第1章

GCCの最適化オプション

見本

GCCが出力するオブジェクト・ファイルを，より高速動作するように，またはよりサイズが小さくなるようにすることを「最適化」と呼ぶ．通常は最適化と言った場合は実行速度を高めるものを指す．
GCCはコンパイル時にさまざまな最適化オプションを指定することができる．しかし，最適化オプションの多くは，プログラミングやCPUアーキテクチャに精通していなければ適切に使用することができない．いい加減に指定するとバグを誘発したり，最悪の場合には動作しなくなる．
この章では最適化オプションについて詳しく紹介する．

GCCを手動で最適化してみる

GCCのWebサイトのトップ・ページを**図1**に示します．GCCに関しては，日本語化パッチなどは必要ないため，最新情報はつねにここから入手できます．

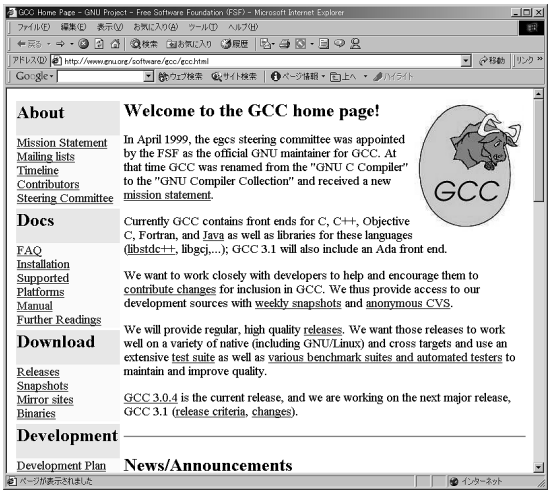

図1　GCCのWebサイトのトップ・ページ
(http://www.gnu.org/software/gcc/gcc.html)

リスト1　test1.c

```
int f()
{
    int   a     =     100;
    int   b     =     200;
    return a-b;
}
```

リスト3　test2.c

```
int f()
{
  register int a     =     100;
  register int b     =     200;
  return a-b;
}
```

まず，簡単なソースをコンパイルして，どのようなアセンブラ・ソースを吐き出すのかを見てみましょう．ここでは，x86用のコード生成を例に説明します．

入力するソースを**リスト1**に示します．Emacsなどを使用して入力してください．

```
gcc -S test1.c
```

と入力してください．吐き出されるアセンブラのコードは**リスト2**のようになります．

このアセンブラ・ソースを見る限りデフォルトではint型変数はスタックされ，レジスタに入らないようです．

では，**リスト3**のようにソースを変更して同じようにアセンブラ・ソースを作成してみてください(**リスト4**)．

リスト2と**リスト4**，二つのアセンブラ・ソースを見比べればわかりますが，変数を意図してレジスタ変

リスト2　test1.s

```
    .file "test2.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl f
    .type  f,@function
f:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $100,-4(%ebp)
    movl $200,-8(%ebp)
    movl -4(%ebp),%eax
    movl -8(%ebp),%ecx
    movl %eax,%edx
    subl %ecx,%edx
    movl %edx,%eax
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  f,.Lfe1-f
    .ident     "GCC: (GNU) 2.95.3 20010315 (release)"
```

リスト4　test2.s

```
    .file "test3.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl f
    .type  f,@function
f:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl $100,%ecx
    movl $200,%edx
    movl %ecx,%ebx
    subl %edx,%ebx
    movl %ebx,%eax
    jmp .L2
.L2:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  f,.Lfe1-f
    .ident     "GCC: (GNU) 2.95.3 20010315 (release)"
```

リスト5　test3.c

```
int f()
{
  register int a asm("ecx")    =    100;
  register int b    =    200;
  return a-b;
}
```

数に指定した後者のソースでは，たしかにレジスタに数値が入力されています．

このように，プログラマが自らソースのコンパイル方法を指定することも最適化の一つです．この機能はグローバル変数，局所変数のどちらにも使用可能ですが，CPUに依存します．

また，**リスト5**，**リスト6**のように，どのレジスタに入れるかという指定もできます．この例ではecxレジスタに入れています．ただし，先のコードに比べてより一層CPUに依存するので注意が必要です．

最適化とは

最適化とは，ソースからオブジェクト・コードを生成する過程において，効率の良いオブジェクト・コードを生成するための手法です．GCCではソース・プログラムを字句解析，構文解析，意味解析した後，中間ファイルとしてアセンブラのコードを出力します．その次にアセンブラのコードを見直して最適化します．

基本はプログラムの実行効率の向上を図ることです．最適化の方法として，以下のものが挙げられます．

- 命令の実行回数を削減する
- RISCチップの場合，1命令に変換できる場合にはコードを書き換える
- 命令実行の高速化

リスト6　test3.s

```
    .file "test4.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl f
    .type  f,@function
f:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl $100,%ecx
    movl $200,%edx
    movl %ecx,%ebx
    subl %edx,%ebx
    movl %ebx,%eax
    jmp .L2
.L2:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  f,.Lfe1-f
    .ident     "GCC: (GNU) 2.95.3 20010315 (release)"
```

リスト7　共通の部分式を置き換える

```
int main(int argc, char* argv[])
{
    int   x;
    int   y;
    int   z;
    int   res;
    x     =     10;
    y     =     20;
    z     =     x     +     y;
/************他の処理*****************/
    res   =     10    *     (x    +     y);          ①
    return      res;
}
```

