x86 Assembly Language Reference Manual

Beta

Part No: 821–1608–02
November 2010

ORACLE

110425@25097

Contents

Tables

Preface

The *x86 Assembly Language Reference Manual* documents the syntax of the Solaris x86 assembly language. This manual is provided to help experienced programmers understand the assembly language output of Solaris compilers. This manual is neither an introductory book about assembly language programming nor a reference manual for the x86 architecture.

Note – This Oracle Solaris release supports systems that use the SPARC and x86 families of processor architectures. The supported systems appear in the Solaris OS: Hardware Compatibility Lists (`http://www.sun.com/bigadmin/hcl`). This document cites any implementation differences between the platform types.

In this document these x86 related terms mean the following:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64" relates specifically to 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

For supported systems, see the *Oracle Solaris OS: Hardware Compatibility Lists*.

Who Should Use This Book

This manual is intended for experienced x86 assembly language programmers who are familiar with the x86 architecture.

Before You Read This Book

You should have a thorough knowledge of assembly language programming in general and be familiar with the x86 architecture in specific. You should be familiar with the ELF object file format. This manual assumes that you have the following documentation available for reference:

- *IA-32 Intel Architecture Software Developer's Manual* (Intel Corporation, 2004). Volume 1: *Basic Architecture*. Volume 2: *Instruction Set Reference A-M*. Volume 3: *Instruction Set Reference N-Z*. Volume 4: *System Programming Guide*.

- *AMD64 Architecture Programmer's Manual* (Advanced Micro Devices, 2003). Volume 1: *Application Programming*. Volume 2: *System Programming*. Volume 3: *General-Purpose and System Instructions*. Volume 4: *128-Bit Media Instructions*. Volume 5: *64-Bit Media and x87 Floating-Point Instructions*.
- *Linker and Libraries Guide*
- *Sun Studio: C User's Guide*
- *Sun Studio: Fortran User's Guide* and *Fortran Programming Guide*
- Man pages for the `as(1)`, `ld(1)`, and `dis(1)` utilities.

How This Book Is Organized

Chapter 1, "Overview of the Solaris x86 Assembler," provides an overview of the x86 functionality supported by the Solaris x86 assembler.

Chapter 2, "Solaris x86 Assembly Language Syntax," documents the syntax of the Solaris x86 assembly language.

Chapter 3, "Instruction Set Mapping," maps Solaris x86 assembly language instruction mnemonics to the native x86 instruction set.

Documentation, Support, and Training

See the following web sites for additional resources:

- Documentation (`http://docs.sun.com`)
- Support (`http://www.oracle.com/us/support/systems/index.html`)
- Training (`http://education.oracle.com`) – Click the Sun link in the left navigation bar.

Oracle Software Resources

Oracle Technology Network (`http://www.oracle.com/technetwork/index.html`) offers a range of resources related to Oracle software:

- Discuss technical problems and solutions on the Discussion Forums (`http://forums.oracle.com`).
- Get hands-on step-by-step tutorials with Oracle By Example (`http://www.oracle.com/technetwork/tutorials/index.html`).
- Download Sample Code (`http://www.oracle.com/technology/sample_code/index.html`).

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

◆ ◆ ◆ CHAPTER 1

Overview of the Solaris x86 Assembler

This chapter provides a brief overview of the Solaris x86 assembler `as`. This chapter discusses the following topics:

Assembler Overview

The Solaris x86 assembler `as` translates Solaris x86 assembly language into Executable and Linking Format (ELF) relocatable object files that can be linked with other object files to create an executable file or a shared object file. (See Chapter 7, "Object File Format," in *Linker and Libraries Guide*, for a complete discussion of ELF object file format.) The assembler supports macro processing by the C preprocessor (`cpp`) or the `m4` macro processor. The assembler supports the instruction sets of the following CPUs:

