This file documents the GNU Assembler "as".

Copyright (C) 1991, 92, 93, 94, 95, 1996 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

AT&T Syntax versus Intel Syntax

In order to maintain compatibility with the output of ‘gcc’, ‘as’ supports AT&T System V/386 assembler syntax. This is quite different from Intel syntax. We mention these differences because almost all 80386 documents used only Intel syntax. Notable differences between the two syntaxes are:

* AT&T immediate operands are preceded by ‘$’; Intel immediate operands are undelimited (Intel ‘push 4’ is AT&T ‘pushl $4’). AT&T register operands are preceded by ‘%’; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by ‘*’; they are undelimited in Intel syntax.

* AT&T and Intel syntax use the opposite order for source and destination operands. Intel ‘add eax, 4’ is ‘addl $4, %eax’. The ‘source, dest’ convention is maintained for compatibility with previous Unix assemblers.

* In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of ‘b’, ‘w’, and ‘l’ specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixes memory operands (*not* the opcodes themselves) with ‘byte ptr’, ‘word ptr’, and ‘dword ptr’. Thus, Intel ‘mov al, byte ptr FOO’ is ‘movb FOO, %al’ in AT&T syntax.

* Immediate form long jumps and calls are ‘lcall/ljmp $SECTION, $OFFSET’ in AT&T syntax; the Intel syntax is ‘call/jmp far SECTION:OFFSET’. Also, the far return instruction is ‘lret $STACK-ADJUST’ in AT&T syntax; Intel syntax is ‘ret far STACK-ADJUST’.

* The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

Opcode Naming

Opcode names are suffixed with one character modifiers which specify the size of operands. The letters ‘b’, ‘w’, and ‘l’ specify byte, word, and long operands. If no suffix is specified by an instruction and it contains no memory operands then ‘as’ tries to fill in the

missing suffix based on the destination register operand (the last one by convention). Thus, ‘mov %ax, %bx’ is equivalent to ‘movw %ax, %bx’; also, ‘mov $1, %bx’ is equivalent to ‘movw $1, %bx’. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing opcode suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the opcode suffix.) Intel’s ‘lea’ for ‘load effective address’ is similarly suffixed.

Almost all opcodes have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend *from* and a size to zero extend *to*. This is accomplished by using two opcode suffixes in AT&T syntax. Base names for sign extend and zero extend are ‘movs...’ and ‘movz...’ in AT&T syntax (‘movsx’ and ‘movzx’ in Intel syntax). The opcode suffixes are tacked on to this base name, the *from* suffix before the *to* suffix. Thus, ‘movsbl %al, %edx’ is AT&T syntax for "move sign extend *from* %al *to* %edx." Possible suffixes, thus, are ‘bl’ (from byte to long), ‘bw’ (from byte to word), and ‘wl’ (from word to long).

The Intel-syntax conversion instructions

* ‘cbw’ -- sign-extend byte in ‘%al’ to word in ‘%ax’,

* ‘cwde’ -- sign-extend word in ‘%ax’ to long in ‘%eax’,

* ‘cwd’ -- sign-extend word in ‘%ax’ to long in ‘%dx:%ax’,

* ‘cdq’ -- sign-extend dword in ‘%eax’ to quad in ‘%edx:%eax’,

are called ‘cbtw’, ‘cwtl’, ‘cwtd’, and ‘cltd’ in AT&T naming. ‘as’ accepts either naming for these instructions.

Far call/jump instructions are ‘lcall’ and ‘ljmp’ in AT&T syntax, but are ‘call far’ and ‘jump far’ in Intel convention.

Register Naming

Register operands are always prefixes with ‘%’. The 80386 registers consist of

* the 8 32-bit registers ‘%eax’ (the accumulator), ‘%ebx’, ‘%ecx’, ‘%edx’, ‘%edi’, ‘%esi’, ‘%ebp’ (the frame pointer), and ‘%esp’ (the stack pointer).

* the 8 16-bit low-ends of these: ‘%ax’, ‘%bx’, ‘%cx’, ‘%dx’, ‘%di’, ‘%si’, ‘%bp’, and ‘%sp’.