命令の実行回数を削減するため，以下のようなコードの書き換えが行われます．

● 例1　共通の部分式を置き換える

リスト7のようなコードが考えられます．このコードで①の部分は，

```
res  = 10 * z;
```

と置き換えることができます．

しかし，並列処理や変数zのアドレスがハードウェアのアドレスとリンクしていた場合，その最適化処理はバグを作り出してしまいます．

● 例2　定数を置き換える

リスト8のようなコードが考えられます．このコードは単純に**リスト9**のように書き換えられます．

● 例3　式のループ外への移動

リスト10のようなコードが考えられます．このコードは**リスト11**のように書き換えられます．

やはり並列処理や変数yのアドレスがハードウェアのアドレスとリンクしていた場合，その最適化処理は

リスト8　定数を置き換える

```
int main(int argc, char* argv[])
{
    int   x;
    int   y;
    int   z;
    x     =     10    +     30;
    y     =     20;
    z     =     x     +     y;
/************ほかの処理****************/
    return     z;
}
```

リスト9　定数を置き換えた結果

```
int main(int argc, char* argv[])
{
/************ほかの処理****************/
    return     10 + 30 +20;
}
```

リスト10　式のループ外への移動

```
for(ix=0;ix<_Max;ix++)
{
    x     =     10;
    y     =     x     *     20;
    ・・・・・
}
```

リスト11　式をループ外へ移動した結果

```
y   =     200;
for(ix=0;ix<_Max;ix++)
{
    ・・・・・
}
```

リスト12　冗長なループを変換する

```
for(ix=0;ix<_Max;ix++)
{
    a[ix] =     0;
}
for(ix=0;ix<_Max;ix++)
{
    b[ix] =     0;
}
```

リスト13　冗長なループを変換した結果

```
for(ix=0;ix<_Max;ix++)
{
    a[ix] =     0;
    b[ix] =     0;
}
```

バグを作り出してしまいます.

● 例4　冗長なループを変換する

リスト12のようなコードが考えられます. このコードは**リスト13**のように書き換えられます.

● 例5　演算の置き換え

次のようなコード,

```
y = x ^ 2;
```

は,

```
y = x * x
```

と等価で，このほうが効率良く処理できるため書き換えられます.

最適化のオプション

コンパイラ・オプションによる自動最適化について解説します.

● -0，-01

このオプションでは基本的な最適化を行います.

たとえばregisterの不要な移行を行わない，メモリ・アクセスを減らすためにregisterに値を保存しておく，などの最適化を行います.

最適化を行うと，コンパイルに時間がかかるうえ，大きな関数についてはたくさんのメモリを余計に使います.

-0を指定しないと，register宣言した変数しかレジスタに割り当てられません.

-0を指定することでコード・サイズと実行時間を小さくしようとしています．また，全機種で-fthread-jumpsと-fdefer-popを有効にします．遅延スロットのある機種では-fdelayed-branchをONにし，フレーム・ポインタなしでもデバッグをサポートできる機種では-fomit-frame-pointerをONにしています．機種によっては，ほかのオプションをONにするものもあります.

フレーム・ポインタに関して補足します．C/C++では関数が呼び出された際に，スタックに積まれた引き数を指し示すためのフレーム・ポインタを設定してあります．しかし，このフレーム・ポインタは省略可能であり，そうすることによってバイナリの最適化が図れます．ただし，デバッガを使ってのバイナリ・ファイルのデバッグは不可能になります.

● -02

コンパイラは，ほとんどすべての最適化を実行します．-02を指定した場合，ループ展開や関数のインライン展開を行いません．-0と比べると，このオプションはコンパイル時間が増えますが，生成コードの効率が良くなります.

-02を指定すると，ループ展開と関数のインライン展開，それに厳密なエイリアシングを除いたすべての最適化が有効になります．また，すべての機種は-fforce-memオプションを付け，デバッグと干渉しない機種ではフレーム・ポインタの削除を行います.

● -03

-03は，-02で指定されるすべての最適化を行い，かつinline-functionsオプションを行います．inline-functionsオプションについては後述します．

● -00

最適化を行いません．gdbが正しく動作しない場合には，このオプションを付けることでデバッグができるようになります．

● -0s

サイズについて最適化します．-02の最適化をすべて有効にし，結果としてサイズが大きくなる最適化を行わなければならない場合には最適化を中止します．

-0オプションを複数指定した場合は，最適化レベルの数字が付いていてもいなくても，最後に指定したオプションが有効になります．

-fflagという形のオプションは機種独立のフラグを指定します．ほとんどのフラグには，肯定形と否定形があります．-ffooの否定形は-fno-fooです．

● -ffloat-store

浮動小数点変数をレジスタにストアしないようにし，浮動小数点値をレジスタもしくはメモリから取り出すかどうかを変更する可能性のあるオプションを禁止します．

このオプションを指定すると，68000のようなプロセッサで好ましくない余分な精度を使わないようになります．68000では，浮動小数点レジスタはdoubleがもつと想定されているよりも余分の精度を保持しています．インテルのCPUについても同様です．いくつかのプログラムはIEEE浮動小数点の厳密な定義を想定しており，そういうプログラムに対しては，適当な中間計算結果をすべて変数に格納するようにプログラムを修正しておけば，-ffloat-storeを使うことができます．

Cのdouble型は64ビットを符号1ビット/指数部11ビット/仮数部52ビットとして浮動小数点形式を表現しています．double型では，仮数部は2の53乗の数を表現することができますが，たいていのCPUでは，浮動小数点レジスタの精度は80ビットです．その分の精度がむだだと思う場合は，-ffloat-storeのオプションを選択します．

x86アーキテクチャでこのオプションを選択しない場合，メモリ・アクセスが多少増加する可能性があるため，筆者としてはこのオプションは指定しないほうが良いと思います．どの最適化オプションでもそうなのですが，可搬性と実行速度を秤にかけることになります．ただし，ROM上で動くものを作る場合，可搬性を犠牲にしてサイズと実行速度を追求することになるだろうと思います．そんなときに試行錯誤し，吐き出したアセンブラ・ソースを見て最適化を検討することが必要となります．

