8.10 M680x0 Dependent Features

8.10.1 M680x0 Options

Motorola 680x0 バージョンの as には、いくつかの機種依存オプションがあります。

`-l`オプション

`-l` オプションを使用すると、未定義シンボルへの参照のサイズを短くすることができます。 `-l` オプションを使用しない場合、未定義シンボルへの参照は完全な長さ(32 ビット)に相当する幅を持ちます。(これらのシンボルがどこに行き着くかはasには分からないので、asにできることははリンカが後で埋めるためのスペースを確保することだけです。asはこれらのシンボルがどの程度離れているか分からないので、できる限り多くのスペースを確保するしかない)。このオプションを使用すると、参照は1ワード幅(16ビット)にしかなりません。これは、オブジェクトファイルをできるだけ小さくしたい場合で、関連するシンボルが常に17ビット未満にあることを知っている場合に便利です。

`--register-prefix-optional`オプション

いくつかの構成、特にコンパイラが通常ユーザー変数名の前にアンダースコアを付けない構成では、アセンブラはレジスタ名の前に `%` をつけることを要求します。これは、アセンブラが `a0` から `a7` などの名前のC言語変数や関数を区別できるようにするためのものです。この `%` は常に受け付けられますが、設定によっては(特に`sun3`では)必要ありません。`--register-prefix-optional`オプションを使用すると、通常は `%` が必要な設定であっても、それを省略することができます。 この場合、一般的にレジスタ名と同じ名前のC変数や関数を参照することは不可能になります。

`--bitwise-or`オプション

通常、文字 `|` はコメント文字として扱われ、式の中で使用することはできません。`--bitwise-or`オプションは、`|`を通常の文字に変えます。 このモードでは、C言語スタイルのコメントを使用するか、行頭の `#` 文字でコメントを開始する必要があります。

`--base-size-default-16 --disp-size-default-32` オプション

サイズを指定せずにベースレジスタでアドレッシングモードを使用した場合、通常、丸ごと32ビットの値が使用されます。例えば，アドレス指定モード`%a0@(%d0)`は，`%a0@(%d0:l)`と同等です．`--base-size-default-16`オプションを使用すると、デフォルトで16ビットの値を使用するように指示することができます。この場合、`%a0@(%d0)`

は`%a0@(%d0:w)`と同等です。デフォルトの動作に戻すには `--base-size-default-32` オプションを使用します。

`--disp-size-default-16` `--disp-size-default-32` オプション

変位を伴うアドレッシングモードを使用し、変位の値がわからない場合、asは通常、その値が32ビットであると仮定します。例えば、シンボル `disp` が定義されていない場合、asはアドレス指定モード `%a0@(disp,%d0)` を `disp` が32ビット値であるかのようにアセンブルします。`--disp-size-default-16`オプションを使用すると、asに変位が16ビットであると仮定するように指示することができます。この場合、asは`disp`が16ビット値であるかのように`%a0@(disp,%d0)`をアセンブルします。デフォルトの挙動に戻すために `--disp-size-default-32` オプションを使用することができます。

CPU種類指定オプション

`as` は Motorola 680x0 ファミリのいくつかの異なるメンバ用のコードをアセンブルすることができます。デフォルトは、as がビルドされたときにどのように設定されたかに依存します。通常、デフォルトでは 68020 マイクロプロセッサのコードをアセンブルします。次のオプションは、デフォルトを変更するために使用することができます。これらのオプションは、どの命令とアドレス指定モードが許可されるかを制御します。 680x0ファミリのメンバは、非常によく似ています。違いについての詳しい情報は、モトローラのマニュアルを参照してください。

`-m68000 -m68ec000 -m68hc000 -m68hc001 -m68008 -m68302 -m68306 -m68307 -m68322 -m68356`

68000用にアセンブルする。ここで上げたチップはアセンブラから見た場合区別がないので、`-m68008`、`-m68302`などは`-m68000`と同義語です。

`-m68010`

68010用にアセンブルします。

`-m68020 -m68ec020`

68020 用にアセンブルします。通常、これがデフォルトです。

`-m68030 -m68ec030`

68030 用にアセンブルします。

`-m68040 -m68ec040`

68040 用にアセンブルします。

`-m68060 -m68ec060`

68060 用にアセンブルします。

`-mcpu32 -m68330 -m68331 -m68332 -m68333 -m68334 -m68336 -m68340 -m68341 -m68349 -m68360`

