッグのために利用可能なインタプリッタの設計を行います。その後、コンパイラとアセンブラの設計を行ってからCPUの論理回路の設計に移ります。

【リスト2】 toyp.l

```
%{
#include <stdlib.h>
#include "toyp.h"
#include "y.tab.h"
void yyerror(char *s);
%}

%%

[a-z]       {
                yylval.sIndex = *yytext - 'a';
                return VARIABLE;
            }

[0-9]+      {
                yylval.iValue = atoi(yytext);
                return INTEGER;
            }

[-()<>=+*/;{}.\[\]] {
                return *yytext;
             }

">="            return GE;
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"return"        return RETURN;
"print"         return PRINT;
"mem"           return MEM;
"io"            return IO;
"arg"           return ARG;
"lo"            return LO;
"void"          return DEF;
"int"           return DEF;
"foo"           return FUNCNAME;
"intflag_"[01]"()"  {
                 yylval.iValue = atoi(yytext+8);
                 return EI;
                }
"reti_"[0-3]"()"    {
                 yylval.iValue = atoi(yytext+5);
                 return RI;
                }

[ \t\n]+        ;       /* ignore whitespace */

.               yyerror("Unknown character");
%%
int yywrap(void) {
    return 1;
}
```

Resource

[1] A Compact Guide to Lex & Yacc
http://www.epaperpress.com/lexandyacc/

[2] ALTERA社Webサイト
http://www.altera.com/

[3] XILINX社Webサイト
http://www.xilinx.com/

[4] 有限会社ヒューマンデータ
http://www.hdl.co.jp/