● -fno-defer-pop

関数呼び出しのたびに，つねにその関数から戻るとすぐ引き数をPOPします．関数呼び出しの後に引き数をPOPしなければならない機種では，GCCは普通は複数の関数呼び出しについてスタックに引き数を蓄積し，それらをすべて一度にPOPします．

これはハードウェアのアドレスを共有して実行するような場合や，そのオブジェクトの永続性が短い関数から戻った場合などに指定するべきオプションです．

● -fforce-mem

メモリ・オペランドについて算術演算を行う前に，そのメモリ・オペランドをレジスタに強制的にコピーさせます．この結果，すべてのメモリ参照を潜在的な共通部分式とすることにより，生成コードが良くなる可能性があります．これらのメモリ参照が共通部分式でない場合は，命令組み合わせフェーズが独立したレジスタへのロードを削除しなければなりません．-02を指定すると，このオプションが有効になります．

筆者には，このオプションは可搬性に有害なだけで最適化の意味があまりないように思えます．

● -fforce-addr

メモリ中のアドレス定数を，それについて算術演算を行う前にレジスタにコピーします．処理速度が多少速くなる場合もあります．

メモリ中の数値を演算するよりも，レジスタに格納された数値を演算するほうがもちろん速くなります．このオプションは試してみる価値があると思います．

● -fomit-frame-pointer

フレーム・ポインタを必要としない関数は，フレーム・ポインタをレジスタにもちません．これによって，フレーム・ポインタをセーブ，設定，リストアする命令をなくすことができ，多くの関数で利用可能なレジスタが一つ増えます．しかし，機種によってはデバッグが不可能になってしまいます．

フレーム・ポインタを扱わないことについてユーザがわかっているのであれば，コードのむだが省かれるので，試す価値のあるオプションだと思います．

リスト14　specs

```
*asm:
%{v:-V} %{Qy:} %{!Qn:-Qy} %{n} %{T} %{Ym,*} %{Yd,*} %{Wa,*:%*}

*asm_final:
%|

*cpp:
%(cpp_cpu) %{fPIC:-D__PIC__ -D__pic__} %{fpic:-D__PIC__ -D__pic__} %{posix:-D_POSIX_SOURCE} %{pthread:-D_REENTRANT}

*cc1:
%(cc1_cpu) %{profile:-p}

*cc1plus:


*endfile:
%{!shared:crtend.o%s} %{shared:crtendS.o%s} crtn.o%s

*link:
-m elf_i386 %{shared:-shared}   %{!shared:     %{!ibcs:       %{!static:           %{rdynamic:-export-dynamic}
                                               %{!dynamic-linker:-dynamic-linker /lib/ld-linux.so.2}}   %{static:-static}}}

*lib:
%{shared: -lc}    %{!shared: %{mieee-fp:-lieee} %{pthread:-lpthread}     %{profile:-lc_p} %{!profile: -lc}}

*libgcc:
-lgcc

*startfile:
%{!shared:      %{pg:gcrt1.o%s} %{!pg:%{p:gcrt1.o%s}                    %{!p:%{profile:gcrt1.o%s}
%{!profile:crt1.o%s}}}}    crti.o%s %{!shared:crtbegin.o%s} %{shared:crtbeginS.o%s}

*switches_need_spaces:


*signed_char:
%{funsigned-char:-D__CHAR_UNSIGNED__}

*predefines:
-I/usr/include -D__ELF__ -Dunix -D__i386__ -Dlinux -Asystem(posix)

*cross_compile:
0

*version:
2.95.3
```

● -fno-inline

キーワードinlineを無視します．このオプションは，関数のインライン展開をいっさい行わせないために使われます．最適化を行わない場合には，どの関数もインライン展開されません．こうした最適化の方法は，ソースに書かれたものを無視するので混乱の元になる可能性があります．

キーワードinlineを無視したいのであれば，ソース上で実際にキーワードinlineを削除したほうが可読性も高まり，可搬性も高まるでしょう．

● -finline-functions

単純な関数をすべて呼び出し元のコードに置きます．ある関数に対するすべての呼び出しが統合される場合，その関数がstaticと宣言されていれば，普通はその関数はアセンブラ・コードとして出力されません．

このオプションで指定するのも良いと思いますが，キーワードinlineを指定したほうが可読性も高まり，可搬性も高まると思います．

● -fkeep-inline-functions

ある関数に対する呼び出しが全部統合され，その関数がstaticと宣言されている場合でも，実行時に呼び出し可能な関数を別個に出力します．このオプションは，extern inline宣言された関数には影響しません．

このオプションに関しては，混乱を招きかねないので注意が必要です．ソース上でプログラマが意図しているのがstaticであるなら，それにしたがわないと障害が起きる可能性があります．