* the 8 8-bit registers: ‘%ah’, ‘%al’, ‘%bh’, ‘%bl’, ‘%ch’, ‘%cl’, ‘%dh’, and ‘%dl’ (These are the high-bytes and low-bytes of ‘%ax’, ‘%bx’, ‘%cx’, and ‘%dx’)

* the 6 section registers ‘%cs’ (code section), ‘%ds’ (data section), ‘%ss’ (stack section), ‘%es’, ‘%fs’, and ‘%gs’.

* the 3 processor control registers ‘%cr0’, ‘%cr2’, and ‘%cr3’.

* the 6 debug registers ‘%db0’, ‘%db1’, ‘%db2’, ‘%db3’, ‘%db6’, and ‘%db7’.

* the 2 test registers ‘%tr6’ and ‘%tr7’.

* the 8 floating point register stack ‘%st’ or equivalently ‘%st(0)’, ‘%st(1)’, ‘%st(2)’, ‘%st(3)’, ‘%st(4)’, ‘%st(5)’, ‘%st(6)’, and ‘%st(7)’.

Opcode Prefixes

Opcode prefixes are used to modify the following opcode. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to give operand and address size (16-bit operands are specified in an instruction by prefixing what would normally be 32-bit operands with a "operand size" opcode prefix). Opcode prefixes are usually given as single-line instructions with no operands, and must directly precede the instruction they act upon. For example, the ‘scas’ (scan string) instruction is repeated with:

```
repne
scas
```

Here is a list of opcode prefixes:

* Section override prefixes ‘cs’, ‘ds’, ‘ss’, ‘es’, ‘fs’, ‘gs’. These are automatically added by specifying using the SECTION:MEMORY-OPERAND form for memory references.

* Operand/Address size prefixes ‘data16’ and ‘addr16’ change 32-bit operands/addresses into 16-bit operands/addresses. Note that 16-bit addressing modes (i.e. 8086 and 80286 addressing modes) are not supported (yet).

* The bus lock prefix ‘lock’ inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).

* The wait for coprocessor prefix ‘wait’ waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.

* The ‘rep’, ‘repe’, and ‘repne’ prefixes are added to string instructions to make them repeat ‘%ecx’ times.

Memory References

An Intel syntax indirect memory reference of the form

```
SECTION:[BASE + INDEX*SCALE + DISP]
```

is translated into the AT&T syntax

```
SECTION:DISP(BASE, INDEX, SCALE)
```

where BASE and INDEX are the optional 32-bit base and index registers, DISP is the optional displacement, and SCALE, taking the values 1, 2, 4, and 8, multiplies INDEX to calculate the address of the operand. If no SCALE is specified, SCALE is taken to be 1. SECTION specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in AT&T syntax *must* have be preceded by a ‘%’. If you specify a section override which coincides with the default section register, ‘as’ does *not* output any section register override prefixes to assemble the given instruction. Thus,

section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

AT&T: '-4(%ebp)', Intel: '[ebp - 4]'
BASE is '%ebp'; DISP is '-4'. SECTION is missing, and the default section is used ('%ss' for addressing with '%ebp' as the base register). INDEX, SCALE are both missing.

AT&T: 'foo(,%eax,4)', Intel: '[foo + eax*4]'
INDEX is '%eax' (scaled by a SCALE 4); DISP is 'foo'. All other fields are missing. The section register here defaults to '%ds'.

AT&T: 'foo(,1)'; Intel '[foo]'
This uses the value pointed to by 'foo' as a memory operand. Note that BASE and INDEX are both missing, but there is only *one* ','. This is a syntactic exception.

AT&T: '%gs:foo'; Intel 'gs:foo'
This selects the contents of the variable 'foo' with section register SECTION being '%gs'.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with '*'. If no '*' is specified, 'as' always chooses PC relative addressing for jump/call labels.

Any instruction that has a memory operand *must* specify its size (byte, word, or long) with an opcode suffix ('b', 'w', or 'l', respectively).

Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e. prefixing the jump instruction with the 'addr16' opcode prefix), since the 80386 insists upon masking '%eip' to 16 bits after the word displacement is added.

