1. Purpose_of_this_Document

This document provides the third draft of the assembler language definition for the 5.3/386 CCS. The goal of this effort is to take the current 286 assembler and upgrade it to a 386 assembler in the minimum possible time. This document describes the resulting product.

1.1 INVOKING_THE_ASSEMBLER

The assembler is invoked by the command:
as [-o outfile] [-n] [-R] [-v] [-u] [-x] infile

The flags have the following meaning:

-o filename
Use filename as the output file. The output file name is generated by the algorithm at the end of this section.

-n
No address optimization.

-R
Remove (unlink) the input file after assembly is completed.

-V
Write the version number of the assembler on the standard error output. This option does allow for normale assembly.

-u
Remove unreferenced debugging symbols from the symbol table.

-x
Extended addressing (48-bit pointers) will be used.

The input assembly language program is read from infile and the output object module is written to outfile. The assembler only accepts one infile on a command line. If outfile is not specified, the name is created from infile by the following algorithm:

+ If the name infile ends with the two characters .s, the name outfile is created by replacing these last two characters with .o.

+ If the name infile does not end with the two characters .s and is no more than 12 characters long, the name outfile is created by appending .o to the name infile.

+ If the name infile does not end with the two characters .s and is greater than 12 characters long, the name outfile is created by appending .o to the first 12 characters of infile. This satisfies the UNIX system requirement that a file name be no more than 14 characters long.

1.2 INPUT_FORMAT

The input to the assembler is a text file. This file must consist of a sequence of lines ending with a newline character (ASCII LF). Each line can contain one or more statements. If several statements appear on a line, they must be separated by semicolons(;). Each statement must be one of the following:

+ An empty statement is one that contains nothing other than spaces, tabs, and form-feed characters. Empty statements have no meaning to the assembler. They can be inserted freely to improve the appearance of a listing.

+ An assignment statement is one that gives a value to a symbol. It consists of a symbol, followed by an equal sign(=), followed by an expression. The expression is evaluated and the result is assigned to the symbol. Assignment statements do not generate any code. They are used only to assign assembly time values to symbols.

+ A pseudo operation statement is a directive to the assembler that does not necessarily generate any code. It consists of a pseudo operation code, followed by zero or more operands. Every pseudo operation code begins with a period(.).

+ A machine operation statement is a mnemonic representation of an executable machine instruction that is translated by the assembler. It consists of an operation code, followed by zero or more operands.

In addition, each statement can be modified by one or more of the following:

+ A label can be placed at the begining of any statement. This consists of a symbol followed by a colon(:). When a label is encountered by the assembler, the value of the location counter is assigned to the label.

+ A comment can be inserted at the end of any statement by preceding the comment with a slash(/). The slash causes the assembler to ignore any characters in the line after the slash. This facility is provided to allow insertion of internal program documentation into the source file for a program.

1.3 OUTPUT_FORMAT

The output of the assembler is an objectfile. The object file produced by the assembler contains at least the following three sections:

.text This is an initialized section, normally it is read only and contains the code from a program. It may also contain read only tables.

.data This is an initialized section, normally it is readable abd writable. It contains initialized data. These can be scalers or tables.

.bss This is an uninitialized section. Space is not

allocated for this segment in the coff file.

An optional section, .comment may also be produced (See the section "Pseudo Ops").

Every statement in the input assembly language program that generates code or data generates it into one of these three sections. The section into which the generated bytes are to be written starts out as .text, and can be switched using section control pseudo operations.

The assembler can produce object modules with any one of four (4) different magic numbers. Each magic number indicates that a different (incompatible) function linkage has been used.

The -x option must be specified to get an object file with 48-bit pointers. The default object file type (with no -x option) is a 32-bit pointer object file.

The -x option does not change the output code - the handling of 48-bit addresses must be done in the assembly code, byt the programer. The -x option tells the assembler what type of magic number to put into the coff file.

SYMBOLS AND EXPRESSIONS

2. SYMBOLS_and_EXPRESSIONS

2.1 Values

Values are represented in the assembler by 32 bit 2's compliment values. All arithmetic is performed using 32 bits of precision. Note that the values used in a 386 instruction may use 8, 16, or 32 bits.

2.1.1 Types Every value is an instance one of the following types:

Undefined
: An undefined symbol is one whose value has not yet been defined. Examples of undefined symbols are forward references and externals.

Absolute
: An absolute type is one whose value does not change with relocation. Examples of absolute symbols are numeric constants and expressions whose operands are only numeric constants.