● -fno-function-cse

関数のアドレスをレジスタに置かないようにします．ある関数を呼び出す命令は，それぞれ関数のアドレスを明示的に保持するようにします．

このオプションは効率の低いコードを生成しますが，アセンブラ出力ソースを書き換えるようなことを行う場合には，このオプションを使用しないと混乱し

```
*multilib:
. ;

*multilib_defaults:


*multilib_extra:


*multilib_matches:


*linker:
collect2

*cpp_486:
%{!ansi:-Di486} -D__i486 -D__i486__

*cpp_586:
%{!ansi:-Di586 -Dpentium}      -D__i586 -D__i586__ -D__pentium -D__pentium__

*cpp_k6:
%{!ansi:-Di586 -Dk6}      -D__i586 -D__i586__ -D__k6 -D__k6__

*cpp_686:
%{!ansi:-Di686 -Dpentiumpro}   -D__i686 -D__i686__ -D__pentiumpro -D__pentiumpro__

*cpp_cpu_default:
%(cpp_586)

*cpp_cpu:
-Acpu(i386) -Amachine(i386) %{!ansi:-Di386} -D__i386 -D__i386__ %{mcpu=i486:%(cpp_486)} %{m486:%(cpp_486)}
                  %{mpentium:%(cpp_586)} %{mcpu=pentium:%(cpp_586)} %{mpentiumpro:%(cpp_686)} %{mcpu=pentiumpro:%(cpp_686)}
                                                  %{mcpu=k6:%(cpp_k6)} %{!mcpu*:%{!m486:%{!mpentium*:%(cpp_cpu_default)}}}

*cc1_cpu:
%{!mcpu*: %{m386:-mcpu=i386 -march=i386} %{m486:-mcpu=i486 -march=i486} %{mpentium:-mcpu=pentium}
                                                                                %{mpentiumpro:-mcpu=pentiumpro}}

*link_command:
%{!fsyntax-only:  %{!c:%{!M:%{!MM:%{!E:%{!S:%(linker) %l %X %{o*} %{A} %{d} %{e*} %{m} %{N} %{n}
%{r} %{s} %{t} %{u*} %{x} %{z} %{Z}            %{!A:%{!nostdlib:%{!nostartfiles:%S}}}                          %{stat-
ic:} %{L*} %D %o              %{!nostdlib:%{!nodefaultlibs:%G %L %G}}                          %{!A:%{!nostdlib:%{!nostart-
files:%E}}}              %{T*}
 }}}}}}
```

てしまいます.

● -ffast-math

このオプションを指定すると，実行速度を最適化するという観点から，ANSIやIEEEの規則や仕様をある面で破ることをコンパイラに許可します.たとえば，このオプションを指定すると，コンパイラは，sqrt関数の引き数が負にならないとか，浮動小数点値がNaNになることはないという仮定を行います.

このオプションは大前提を崩す恐れがあるので，使用しないほうが良いでしょう.

オプション実行のカスタマイズ

いくつか最適化の例を挙げました．そのほかにも，プログラム開発作業でマシンを使用する数人のユーザが，同一の環境でコンパイルしたいというときに，オプションはこれを使ってくださいと伝えるのも，使い勝手が悪くミスを誘発する元になります.

その場合，/usr/lib/gcc-lib/i586-pc-Linux/2.95.3（TurboLinux7の場合）のディレクトリ下にあるspecsを編集してください．specsの例を**リスト14**に示します．**リスト14**にあるように，

```
*cc1:
```

と，デフォルト・フラグは何もありません．もし，Cコードのコンパイルでいつも“ -m486 ”を使いたければ，下記のように変更を加えます.

```
*cc1:
- -m486
```

このようにすれば，一つのマシンでプログラム開発を数名で行う際の整合性がとれます.

GCCの最適化オプション——Cとアセンブラの比較

GCCの最適化オプションについて，さらに詳しく説明します．コンパイルされたアセンブラ・ソースと元のCソースを比較して，どのような最適化が行われ

リスト15　test00.c

```
#include  <stdio.h>
main(int  argc,char* argv[])
{
    register double i=4503599627370496e+11;
    printf("%e¥n",i);
    return;
}
```

ているか見てもらいたいと思います．

最適化オプションの出力

● -ffloat-store

まず，最適化オプション-ffloat-storeを指定すると吐き出したアセンブラ・コードがどのように変化するのかを見てみましょう．

検証用のコードを**リスト15**に示します．

このコードをオプションなしでコンパイルすると，**リスト16**のようなアセンブラ・コードを吐き出します．

ここではbpレジスタに浮動小数点値をセットしていますが，セットしたくない場合もあります．その場合，オプションを-ffloat-storeにすると，**リスト17**のようなアセンブラ・コードを出力します．

● -fno-defer-pop

次に-fno-defer-popオプションを付けた場合について検証してみましょう．

検証用のコードを**リスト18**に示します．

このコードをオプションなしでコンパイルすると，**リスト19**のようなアセンブラ・コードを出力します．

関数の引き数が参照渡しの場合，関数呼び出し後に無条件にPOPされますが，この場合は関数呼び出し後に値をスタックしているのがわかります．

では，-fno-defer-popオプションを付けた場合を**リスト20**に示します．

関数呼び出し後にaレジスタの値をbレジスタに蓄積しています．

● -fforce-mem

次に，-fforce-memオプションを付けた場合につ

リスト16　test00.s

```
3    .file "test00.c"
     .version   "01.01"
gcc2_compiled.:
.section        .rodata
.LC1:
     .string    "%e¥n"
     .align 8
.LC0:
     .long 0xe8000000,0x45774876
.text
     .align 4
.globl main
     .type  main,@function
main:
     pushl %ebp
     movl %esp,%ebp
     subl $24,%esp
     fldl .LC0
     fstpl -8(%ebp)
     addl $-4,%esp
     pushl -4(%ebp)
     pushl -8(%ebp)
     pushl $.LC1
     call printf
     addl $16,%esp
     jmp .L2
.L2:
     movl %ebp,%esp
     popl %ebp
     ret
.Lfe1:
     .size       main,.Lfe1-main
     .ident     "GCC: (GNU)
               2.95.3 20010315 (release)"
```

リスト17　test00.s（-ffloat-store）

```
     .file "test00.c"
     .version   "01.01"
gcc2_compiled.:
.section  .rodata
.LC1:
     .string    "%e¥n"
     .align 8
.LC0:
     .long 0xe8000000,0x45774876
.text
     .align 4
.globl main
     .type       main,@function
main:
     pushl %ebp
     movl %esp,%ebp
     subl $24,%esp
     fldl .LC0
     fstpl -8(%ebp)
     addl $-4,%esp
     fldl -8(%ebp)
     subl $8,%esp
     fstpl (%esp)
     pushl $.LC1
     call printf
     addl $16,%esp
     jmp .L2
     .p2align 4,,7
.L2:
     movl %ebp,%esp
     popl %ebp
     ret
.Lfe1:
     .size       main,.Lfe1-main
     .ident     "GCC: (GNU)
               2.95.3 20010315 (release)"
```