CPU32ファミリのチップに対応したアセンブルを行います。

`-m5200`

ColdFireファミリのチップ用にアセンブルします。

`-m68881 -m68882`

68881 浮動小数点演算命令をアセンブルします。これは、68020、68030、および CPU32 のデフォルトです。68040と68060は常に浮動小数点命令をサポートしています。

`-mno-68881`

68881浮動小数点命令をアセンブルしない。これは68000と68010のデフォルトです。68040と68060は、このオプションが使用されていても、常に浮動小数点命令をサポートしています。

`-m68851`

68851 MMU 命令をアセンブルします。これは 68020、68030、および 68060 のデフォルトです。68040 では、MMU 命令のセットが多少異なるため、`-m68851` と `-m68040` を一緒に使用するべきではありません。

`-mno-68851`

68851 MMU 命令をアセンブルしません。これは68000、68010、CPU32のデフォルトです。68040では、多少異なるMMU命令のセットを受け付けます。

8.10.2 Syntax

モトローラ680x0用のこの構文は、mitで開発されました。

680x0版のasは、Sunアセンブラと互換性のある命令名と構文を使用しています。ピリオドは無視されます。例えば`movl`は`mov.l`と等価です。

以下の表では、apc はアドレスレジスタ (%a0 から %a7), プログラムカウンタ (%pc), プログラムカウンタに対するゼロアドレス (%zpc), 抑止(suppressed)アドレスレジスタ (%za0 から %za7) のいずれかを表しており、完全に省略することもできます。サイズは w または l のいずれかを意味し、スケールも指定されない限り先頭のコロンとともに省略することができます。スケールは，1，2，4，8のいずれかを意味し，常に先頭のコロンと一緒に省略することができます。

次のアドレス指定モードが理解できる：

イミディエイト	#number	
データレジスタ	%d0 through %d7	
アドレスレジスタ	%a0 through %a7	a7の別名%sp (スタックポインタ) a6の別名%fp (フレームポインタ)
アドレスレジスタ間接	%a0@ through %a7@	
アドレスレジスタポストインクリメント	%a0@+ through %a7@+	
アドレスレジスタプレデクリメント	%a0@- through %a7@-	
間接プラスオフセット	apc@(number)	
インデクス	apc@(number, register:size:scale)	numberは省略可能です。
ポストインデクス	apc@(number)@(onumber, register:size:scale)	onumberまたはregisterのいずれか一方は省略可能です(両方同時の省略はできません)。

プレインデクス	`apc@(number, register:size:scale)` `@(onumber)`	numberは省略可能です。registerを省略するとポストインデクスモードになってしまいます。
絶対	`symbol` または `digits`	この後ろに`:b`, `:w`, `:l`をつけることができます。

8.10.3 Motorola Syntax

このチップの標準的なモトローラ構文は、すでに説明した構文とは異なります(8.10.2 Syntaxを参照)。 `as`は、同じ命令内の他のオペランドにmit構文が使用されていても、オペランドにモトローラ構文を受け入れることができます。この2種類の構文には完全な互換性があります。

以下の表では、apc はアドレスレジスタ(`%a0` から `%a7`)、プログラムカウンタ(`%pc`)、プログラムカウンタに対するゼロアドレス(`%zpc`)、抑制(suppressed)アドレスレジスタ(`%za0` から `%za7`)のいずれかを表しています。サイズは、`w`または`l`のいずれかを意味し、常に先頭のドットとともに省略することができます。スケールは、`1`、`2`、`4`、`8`のいずれかを意味し、常に先頭のアスタリスクとともに省略することができます。

以下の追加アドレス指定モードが理解できる：

アドレスレジスタ間接	`(%a0)` ~ `(%a7)`	`a7`の別名`%sp` (スタックポインタ) `a6`の別名`%fp` (フレームポインタ)
アドレスレジスタポストインクリメント	`(%a0)+` ~ `(%a7)+`	
アドレスレジスタプレデクリメント	`-(%a0)` ~ `-(%a7)`	

インデクスプラスオフセット	`number(%a0)` ~ `number(%a7)`, or `number(%pc)`.	また，`(number,%a0)`のように，括弧の中にnumberが入ることもある。 pcと併用する場合は，数字を省略することができる(アドレスレジスタの場合は， 数字を省略するとアドレスレジスタ間接モードになる)．
インデクス	`number(apc, register.size *scale)`	numberは省略してもよいし、括弧の中に 入れてもよい。apcは省略することができる。 registerとapcはどちらの順序で現れてもよい。 apc と register の両方がアドレスレジスタで、size と scale が省略された場合、最初のレジスタがベースレジスタ、2 番目のレジスタがインデックスレジスタとして扱われます。
ポストインデクス	`([number,apc], register.size *scale,onumber)`	onumberまたはregisterを省略することができます(両方省略も可能です)。 numberかapcいずれか一方を省略することができます(両方同時に 省略はできません)。
プレインデクス	`([number,apc, register.size *scale],onumber)`	number, apc, registerのうちの任意の2つを省略することができます。 onumberは省略することができます。registerとapcは、どちらの順序で現れても構いません。 apcとregisterの両方がアドレスレジスタで、サイズとスケールが省略された場合、 最初のレジスタがベースレジスタとして、2番目のレジスタがインデックスレジスタとして扱われます。