Text
: A text type symbol is one whose value is relative to the text segment.

Data
: A data type symbol is one whose value is relative to the data segment.

Bss

A bss type symbol is one whose value is relative to the bss segment.

Any of the above symbol types can be given the attribute EXTERNAL.

2.2 Symbols

A symbol has a value and a type each of which is either specified explicitly by an assignment statement or from it's context. Refer to section 2.3 (Expressions) for the regular expression definition of a symbol.

2.2.1 Reserved_Symbols The following symbols are reserved by the assembler.

. Commonly refered to as dot. This is the location counter while assembling a program. It takes on the current location in the text, data, or bss section.

.text
This symbol is of type text. It is used to label the beginning of a text section in the program being assembled.

.data
This symbol is of type data. It is used to label the beginning of a data section in the program being assembled.

.bss
This symbol is of type bss. It is used to label the beginning of a bss section in the program being assembled.

2.3 Expressions

2.3.1 General The expressions accepted by the UNIX 386 assembler can be described by their semantic and syntactic rules.

The following are the operators supported by the assembler:

OPERATOR	ACTION
+	addition
-	subtraction
*	multiplication
\/	division
&	bit wise logical and
|	bit wise logical or
>	right shift
<	left shift
\%	remainder operator
!	bit wise logical and not

In the following syntactic rules the non-terminals are represented by lower case letters. The terminal symbols are represented by upper case letters and the symbols enclosed in double quotes ("") are terminal symbols.

There is no precedence to the operators. Square brackets

must be used to establish precedence.

SYNTACTIC RULES FOR THE ASSEMBLER

```
expr    : term
        | expr "+" term
        | expr "   term
        | expr "/" term
        | expr "&" term
        | expr "|" term
        | expr ">" term
        | expr "<" term
        | expr "" term
        | expr "!" term
        | expr "-" term
        ;


term    : id
        | number
        | "-" term
        | "[" expr "]"
        | "<o>" term
        | "<s>" term
        ;

id      : LABEL
        ;

number  : DEC_VAL
        | HEX_VAL
        | OCT_VAL
        | BIN_VAL
        ;
```

The Terminal nodes can be described by the following regular expressions.

```
LABEL   = [a-zA-Z_][a-zA-Z0-9_]*:
DEC_VAL = [1-9][0-9]*
HEX_VAL = 0[Xx][0-9a-fA-F][0-9a-fA-F]*
OCT_VAL = 0[0-7]*
BIN_VAL = 0[Bb][0-1][0-1]*
```

In the above regular expressions choices are enclosed in square brackets, a range of letters or numbers are separated by a dash (-), and the star (*) indicates zero (0) or more instances of the previous character.

Semantically the expressions fall into two groups, they are absolute and relocatable. The following table shows the legal combinations of absolute and relocatable operands, for the addition and subtraction operators. All other operations are only legal on absolute valued expressions.

All numbers have the absolute attribute. Symbols used to reference storage, text or data, are relocatable. In an assignment statement Symbols on the left hand side inherit their relocation attributes from the right hand side.

In the table "a" is an absolute valued expression, and "r" is a relocatable valued expression. The resulting type of the operation is given to the right of the equal sign.

```
a + a = a
r + a = r

a - a = a
r - a = r
r - r = a
```

In the last example, the relocatable expressions must be declared before their difference can be taken.

Following are some examples of valid expressions:

1. label"

2. $label"

3. [label + 0x100]"

4. [label1 - label2]"

5. $[label1 - label2]"

Following are some examples of invalid expressions:

1. [$label - $label]"

2. [label1 * 5]"

3. (label + 0x20)"

PSEUDO OPERATIONS

.align val — The align pseudo op causes the next data generated to be aligned modulo val. Val must be an positive integer value.

.bcd val — The bcd pseudo op generates a packed decimal (80-bit) value into the current section. This is not valid for the .bss section. Val is a non-floating point constant.

.bss — The bss pseudo op changes the current section to .bss

.bss tag, bytes — Define symbol tag in the .bss section and add bytes to the value of dot for .bss. This does not change the current section to .bss. Tag is a symbol name. Bytes must be an positive integer value.

.byte val [,val] — The byte pseudo op generates initialized bytes into the current section. This is not valid for .bss. Each val must be an 8-bit value.