リスト18　test01.c

```
#include  <stdio.h>
main(int  argc,char* argv[])
{
    void  func01(int x,int y,int z);
    void  func02(int x,int y,int z);
    void  func03(int x,int y,int z);
    void  func04(int x,int y,int z);
    int   z1    = 0;
    int   z2    = 0;
    int   z3    = 0;
    int   z4    = 0;
    func01(100,150,z1);
    func02(300,150,z2);
    func03(10,150,z3);
    func04(1000,100,z4);
    return;
}
int func01(int x,int y,int z)
{
    z=    x+y;
}
int func02(int x,int y,int z)
{
    z=    x-y;
}
int func03(int x,int y,int z)
{
    z=    x*y;
}
int func04(int x,int y,int z)
{
    z=    x/y;
}
```

いて検証してみましょう．

同じく検証用のコードを**リスト21**に示します．

このコードをオプションなしでコンパイルすると**リスト22**のようなアセンブラ・コードを出力します．同一のメモリ・アドレスでも別のレジスタに格納されています．

では，-fforce-memオプションを付けた場合の出力を**リスト23**に示します．

同一のメモリ・アドレスは，同一のレジスタに一度だけ格納されています．問題は固定値で表されたメモ

リスト19　test01.s

```
    .file "test01.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    addl $-4,%esp
    pushl $0
    pushl $150
    pushl $100
    call func01
    addl $-4,%esp
    pushl $0
    pushl $150
    pushl $300
    call func02
    addl $32,%esp
    addl $-4,%esp
    pushl $0
    pushl $150
    pushl $10
    call func03
    addl $-4,%esp
    pushl $0
    pushl $100
    pushl $1000
    call func04
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .align 4
.globl func01
    .type  func01,@function
func01:
    pushl %ebp
    movl %esp,%ebp
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe2:
    .size  func01,.Lfe2-func01
    .align 4
.globl func02
    .type  func02,@function
func02:
    pushl %ebp
    movl %esp,%ebp
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe3:
    .size  func02,.Lfe3-func02
    .align 4
.globl func03
    .type  func03,@function
func03:
    pushl %ebp
    movl %esp,%ebp
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe4:
    .size  func03,.Lfe4-func03
    .align 4
.globl func04
    .type  func04,@function
func04:
    pushl %ebp
    movl %esp,%ebp
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe5:
    .size  func04,.Lfe5-func04
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト20　test01.s（-fno-defer-pop）

```
    .file "test01.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $0,-4(%ebp)
    movl $0,-8(%ebp)
    movl $0,-12(%ebp)
    movl $0,-16(%ebp)
    addl $-4,%esp
    movl -4(%ebp),%eax
    pushl %eax
    pushl $150
    pushl $100
    call func01
    addl $16,%esp
    addl $-4,%esp
    movl -8(%ebp),%eax
    pushl %eax
    pushl $150
    pushl $300
    call func02
    addl $16,%esp
    addl $-4,%esp
    movl -12(%ebp),%eax
    pushl %eax
    pushl $150
    pushl $10
    call func03
    addl $16,%esp
    addl $-4,%esp
    movl -16(%ebp),%eax
    pushl %eax
    pushl $100
    pushl $1000
    call func04
    addl $16,%esp
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .align 4
.globl func01
    .type  func01,@function
func01:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    movl %ecx,16(%ebp)
.L3:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe2:
    .size  func01,.Lfe2-func01
    .align 4
.globl func02
    .type  func02,@function
func02:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    movl %eax,%ecx
    subl %edx,%ecx
    movl %ecx,16(%ebp)
.L4:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe3:
    .size  func02,.Lfe3-func02
    .align 4
.globl func03
    .type  func03,@function
func03:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    imull 12(%ebp),%eax
    movl %eax,16(%ebp)
.L5:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe4:
    .size  func03,.Lfe4-func03
    .align 4
.globl func04
    .type  func04,@function
func04:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    leal 12(%ebp),%ecx
    cltd
    idivl (%ecx)
    movl %eax,16(%ebp)
.L6:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe5:
    .size  func04,.Lfe5-func04
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト21　test02.c

```
#include <stdio.h>
main(int argc,char* argv[])
{
    int* pt   =    (long)0xbffffcb8;
    int* pt1  =    (long)0xbffffcb8;
    int* pt2  =    (long)0xbffffcb8;
    int* pt3  =    (long)0xbffffcb8;
    *pt  =    (int)1;
    printf("%d¥n",pt);
    printf("%d¥n",pt1);
    printf("%d¥n",pt2);
    printf("%d¥n",pt3);
    return;
}
```

リ・アドレスの内容が外部のハードウェアによって更新される場合などに，最適化が意図しないものになってしまうことです．

● -fomit-frame-pointer

今度は-fomit-frame-pointerオプションを付けた場合について検証してみましょう．

同じく検証用のコードを**リスト24**に示します．

このコードをオプションなしでコンパイルすると，**リスト25**のようなアセンブラ・コードを出力します．

フレーム・ポインタであるEBPレジスタに値がセットされています．

では，-fomit-frame-pointerオプションを付けた場合の出力を**リスト26**に示します．

リスト24　test03.c

```
#include <stdio.h>
main(int argc,char* argv[])
{
    register int    pt0      = 10;
    register int    pt1      = 20;
    register int    pt2      = 30;
    register int    pt3      = 40;
    register int    pt4      = 50;
    register int    pt5      = 60;
    register int    pt6      = 70;
    register int    pt7      = 80;
    register int    pt8      = 100;
    register int    pt9      = 200;
    register unsigned    long pt10 =    4294967290UL;
    register unsigned    long pt11 =    4294967290UL;
    register unsigned    long pt12 =    4294967290UL;
    register unsigned    long pt13 =    4294967200UL;
    register unsigned    long pt14 =    4294967290UL;
    return;
}
```