Note that the 'jcxz', 'jecxz', 'loop', 'loopz', 'loope', 'loopnz' and 'loopne' instructions only come in byte displacements, so that if you use these instructions ('gcc' does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding 'jcxz foo' to

```
        jcxz cx_zero
        jmp cx_nonzero
cx_zero: jmp foo
cx_nonzero:
```

Floating Point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an opcode suffix and a constructor associated with it. Opcode suffixes specify operand's data types. Constructors build these data types into memory.

* Floating point constructors are ‘.float’ or ‘.single’, ‘.double’, and ‘.tfloat’ for 32-, 64-, and 80-bit formats. These correspond to opcode suffixes ‘s’, ‘l’, and ‘t’. ‘t’ stands for temporary real, and that the 80387 only supports this format via the ‘fldt’ (load temporary real to stack top) and ‘fstpt’ (store temporary real and pop stack) instructions.

* Integer constructors are ‘.word’, ‘.long’ or ‘.int’, and ‘.quad’ for the 16-, 32-, and 64-bit integer formats. The corresponding opcode suffixes are ‘s’ (single), ‘l’ (long), and ‘q’ (quad). As with the temporary real format the 64-bit ‘q’ format is only present in the ‘fildq’ (load quad integer to stack top) and ‘fistpq’ (store quad integer and pop stack) instructions.

Register to register operations do not require opcode suffixes, so that ‘fst %st, %st(1)’ is equivalent to ‘fstl %st, %st(1)’.

Since the 80387 automatically synchronizes with the 80386 ‘fwait’ instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, ‘as’ suppresses the ‘fwait’ instruction whenever it is implicitly selected by one of the ‘fn...’ instructions. For example, ‘fsave’ and ‘fnsave’ are treated identically. In general, all the ‘fn...’ instructions are made equivalent to ‘f...’ instructions. If ‘fwait’ is desired it must be explicitly coded.

Writing 16-bit Code

While GAS normally writes only "pure" 32-bit i386 code, it has limited support for writing code to run in real mode or in 16-bit protected mode code segments. To do this, insert a ‘.code16’ directive before the assembly language instructions to be run in 16-bit mode. You can switch GAS back to writing normal 32-bit code with the ‘.code32’ directive.

GAS understands exactly the same assembly language syntax in 16-bit mode as in 32-bit mode. The function of any given instruction is exactly the same regardless of mode, as long as the resulting object code is executed in the mode for which GAS wrote it. So, for example, the ‘ret’ mnemonic produces a 32-bit return instruction regardless of whether it is to be run in 16-bit or 32-bit mode. (If GAS is in 16-bit mode, it will add an operand size prefix to the instruction to force it to be a 32-bit return.)

This means, for one thing, that you can use GNU CC to write code to be run in real mode or 16-bit protected mode. Just insert the statement ‘asm(".code16");’ at the beginning of your C source file, and while GNU CC will still be generating 32-bit code, GAS will automatically add all the necessary size prefixes to make that code run in 16-bit mode. Of course, since GNU CC only writes small-model code (it doesn’t know how to attach segment selectors to pointers like native x86 compilers do), any 16-bit code you write with GNU CC will essentially be limited to a 64K address space. Also, there will be a code size and performance penalty due to all the extra address and operand size prefixes GAS has to add to the instructions.

Note that placing GAS in 16-bit mode does not mean that the resulting code will necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you would have to refrain from using *any* 32-bit constructs which require GAS to output address or operand size prefixes. At the moment this would be rather difficult, because GAS currently supports *only* 32-bit addressing modes: when writing 16-bit code, it *always* outputs address size prefixes for any

instruction that uses a non-register addressing mode. So you can write code that runs on 16-bit processors, but only if that code never references memory.

Notes

There is some trickery concerning the ‘mul’ and ‘imul’ instructions that deserves mention. The 16-, 32-, and 64-bit expanding multiplies (base opcode ‘0xf6’; extension 4 for ‘mul’ and 5 for ‘imul’) can be output only in the one operand form. Thus, ‘imul %ebx, %eax’ does *not* select the expanding multiply; the expanding multiply would clobber the ‘%edx’ register, and this would confuse ‘gcc’ output. Use ‘imul %ebx’ to get the 64-bit product in ‘%edx:%eax’.

We have added a two operand form of ‘imul’ when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying ‘%eax’ by 69, for example, can be done with ‘imul $69, %eax’ rather than ‘imul $69, %eax, %eax’.