.comm name, expr The comm pseudo op allocates storage in the .data section. The storage is referenced by name, and has a size in bytes of expr. Name is a symbol. Expr must be an positive integer. The name can not be pre-defined.

.data The data pseudo op changes the current section to .data.

.double val The double pseudo op generates an 80287 long real (64-bit) into the current section. Not valid the .bss section. Val is a floating point constant.

.even The even pseudo op aligns the current program counter, (.) to an even boundary.

.float val The float pseudo op generates a 80287 short real (32 bit) into the current section. This is not valid in the .bss section. Val is a floating point constant.

.globl name This pseudo op makes the variable, name, accessible to other programs.

.ident string The ident pseudo op creates an entry in the comment section containing string. String is any sequence of characters, not including the double quote, '"'.

.lcomm name, expr The lcomm pseudo op allocates storage in the .bss section. The storage is referenced by name, and has a size of expr. Name is a symbol. Expr must be of type positive integer. Name can not be pre-defined.

.long val The long pseudo op generates a long integer (32-bit two's complement value) into the current section. This pseudo op is not valid for the .bss section Val is a non-floating point constant.

.noopt The noopt pseudo op

.optim The optim pseudo op

.set name, expr The set pseudo op sets the value of symbol name to expr. This is equivalent to an assignment.

.string str This pseudo places the characters in str into the object module at the current loc and terminates the string with a null. The string must be enclosed in double quotes (""). This pseudo op is not valid for the .bss section.

.text The text pseudo op defines the current section as .text.

.value expr [,expr] The value pseudo op is used to generate an initialized word (16-bit two's complement value) into the current section. This pseudo op is not valid in the .bss section. Each expr must be a 16-bit value.

.version string The version pseudo op puts the C compiler version into the comment section.

SDB PSEUDO OPS

.type expr The type pseudo op is used with in a .def-.endef pair. It gives the name the C compiler type representation expr.

.val expr The val pseudo op is used with a .def-.endef pair. It gives name the value of expression. The type of expr determines the section for name.

.tag str The tag pseudo op is used in relation with a previously defined .def pseudo op. If the name of a .def is a structure or a union, str should be the name of that structure or union tag defined in a previous .def-.endef pair.

.size expr The size pseudo op is used with the .def pseudo op. If name of .def is an object such as a structure or an array, this gives it a total size of expr. Expr must be a positive integer.

.scl expr The scl pseudo op is used with the .def pseudo op. With in the .def it gives name the storage class of expr. The type of expr should be positive.

.line expr The line pseudo op is used with the .def pseudo op. It defines the source line number of the definition of symbol name in the .def. Expr should yield an positive value.

.ln line [,addr] This pseudo op provides the relative source line number to the beginning of a function. It is used to pass info through to sdb.

.file name The file pseudo op is the source file name. Only one is allowed per source file. Name must be between 1 and 14 characters. This must be the first line an assembly file.

.endef The endef pseudo op is the ending bracket for a .def.

.def name The def pseudo op starts a symbolic description for symbol name. See

.endef. Name is a symbol name.

.dim expr [,expr] The dim pseudo op is used with the .def pseudo op. If the name of a .def is an array, the expressions give the dimensions. Up to 4 dimensions are accepted. The type of each expression should be positive.

3. Machine_Instructions

3.1 Differences between the UNIX 386 and the Intel 386 assemblers

This section describes the instructions that the assembler accepts. The detailed specification of how the particular instructions operate are not included. The operation of particular instructions is described in the Intel documentation.

The following describes the differences between the Unix 386 and Intel 386 assembly languages. This explanation covers all aspects of translation from Intel assembler to Unix 386 assembler.

This is a list of the differences between the Unix 386 assembly language and Intel's.

1. All register names use percent sign (%) as a prefix to distinguish them from symbol names.

2. Instructions with two (2) operands use the left as the source and the right as the destination. This follows the UNIX system's assembler convention, and it is reversed from Intel's notation.

3. Most instructions that can operate on a byte, word, or long may have "b", "w", or "l" appended to them. In general when an opcode is specified with no type suffix, it defaults to long. In general the UNIX 386 assembler derives its type information from the opcode, where as the Intel 386 assembler can derive its type information from the operand types. Where the type information is derived, motivates the b, w, and l suffixes used in the Unix 386 assembler.

3.2 Operands

Three kinds of operands are generally available to the instructions: register, memory, and immediate operands. Full descriptions of each type appear below. Indirect operands are available to jump and call instructions; but NO other instructions can use memory indirect operands.