フレーム・ポインタであるEBPレジスタを使わずにスタック・ポインタで指し示される領域に値が設定されています．

ターゲット・マシンによっては，このオプションを指定しても何も変化しません．VAXなどはフレーム・ポインタを自動的に扱うため，プログラマが意識する必要がないからです．マクロFRAME_POINTER_REQUIREDの式の値がゼロであれば，このオプション指定に意味はありません．

リスト22　test02.s

```
    .file "test02.c"
    .version    "01.01"
gcc2_compiled.:
.section  .rodata
.LC0:
    .string    "%d¥n"
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $-1073742664,-4(%ebp)
    movl $-1073742664,-8(%ebp)
    movl $-1073742664,-12(%ebp)
    movl $-1073742664,-16(%ebp)
    movl -4(%ebp),%eax
    movl $1,(%eax)
    addl $-8,%esp
    movl -4(%ebp),%eax
    pushl %eax
    pushl $.LC0
    call printf
    addl $16,%esp
    addl $-8,%esp
    movl -8(%ebp),%eax
    pushl %eax
    pushl $.LC0
    call printf
    addl $16,%esp
    addl $-8,%esp
    movl -12(%ebp),%eax
    pushl %eax
    pushl $.LC0
    call printf
    addl $16,%esp
    addl $-8,%esp
    movl -16(%ebp),%eax
    pushl %eax
    pushl $.LC0
    call printf
    addl $16,%esp
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト23　test02.s(-fforce-mem)

```
    .file "test02.c"
    .version    "01.01"
gcc2_compiled.:
.section  .rodata
.LC0:
    .string    "%d¥n"
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $20,%esp
    pushl %ebx
    movl $-1073742664,%ebx
    movl $1,-1073742664
    addl $-8,%esp
    pushl %ebx
    pushl $.LC0
    call printf
    addl $-8,%esp
    pushl %ebx
    pushl $.LC0
    call printf
    addl $32,%esp
    addl $-8,%esp
    pushl %ebx
    pushl $.LC0
    call printf
    addl $-8,%esp
    pushl %ebx
    pushl $.LC0
    call printf
    movl -24(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト25　test03.s

```
    .file "test03.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $44,%esp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl $10,%eax
    movl $20,%edx
    movl $30,%ecx
    movl $40,%ebx
    movl $50,%esi
    movl $60,%edi
    movl $70,-4(%ebp)
    movl $80,-8(%ebp)
    movl $100,-12(%ebp)
    movl $200,-16(%ebp)
    movl $-6,-20(%ebp)
    movl $-6,-24(%ebp)
    movl $-6,-28(%ebp)
    movl $-96,-32(%ebp)
    movl $-6,-36(%ebp)
    jmp .L2
.L2:
    leal -56(%ebp),%esp
    popl %ebx
    popl %esi
    popl %edi
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト26　test03.s（-fomit-frame-pointer）

```
    .file "test03.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    subl $48,%esp
    pushl %edi
    pushl %esi
    pushl %ebx
    movl $10,%eax
    movl $20,%edx
    movl $30,%ecx
    movl $40,%ebx
    movl $50,%esi
    movl $60,%edi
    movl $70,44(%esp)
    movl $80,40(%esp)
    movl $100,36(%esp)
    movl $200,32(%esp)
    movl $-6,28(%esp)
    movl $-6,24(%esp)
    movl $-6,20(%esp)
    movl $-96,16(%esp)
    movl $-6,12(%esp)
    jmp .L2
.L2:
    popl %ebx
    popl %esi
    popl %edi
    addl $48,%esp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

より高度な最適化オプション

最適化オプションはまだあります．以下は，より高度な最適化を行いたいという場合に使うものです．

● -fstrength-reduce

ループの強度削減と繰り返し変数の削除の最適化を実行します．ループの外に出せる計算は外に出して，ステップ数を減らします．

元のCソースを**リスト27**に，最適化オプションなしのアセンブラ・ソースを**リスト28**に，最適化オプション付きのアセンブラ・ソースを**リスト29**に示します．

この例では変数jとkの積が不変になるはずなので，最適化後は即値30（5と6の積）をレジスタにセットしています．最適化前はそのままループの中で乗算しています．

リスト27　test04.c

```
#include <stdio.h>
int main()
{
    int  iTbl[3];
    int  max=3;
    int i;
    int j =    5;
    int k =    6;
    for(i = 0; i < max; i++)
    {
        iTbl[i] = j * k;
    }
    return 0;
}
```

● -fthread-jumps

条件分岐の後に別の条件分岐があり，その比較が最初の比較に包括されるものかどうかを検査する最適化

リスト28　test04.s

```
    .file "test04.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $40,%esp
    movl $3,-16(%ebp)
    movl $5,-24(%ebp)
    movl $6,-28(%ebp)
    movl $0,-20(%ebp)
    .p2align 4,,7
.L3:
    movl -20(%ebp),%eax
    cmpl -16(%ebp),%eax
    jl .L6
    jmp .L4
    .p2align 4,,7
.L6:
    movl -20(%ebp),%eax
    movl %eax,%edx
    leal 0(,%edx,4),%eax
    leal -12(%ebp),%edx
    movl -24(%ebp),%ecx
    imull -28(%ebp),%ecx
    movl %ecx,(%eax,%edx)
.L5:
    incl -20(%ebp)
    jmp .L3
    .p2align 4,,7
.L4:
    xorl %eax,%eax
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト29　test04.s（-fstrength-reduce）