8.10.4 Floating Point

パックド・デシマル(P)形式の浮動小数点数リテラルには対応していません。対応させるためのコード追加はご自由にどうぞ。ディレクティブで生成される浮動小数点フォーマットは以下の通りです。

`.float` 単精度浮動小数点数の定数

`.double` 倍精度浮動小数点数の定数

`.extend .ldouble` 拡張精度(long double)浮動小数点数の定数

8.10.5 680x0 Machine Directives

Sunアセンブラと互換性を持たせるために、680x0アセンブラは以下のディレクティブを理解します。

`.data1` This directive is identical to a `.data 1` directive.

`.data2` This directive is identical to a `.data 2` directive.

`.even` This directive is a special case of the `.align` directive; it aligns the output to an even byte boundary.

`.skip` This directive is identical to a `.space` directive.

8.10.6 Opcodes

8.10.6.1 Branch Improvement

分岐命令には、ある種の疑似オペコードが認められています。これらは、ターゲットに到達する最短の分岐命令に展開されます。一般にこれらのニーモニックは、モトローラのニーモニックの先頭にある`j`を`b`に置き換えることで作られます。

次の表は、擬似演算をまとめたものです。フラグ`*`は，表の後にさらに詳しく説明されるケースを示す:

```
           Displacement
           +-----------------------------------------------
           |          68020 68000/10
Pseudo-Op |BYTE WORD LONG  LONG      non-PC relative
           +-----------------------------------------------
     jbsr |bsrs bsr  bsrl  jsr       jsr
      jra |bras bra  bral  jmp       jmp
*     jXX |bXXs bXX  bXXl  bNXs;jmpl bNXs;jmp
*    dbXX |dbXX dbXX dbXX; bra; jmpl
*    fjXX |fbXXw fbXXw fbXXl         fbNXw;jmp

XX: condition
NX: negative of condition XX
*—see full description below
```

`jbsr`: `jra` これらは最も単純なジャンプ擬似演算であり、分岐先への変位の大きさに応じて、常に特定のマシン命令にマッピングされます。

`jXX`: ここで、`jXX`は擬似演算のファミリ全体を表し、XXは条件分岐または条件コードテストである。このファミリの擬似演算の全リストは以下の通りです：

```
jhi jls jcc jcs jne jeq jvc
jvs jpl jmi jge jlt jgt jle
```

非PC相対変位と68000または68010の長変位の場合、NXによってより長いコードになってしまうため、XXとは逆の条件となる。例えば、非PC相対の場合：

```
    jXX foo
```

は、

```
    bNXs oof
    jmp foo
  oof:
```

を生成する。

`dbXX`: このファミリの議事演算のすべては以下の通りです。

```
dbhi dbls dbcc dbcs dbne dbeq dbvc
dbvs dbpl dbmi dbge dblt dbgt dble
dbf dbra dbt
```

ワード変位とバイト変位以外は、ソースが`dbXX foo`を読むと、次のようにコードを生成します。

```
    dbXX oo1
    bra oo2
  oo1:jmpl foo
  oo2:
```

`fjXX`: このファミリには以下が含まれます。

```
fjne fjeq fjge fjlt fjgt fjle fjf
fjt fjgl fjgle fjnge fjngl fjngle fjngt
fjnle fjnlt fjoge fjogl fjogt fjole fjolt
fjor fjseq fjsf fjsne fjst fjueq fjuge
fjugt fjule fjult fjun
```

PC相対でないブランチターゲットに対しては、`fjXX foo`に対して以下のようにコードが生成されます。

```
    fbNX oof
    jmp foo
  oof:
```

8.10.6.2 Special Characters

イミディエイト文字はSunとの互換性のために `#` です。行コメント文字は `|` です (`--bitwise-or` オプションが使用されていない限り)。行頭に `#` が現れると、それが `# line file` のように見えない限りコメントとして扱われ、その場合は普通に扱われます。