The assembler always assumes it is generating code for a 32 bit segment. So when 16 bit data is called for (i.e. movw %ax, %bx) it will automatically generate the 16 bit data prefix byte.

Byte, Word, and Long registers are available on the 80386

processor. The code segment (%cs), instruction pointer (%eip), and the flag register are not available as explicit operands to the instructions.

The names of the byte, word, and long registers available as operands and a brief description appear below:

1. 8-bit (byte) general registers

   %al low byte of %ax register

   %ah high byte of %ax register

   %cl low byte of %cx register

   %ch high byte of %cx register

   %dl low byte of %dx register

   %dh high byte of %dx register

   %bl low byte of %bx register

   %bh high byte of %bx register

2. 16-bit general registers

   %ax low 16-bits of %eax register

   %cx low 16-bits of %ecx register

   %dx low 16-bits of %edx register

   %bx low 16-bits of %ebx register

   %sp low 16-bits of the stack pointer (%esp)

   %bp low 16-bits of the frame pointer (%ebp)

   %si low 16-bits of the source index register (%esi)

   %di low 16-bits of the destination index register (%edi)

3. 32-bit General Registers

   %eax 32-bit accumulator

   %ecx 32-bit general register

   %edx 32-bit general register

   %ebx 32-bit general register

   %esp 32-bit stack pointer

   %ebp 32-bit frame pointer

   %esi 32-bit source index register

   %edi 32-bit destination index register

4. Segment registers

%cs Code segment register, all references to the instruction space use this register.

%ds Data segment register, the default segment register for most references to memory operands.

%ss Stack segment register, the default segment register for memory operands in the stack. (i.e. default segment register for %bp %sp %esp and %ebp).

%es General purpose segment register
Some string instructions use this extra segment as their default segment.

%fs General purpose segment register

%gs General purpose segment register

3.3 Instruction_Descriptions

This section describes the Unix 5.3/386 instruction syntax. Refer to section 3.13.13.1 for the differences between the UNIX 386 and the Intel 386 assemblers.

Since the assembler assumes it is always generating code for a 32 bit segment it always assumes a 32 bit address, and it automatically predeeds word operations with a 16 bit data prefix byte.

In this section the following notation is used:

1. The mnemonics are expressed in a regular expression type syntax. Alternatives separated by a vertical bar (|) and enclosed with in square brackets, "[]", denote one of them must be chosen. Alternatives enclosed with in curly braces, "{}", denote one or none of the them may be used. The vertical bar (|) separates different suffixes for operators or operands. As an example when an 8, 16, or 32 bit immediate value is permitted in an instruction we would write: imm[8|16|32].

2. imm[8|16|32|48] - any immediate value, as they are defined above. Immediate values are defined using the regular expression syntax previously defined. When there is a choice between operand sizes the assembler will choose the smallest representation.

3. reg[8|16|32] - any general purpose register. Where each number indicates one of the following:
32: %eax, %ecx, %edx, %ebx, %esi, %edi,%ebp, %esp.
16: %ax, %cx, %dx, %bx, %si, %di, %bp, %sp.
8: %al, %ah, %cl, %ch, %dl, %dh, %bl, %bh.

4. mem[8|16|32|48] - any memory operand. The 8, 16, 32, and 48 suffixes represent byte, word, dword, and inter-segment memory address quantities, respectively.

5. r/m[8|16|32] - any general purpose register or memory operand. The operand type is determined from the suf-

fix. They are 8 = byte, 16 = word, and 32 = dword. The registers for each operand size are the same as reg[8|16|32] above.

6. creg - any control register The control registers are: %cr0, %cr2, or %cr3.

7. dreg - the debug register. The debug registers are: %db0, %db1, %db2, %db3, %db6, %db7.

8. sreg - any segment register The segment registers are: %cs, %ds, %ss, %es, %fs, %gs.

9. treg - the test register. The test registers are: %tr6 and %tr7

10. cc - condition codes. The condition codes are:

    1. a - jmp above
    2. ae - above or equal
    3. b - below
    4. be - below or equal
    5. c - carry
    6. e - equal
    7. g - greater
    8. ge - greater than or equal to
    9. l - less than
    10. le - less than or equal to
    11. na - not above
    12. nae - not above or equal to
    13. nb - not below
    14. nbe - not above or equal to
    15. nc - no carry
    16. ne - not equal
    17. ng - not greater than
    18. nge - not greater than or equal to
    19. nl - not less than
    20. nle - not less than or equal to
    21. no - not over flow
    22. np - not parity
    23. ns - not sign

24. nz - not zero

25. o - overflow

26. p - parity

27. pe - parity even

28. po - parity odd

29. s - sign

30. z - zero

11. disp[8|32] - the number of bits used to define the distance of a relative jump. Since the assembler only supports a 32 bit address space only 8 bit sign extended, and 32 bit address are supported.