```
    .file "test04.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $30,%edx
    movl $2,%ecx
    leal -4(%ebp),%eax
    .p2align 4,,7
.L21:
    movl %edx,(%eax)
    addl $-4,%eax
    decl %ecx
    jns .L21
    xorl %eax,%eax
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト30　test05.c

```
#include <stdio.h>
int main()
{
    int res    =    0;
    int i =    0;
    if    (i    ==    0)
    {
         if    (i    <=    0    )
         {
              res   =    1;
         }
         if    (i    ==    0    )
         {
              res   =    1;
         }
         if    (i    >=    0    )
         {
              res   =    0;
         }
    }
    return     res;
}
```

リスト32　test05.s（-fthread-jumps）

```
    .file "test05.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    xorl %eax,%eax
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU) 2.95.3 20010315 (release)"
```

リスト31　test05.s

```
    .file "test05.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $0,-4(%ebp)
    movl $0,-8(%ebp)
    cmpl $0,-8(%ebp)
    jne .L3
    cmpl $0,-8(%ebp)
    jg .L4
    movl $1,-4(%ebp)
.L4:
    cmpl $0,-8(%ebp)
    jne .L5
    movl $1,-4(%ebp)
.L5:
    cmpl $0,-8(%ebp)
    jl .L3
    movl $0,-4(%ebp)
.L6:
.L3:
    movl -4(%ebp),%edx
    movl %edx,%eax
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

を実行します．もし包括されるものなら，最初の条件分岐先は，二番目の条件分岐先かその直後のどちらかに，二番目の比較条件の真偽が既知かどうかにしたがって変更されます．

元のCソースを**リスト30**に，最適化オプションなしのアセンブラ・ソースを**リスト31**に，最適化オプション付きのアセンブラ・ソースを**リスト32**に示します．

この例では変数iは必ずゼロです．ネストした条件の真中にしか分岐しません．最適化後は単純明快に1をレジスタにセットして戻しています．最適化前はそのまま比較を繰り返しています．

● -fcse-follow-jumps

共通部分式削除（CSE）の際に，条件分岐命令のうち，その条件分岐先にほかの経路を通って到達することがないものを探し出します．

元のCソースを**リスト33**に，最適化オプションなしのアセンブラ・ソースを**リスト34**に，最適化オプション付きのアセンブラ・ソースを**リスト35**に示します．

この例では変数iは必ずゼロです．最適化後は単純明快に1をレジスタにセットして戻しています．最適化前はそのまま冗長な処理，条件分岐を繰り返しています．

● -fcse-skip-blocks

上記の項目に似ています．絶対に到達しない条件分岐ブロックを無視し，コードを作成しません．

元のCソースを**リスト36**に，最適化オプションなし

リスト33　test06.c

```
#include <stdio.h>
int main()
{
    int res    =    0;
    int i =    0;
    if    (i    ==    0)
    {
         if    (i    ==    0    )
         {
              res   =    1;
         }
         else
         {
              res   =    0;
         }
    }
    else
    {
         res   =    0;
    }
    return     res;
}
```

リスト34　test06.s

```
    .file "test06.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $0,-4(%ebp)
    movl $0,-8(%ebp)
    cmpl $0,-8(%ebp)
    jne .L3
    cmpl $0,-8(%ebp)
    jne .L4
    movl $1,-4(%ebp)
    jmp .L5
    .p2align 4,,7
.L4:
    movl $0,-4(%ebp)
.L5:
    jmp .L6
    .p2align 4,,7
.L3:
    movl $0,-4(%ebp)
.L6:
    movl -4(%ebp),%edx
    movl %edx,%eax
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
         2.95.3 20010315 (release)"
```

リスト35　test06.s(-fcse-follow-jumps)

```
    .file "test06.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
         2.95.3 20010315 (release)"
```

リスト36　test07.c

```
#include <stdio.h>
int main()
{
    int res    =    0;
    int i      =    0;
    if    (i    !=    0)
    {
         res   =    1;
    }
    return     res;
}
```

リスト37　test07.s

```
    .file "test07.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $0,-4(%ebp)
    movl $0,-8(%ebp)
    cmpl $0,-8(%ebp)
    je .L3
    movl $1,-4(%ebp)
.L3:
    movl -4(%ebp),%edx
    movl %edx,%eax
    jmp .L2
    .p2align 4,,7
.L2:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
         2.95.3 20010315 (release)"
```

リスト38　test07.s(-fcse-skip-blocks)

```
    .file "test07.c"
    .version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    xorl %eax,%eax
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident     "GCC: (GNU)
         2.95.3 20010315 (release)"
```

のアセンブラ・ソースを**リスト37**に，最適化オプション付きのアセンブラ・ソースを**リスト38**に示します．

この例では変数iは必ずゼロなので最適化後は変数iがゼロでない条件分岐を削除し，レジスタに0をセットして戻しています．最適化前はそのまま冗長な処理，条件分岐を繰り返しています．

● -frerun-cse-after-loop

ループ最適化の実行後に共通部分式の削除をもう一度実行します．-fstrength-reduceで行ったことを徹底して行います．

● -frerun-loop-opt

ループ最適化を徹底して行います．これは-fstrength-reduceで行ったことです．ループ内で変化しない式やアドレス計算がチェックされます．そして，そうした計算はループの外に移され，その評価値がレジスタに格納されます．

● -fgcse

グローバル共通部分式を最後に削除するパスをコンパイラが実行します．元のCソースを**リスト39**に，最適化オプションなしのアセンブラ・ソースを**リスト**

リスト39　test08.c