12. immPtr - When the immediate form of a long call or a long jump is used the selector and offset are encoded as an immediate pointer (immPtr).

Addressing modes
Represented by: [sreg:][offset][([base][,index][,scale])].
Where all the items in the square brackets are optional, and at least one is necessary. If any of the items in side the parenthesis are used the parenthesis are mandatory.

Sreg is a segment register over ride prefix. It may be any segment register. If a segment over ride prefix is present it must be followed by a colon (:), before the offset component of the address. Sreg does not represent an address by itself. An address must contain an offset component.

Offset is a displacement from a segment base. It may be absolute or relocatable. A label is an example of a relocatable offset. A number is an example of an absolute offset.

Base and index can be any 32 bit register. Scale is a multiplication factor for the index register field. Please refer to the Intel documentation for more details on the 80386 addressing modes.

Following are some examples of addresses:

movl var, %eax
Move the contents of memory location var into %eax.

movl %cs:var, %eax
Move the contents of the memory location, var in the code segment into %eax.

movl $var, %eax
Move the address of var into %eax.

movl array_base(%esi), %eax
Add the address of memory location array_base to the content of %esi to get an address in memory. Move the content of this address into %eax.

movl (%ebx, %esi, 4), %eax
Multiply the content of %esi by 4, add this to the content of %ebx, to produce a memory reference. Move the content of this memory location into %eax.

movl struct_base(%ebx, %esi, 4), %eax
Multiply the content of %esi by 4, add this to the content of %ebx, add this to the address of struct_base, to produce an address. Move the content of this address into %eax.

A note about expressions and immediate values. An immediate value is an expression preceded by a dollar sign.

immediate: "$" expr

Immediate values carry the absolute or relocatable attributes of their expression component. Immediate values can not be used in an expression.

Immediate values should be considered as another form of address. The immediate form of address.

3.3.1 Processor_Extension_Instructions Please refer to the chapter on floating point support.

3.3.1.1 Control_and_Test_Register_Instructions

1. mov{l} creg, reg32

2. mov{l} dreg, reg32

3. mov{l} reg32, creg

4. mov{l} reg32, dreg

5. mov{l} treg, reg32

6. mov{l} reg32, treg

NOTE: The Unix assembler accepts "mov" or "movl" as exactly the same instruction for the control and test register group.

3.3.1.2 New_Condition_Code_Instructions

1. jcc disp32

2. setcc r/m8

3.3.1.3 Bit_Instructions All the new bit instructions are only defined for word and long register or memory operands.

1. bt{wl} reg[16|32], r/m[16|32]

2. bt{wl} imm8, r/m[16|32]

3. bts{wl} imm8, r/m[16|32]

4. bts{wl} reg[16|32], r/m[16|32]

5. btr{wl} imm8, r/m[16|32]

6. btr{wl} reg[16|32], r/m[16|32]

7. btc{wl} imm8, r/m[16|32]

8. btc{wl} reg[16|32], r/m[16|32]

9. bsf{wl} reg[16|32], r/m[16|32]

10. bsr{wl} reg[16|32], r/m[16|32]

11. shld{wl} imm8, reg[16|32], r/m[16|32]

12. shld{wl} reg[16|32], r/m[16|32]

13. shrd{wl} imm8, reg[16|32], r/m[16|32]

14. shrd{wl} reg[16|32], r/m[16|32]

NOTE: All the bit operation mnemonics with out a type suffix default to long.

3.3.1.4 New_Arithmetic_Instruction

1. imul r/m[16|32], reg[16|32]

NOTE: This is the uncharacterized multiply. It has a 16 or 32 bit product, as opposed to a 32 or 64 bit product.