```
#include <stdio.h>
voidtest();
int gData=8;
int i     =     1;
int main()
{
    int resTbl[100];
    test();
    gData =     8+i;
          for   (i=0;i<100;i++)
    {
          resTbl[i]  =     i*20;
    }
    gData =     8+i;
    test();
    return      0;
}
void      test()
{
    gData =     8+i;
}
```

40に，最適化オプション付きのアセンブラ・ソースを**リスト41**に示します．

この例では何度も繰り返される`gData = 8+i;`をひとまとめにセットしています．`8+i`の値も不変なのでまとめられます．

● -foptimize-register-moves，-fregmove

このオプションは，2オペランド命令の機種で役立ちます．インテル・アーキテクチャのマシンではあまり意味をもちません．

TRON仕様やRISC系マイクロプロセッサをもつマシンに指定してレジスタの割り当てを最適化します．TRON仕様のマシンの32ビット仕様のプロセッサに

リスト40　test08.s

```
    .file "test08.c"
    .version   "01.01"
gcc2_compiled.:
.globl gData
.data
    .align 4
    .type gData,@object
    .size gData,4
gData:
    .long 8
.globl i
    .align 4
    .type i,@object
    .size i,4
i:
    .long 1
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $420,%esp
    pushl %ebx
    call test
    movl i,%eax
    addl $8,%eax
    movl %eax,gData
    movl $0,i
    .p2align 4,,7
.L3:
    cmpl $99,i
    jle .L6
    jmp .L4
    .p2align 4,,7
.L6:
    movl i,%eax
    movl %eax,%edx
    leal 0(,%edx,4),%eax
    leal -400(%ebp),%edx
    movl i,%ebx
    movl %ebx,%ecx
    sall $2,%ecx
    addl %ebx,%ecx
    leal 0(,%ecx,4),%ebx
    movl %ebx,(%eax,%edx)
.L5:
    incl i
    jmp .L3
    .p2align 4,,7
.L4:
    movl i,%eax
    addl $8,%eax
    movl %eax,gData
    call test
    xorl %eax,%eax
    jmp .L2
    .p2align 4,,7
.L2:
    movl -424(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size main,.Lfe1-main
    .align 4
.globl test
    .type test,@function
test:
    pushl %ebp
    movl %esp,%ebp
    movl i,%eax
    addl $8,%eax
    movl %eax,gData
.L7:
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe2:
    .size test,.Lfe2-test
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

リスト41　test08.s(`-fgcse`)

```
    .file "test08.c"
    .version   "01.01"
gcc2_compiled.:
.globl gData
.data
    .align 4
    .type gData,@object
    .size gData,4
gData:
    .long 8
.globl i
    .align 4
    .type i,@object
    .size i,4
i:
    .long 1
.text
    .align 4
.globl test
    .type test,@function
test:
    pushl %ebp
    movl i,%eax
    movl %esp,%ebp
    addl $8,%eax
    movl %eax,gData
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe1:
    .size test,.Lfe1-test
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $420,%esp
    pushl %ebx
    call test
    movl i,%eax
    addl $8,%eax
    movl %eax,gData
    movl $0,i
    leal -400(%ebp),%ebx
    .p2align 4,,7
.L21:
    movl i,%ecx
    leal 0(,%ecx,4),%eax
    leal (%ecx,%eax),%edx
    sall $2,%edx
    movl %edx,(%eax,%ebx)
    leal 1(%ecx),%eax
    movl %eax,i
    cmpl $99,%eax
    jle .L21
    addl $9,%ecx
    movl %ecx,gData
    call test
    xorl %eax,%eax
    popl %ebx
    movl %ebp,%esp
    popl %ebp
    ret
.Lfe2:
    .size main,.Lfe2-main
    .ident     "GCC: (GNU)
             2.95.3 20010315 (release)"
```

は32ビット汎用レジスタが16あるので，効率良く変数をレジスタに割り当てられます．

● -ffunction-sections，-fdata-sections

このオプションもまた，インテル・アーキテクチャのマシンではあまり意味をもちません．

リンカに命令空間の参照の局所性を改善する最適化を行う機能があるシステムでは，これらのオプションを使うと良いでしょう．HP-UXの稼働するHP PAプロセッサとSolaris2の稼働するSPARCプロセッサには，このような最適化機能のあるリンカが存在します．このオプションを使うとデバッグ時に問題が生じる可能性があるので，デバッグ中には使用しないほうが良いでしょう．しかし，付けたときと付けないときで実行形式ファイルが変わるため，試験を慎重に行わないとバグの元になるおそれがあります．

● -fcaller-saves

関数呼び出しにより破壊されるレジスタに値を割り当てることを可能にしますが．その結果として，命令を余分に生成し，呼び出しの前後でレジスタのセーブとリストアを行うので汎用レジスタがふんだんにあるマシンでないと意味がないように思います．

● -funroll-loops

ループ展開最適化を実行します．これは，コンパイル時か実行時に繰り返し回数が決められるループにしか行われません．`-funroll-loops`は，前述の`-fstrength-reduce`と`-frerun-cse-after-loop`を含みます．

● -funroll-all-loops

ループ展開最適化を実行します．これは，すべてのループに対して行われ，普通はプログラムの実行を遅くしてしまいます．

● -fmove-all-movables

ループ中の不変な計算をすべてループの外に移動します．

● -freduce-all-givs

ループ中の一般誘導変数を削減します．

最適化というものは両刃の剣であり，あちらを立てればこちらが立たずという状態にもなります．せっかく速度を追求してもまちがった最適化を行うと，サイズは小さくなったものの速度が遅くなってしまうようなこともよくあります．