3.3.1.5 New_Move_with_Zero_or_Sign_Extension_Instructions

1. movzbw r/m8, reg16

2. movzbl r/m8, reg32

3. movzwl r/m16, reg32

4. movsbw r/m8, reg16

5. movsbl r/m8, reg32

6. movswl r/m16, reg32

3.3.2 Data_Movement_Instructions

1. clr{bwl} r/m[8|16|32]

2. lea{wl} mem32, reg[16|32]

3. mov{bwl} r/m[8|16|32], reg[8|16|32]

4. mov{bwl} reg[8|16|32], r/m[8|16|32]

5. mov{bwl} imm[8|16|32], r/m[8|16|32]

6. pop{wl} r/m[16|32]

7. popa{wl}

8. push{bwl} imm[8|16|32]

9. push{wl} r/m[16|32]

10. pusha{wl}

11. xchg{bwl} reg[8|16|32], r/m[8|16|32]

NOTE1: pushb sign extends the immediate byte to a long, and pushes a long (4 bytes) onto the stack.

NOTE2: When a type suffix is not used with a data movement mnemonic the type defaults to long. The Unix assembler does not derive the type of the operands from the operands.

3.3.3 Segment_Register_Instructions

1. lds{wl} mem[32|48], reg[16|32]

2. les{wl} mem[32|48], reg[16|32]

3. lfs{wl} mem[32|48], reg[16|32]

4. lgs{wl} mem[32|48], reg[16|32]

5. lss{wl} mem[32|48], reg[16|32]

6. movw sreg[cs|ds|ss|es] , r/m16

7. movw r/m16, sreg[cs|ds|ss|es]

8. popw sreg[ds|ss|es|fs|gs]

9. pushw sreg[cs|ds|ss|es|fs|gs]

NOTE1: The pushw and popw push and pop 16 bit quantities. This is done by using an data size over ride byte (OSP) byte.

NOTE2: When the type suffix is not used with the lds, les, lfs, lgs, and lss instructions a 48 bit pointer is assumed.

NOTE3: Since the assembler assumes no type suffix means a type of long, the type suffix of "w" when working with the segment registers is mandatory.

3.3.4 I/O_Instructions

1. in{bwl} imm8

2. in{bwl} %dx

3. ins{bwl} %dx

4. out{bwl} imm8

5. out{bwl} %dx

6. outs{bwl} %dx

NOTE1: When the type suffix is left off the I/O instructions they default to long. So in = inl, out = outl, ins = insl,

and outs = outsl.

3.3.5 Flag_Instructions

1. lahf
2. sahf
3. popf{wl}
4. pushf{wl}
5. cmc
6. clc
7. stc
8. cli
9. sti
10. cld
11. std

NOTE: When the type suffix not used the pushf and popf instructions default to long. Pushf = pushfl and popf = popfl. A pushw or popw will push or pop a 16 bit quantity. This is done by using the OSP prefix byte

3.3.6 Arithmetic/Logical_Instructions

1. add{bwl} reg[8|16|32], r/m[8|16|32]
2. add{bwl} r/m[8|16|32], reg[8|16|32]
3. add{bwl} imm[8|16|32], r/m[8|16|32]
4. adc{bwl} reg[8|16|32], r/m[8|16|32]
5. adc{bwl} r/m[8|16|32], reg[8|16|32]
6. adc{bwl} imm[8|16|32], r/m[8|16|32]
7. sub{bwl} reg[8|16|32], r/m[8|16|32]
8. sub{bwl} r/m[8|16|32], reg[8|16|32]
9. sub{bwl} imm[8|16|32], r/m[8|16|32]
10. sbb{bwl} reg[8|16|32], r/m[8|16|32]
11. sbb{bwl} r/m[8|16|32], reg[8|16|32]
12. sbb{bwl} imm[8|16|32], r/m[8|16|32]
13. cmp{bwl} reg[8|16|32], r/m[8|16|32]
14. cmp{bwl} r/m[8|16|32], reg[8|16|32]
15. cmp{bwl} imm[8|16|32], r/m[8|16|32]

16. inc{bwl} r/m[8|16|32]

17. dec{bwl} r/m[8|16|32]

18. test{bwl} reg[8|16|32], r/m[8|16|32]

19. test{bwl} r/m[8|16|32], reg[8|16|32]

20. test{bwl} imm[8|16|32], r/m[8|16|32]

21. sal{bwl} imm8, r/m[8|16|32]

22. sal{bwl} %cl, r/m[8|16|32]

23. shl{bwl} imm8, r/m[8|16|32]

24. shl{bwl} %cl, r/m[8|16|32]

25. sar{bwl} imm8, r/m[8|16|32]

26. sar{bwl} %cl, r/m[8|16|32]

27. shr{bwl} imm8, r/m[8|16|32]

28. shr{bwl} %cl, r/m[8|16|32]

29. not{bwl} r/m[8|16|32]

30. neg{bwl} r/m[8|16|32]

31. bound{wl} reg[16|32], r/m[16|32]

32. and{bwl} reg[8|16|32], r/m[8|16|32]

33. and{bwl} r/m[8|16|32], reg[8|16|32]

34. and{bwl} imm[8|16|32], r/m[8|16|32]

35. or{bwl} reg[8|16|32], r/m[8|16|32]

36. or{bwl} r/m[8|16|32], reg[8|16|32]

37. or{bwl} imm[8|16|32], r/m[8|16|32]

38. xor{bwl} reg[8|16|32], r/m[8|16|32]

39. xor{bwl} r/m[8|16|32], reg[8|16|32]

40. xor{bwl} imm[8|16|32], r/m[8|16|32]

NOTE: When the type suffix is not included in an arithmetic or logical instruction it defaults to a long.

3.3.7 Multiply_and_Divide

1. imul{wl} imm[16|32], r/m[16|32], reg[16|32]

2. mul{bwl} r/m[8|16|32]

3. div{bwl} r/m[8|16|32]

4. idiv{bwl} r/m[8|16|32]

NOTE: When the type suffix is not included in a multiply or divide instruction it defaults to a long.

3.3.8 Conversion_Instructions

1. cbtw
2. cwtd
3. cwtl
4. cltd

NOTE: convert byte to word: %al -> %ax
convert word to double: %ax -> %dx:%ax
convert word to long: %ax -> %eax
convert long to double: %eax -> %edx:%eax

3.3.9 Decimal_Arithmetic_Instructions

1. daa
2. das
3. aaa
4. aas
5. aam
6. aad

3.3.10 _Coprocessor_Instructions

1. wait
2. esc

3.3.11 String_Instructions

1. movs[bwl]
2. movs - same as movsl
3. smov[bwl] same as movs[bwl]
4. smov - same as smovl
5. cmps[bwl]
6. cmps - same as cmpsl
7. scmp[bwl] same as cmps[bwl]
8. scmp - same as scmpl
9. stos[bwl]
10. stos - same as stosl

11. ssto[bwl] same as stos[bwl]
12. ssto - same as sstol
13. lods[bwl]
14. lods - same as lodsl
15. slod[bwl] same as lods[bwl]
16. slod - same as slodl
17. scas[bwl]
18. scas - same as scasl
19. ssca[bwl] same as scas[bwl]
20. ssca - same as sscal
21. xlat
22. rep
23. repnz
24. repz

NOTE: All Intel string op mnemonics default to longs.

3.3.12 _Procedure_Call_and_Return

1. lcall immPtr
2. lcall r/m48 (indirect)
3. lret
4. lret imm16
5. call disp32
6. call r/m32 (indirect)
7. ret
8. ret imm16
9. enter imm16, imm8
10. leave

3.3.13 Jump_Instructions

1. jcc disp[8|32]
2. jcxz disp[8|32]
3. loop disp[8|32]
4. loopnz disp[8|32]

5. loopz disp[8|32]

6. jmp disp[8|32]

7. ljmp immPtr

8. jmp r/m32 (indirect)

9. ljmp r/m48 (indirect)

NOTE: The UNIX 386 assembler optimizes for SDI's (Span Dependent Instructions). So intra-segment jumps are optimized to their short forms when possible.

3.3.14 Interrupt_Instructions

1. int 3

2. int imm8

3. into

4. iret

3.3.15 Protection_Model_Instructions

1. sldt r/m16

2. str r/m16

3. lldt r/m16

4. ltr r/m16

5. verr r/m16

6. verw r/m16

7. sgdt r/m32

8. sidt r/m32

9. lgdt r/m32

10. lidt r/m32

11. smsw r/m32

12. lmsw r/m32

13. lar r/m32, reg32

14. lsl r/m32, reg32

15. clts

3.3.16 Miscellaneous_Instructions

1. lock

2. nop

3. hlt

4. addr16

5. data16

TRANSLATION TABLES FOR UNIX TO INTEL FLOAT MNEMONICS

The following tables show the relationship between the Unix and Intel mnemonics. The mnemonics are organized into the same functional categories as the Intel mnemonics. The Intel mnemonics appear in section two of the 80287 numeric supplement.

The notational conventions used in the table are: When letters appear with in square brackets , "[]", exactly one of the letters are required. If letters appear with in curly braces, "{}", then either one or none of the letters are required. When a a group of letters is separated from other letters by a bar, "|", with in square brackets or curly braces then the group of letters between the bars or a bar and a closing bracket or brace are considered an atomic unit. As an example, "fld[lst] means: fldl, flds, or fldt. Where fst{ls} means: fst, fstl, or fsts. And fild{l|ll} means: fild, fildl, or fildll.

The Unix operators are built from the Intel operators by adding suffixes to them. The 80287 deals with three data types, integer, packed decimal, and reals. The Unix assembler is not typed. So the operator has to carry with it the type of data item it is operating on. If the operation is on an integer the following suffixes apply: l for Intel's short (32 bit), and ll for Intel's long (64 bits). If the operator applies to reals then: s is short (32 bits), l is long (64 bits), and t is temporary real (80 bits).

Real Transfers

UNIX	INTEL	Operation
fld[lst]	fld	load real
fst{ls}	fst	store real
fstp{lst}	fstp	store real and pop
fxch	fxch	exchange registers

Integer Transfers

UNIX	INTEL	Operation
fild{l|ll}	fild	integer load
fist{l}	fist	integer store
fistp{l|ll}	fistp	integer store and pop

Packed Decimal Transfers

UNIX	INTEL	Operation

fbld	fbld	Packed decimal (BCD) load
fbstp	fbstp	Packed decimal (BCD) store and pop

Addition

UNIX	INTEL	Operation
fadd{ls}	fadd	real add
faddp	faddp	real add and pop
fiadd{l}	fiadd	integer add

Subtraction

UNIX	INTEL	Operation
fsub{ls}	fsub	subtract real
fsubp	fsubp	subtract real and pop
fsubr{ls}	fsubr	subtract real reversed
fsubrp	fsubrp	subtract real reversed and pop
fisub{l}	fisub	integer subtract
fisubr{l}	fisubr	integer subtract reverse

Multiplication

UNIX	INTEL	Operation
fmul{ls}	fmul	multiply real
fmulp	fmulp	multiply real and pop
fimul{l}	fimul	integer multiply

Division

UNIX	INTEL	Operation
fdiv{ls}	fdiv	divide real
fdivp	fdivp	divide real and pop
fdivr{ls}	fdivr	divide real reversed
fdivrp	fdivrp	divide real reversed and pop
fidiv{l}	fidiv	integer divide
fidivr{l}	fidivr	integer divide reversed

Other Arithmetic Operations

UNIX	INTEL	Operation
fsqrt	fsqrt	square root
fscale	fscale	scale
fprem	fprem	partial remainder
frndint	frndint	round to integer
fxtract	fxtract	extract exponent and significand
fabs	fabs	absolute value
fchs	fchs	change sign

Comparison Instructions

UNIX	INTEL	Operation

fcom{ls}	fcom	compare real
fcomp{ls}	fcomp	compare real and pop
fcompp	fcompp	compare real and pop twice
ficom{l}	ficom	integer compare
ficomp{l}	ficomp	integer compare and pop
ftst	ftst	test
fxam	fxam	examine

Transcendental Instructions

UNIX	INTEL	Operation
fptan	fptan	partial tangent
fpatan	fpatan	partial arctangent
f2xm1	f2xm1	2^x - 1
fyl2x	fyl2x	Y * log2X
fyl2xp1	fyl2xp1	Y * log2(X+1)

Constant Instructions

UNIX	INTEL	Operation
fldl2e	fldl2e	load logeE
fldl2t	fldl2t	load log2 10
fldlg2	fldlg2	load log2 2
fldln2	fldln2	load loge2
fldpi	fldpi	load pie
fldz	fldz	load + 0

Processor Control Instructions

UNIX	INTEL	Operation
finit/fnint	finit/fnint	initialize processor
fnop	fnop	no operation
fsave/fnsave	fsave/fnsave	save state
fstcw/fnstcw	fstcw/fnstcw	store control word
fstenv/fnstenv	fstenv/fnstenv	store environment
fstsw/fnstsw	fstsw/fnstsw	store status word
frstor	frstor	restore state
fsetpm	fsetpm	set protected mode
fwait	fwait	CPU wait
fclex/fnclex	fclex/fnclex	clear exceptions
fdecstp	fdecstp	decrement stack pointer
ffree	ffree	free registers
fincstp	fincstp	increment stack pointer