Intel 8086/8088 processors
Intel 286 processor
Intel386 processor
Intel486 processor
Intel Pentium processor
Intel Pentium Pro processor
Intel Pentium II processor
Pentium II Xeon processor
Intel Celeron processor
Intel Pentium III processor
Pentium III Xeon processor
Advanced Micro Devices Athlon processor
Advanced Micro Devices Opteron processor

Syntax Differences Between x86 Assemblers

There is no standard assembly language for the x86 architecture. Vendor implementations of assemblers for the x86 architecture instruction sets differ in syntax and functionality. The syntax of the Solaris x86 assembler is compatible with the syntax of the assembler distributed with earlier releases of the UNIX operating system (this syntax is sometimes termed "AT&T syntax"). Developers familiar with other assemblers derived from the original UNIX assemblers, such as the Free Software Foundation's `gas`, will find the syntax of the Solaris x86 assembler very straightforward.

However, the syntax of x86 assemblers distributed by Intel and Microsoft (sometimes termed "Intel syntax") differs significantly from the syntax of the Solaris x86 assembler. These differences are most pronounced in the handling of instruction operands:

- The Solaris and Intel assemblers use the opposite order for source and destination operands.
- The Solaris assembler specifies the size of memory operands by adding a suffix to the instruction mnemonic, while the Intel assembler prefixes the memory operands.
- The Solaris assembler prefixes immediate operands with a dollar sign ($) (ASCII 0x24), while the Intel assembler does not delimit immediate operands.

See Chapter 2, "Solaris x86 Assembly Language Syntax," for additional differences between x86 assemblers.

Assembler Command Line

During the translation of higher-level languages such as C and Fortran, the compilers might invoke `as` using the alias `fbe` ("Fortran back end"). You can invoke the assembler manually from the shell command line with either name, `as` or `fbe`. See the `as`(1) man page for the definitive discussion of command syntax and command line options.

◆ ◆ ◆ CHAPTER 2

Solaris x86 Assembly Language Syntax

This chapter documents the syntax of the Solaris x86 assembly language.

Lexical Conventions

This section discusses the lexical conventions of the Solaris x86 assembly language.

Statements

An x86 assembly language program consists of one or more files containing *statements*. A *statement* consists of *tokens* separated by *whitespace* and terminated by either a newline character (ASCII 0x0A) or a semicolon (;) (ASCII 0x3B). *Whitespace* consists of spaces (ASCII 0x20), tabs (ASCII 0x09), and formfeeds (ASCII 0x0B) that are not contained in a string or comment. More than one statement can be placed on a single input line provided that each statement is terminated by a semicolon. A statement can consist of a *comment*. *Empty statements*, consisting only of whitespace, are allowed.

Comments

A *comment* can be appended to a statement. The comment consists of the slash character (/) (ASCII 0x2F) followed by the text of the comment. The comment is terminated by the newline that terminates the statement.

Labels

A *label* can be placed at the beginning of a statement. During assembly, the label is assigned the current value of the active location counter and serves as an instruction operand. There are two types of lables: *symbolic* and *numeric*.

Symbolic Labels

A *symbolic* label consists of an *identifier* (or *symbol*) followed by a colon (:) (ASCII 0x3A). Symbolic labels must be defined only once. Symbolic labels have *global* scope and appear in the object file's symbol table.

Symbolic labels with identifiers beginning with a period (.) (ASCII 0x2E) are considered to have *local* scope and are not included in the object file's symbol table.

Numeric Labels

A *numeric* label consists of a single digit in the range zero (0) through nine (9) followed by a colon (:). Numeric labels are used only for local reference and are not included in the object file's symbol table. Numeric labels have limited scope and can be redefined repeatedly.

When a numeric label is used as a reference (as an instruction operand, for example), the suffixes `b` ("backward") or `f` ("forward") should be added to the numeric label. For numeric label *N*, the reference *N*`b` refers to the nearest label *N* defined *before* the reference, and the reference *N*`f` refers to the nearest label *N* defined *after* the reference. The following example illustrates the use of numeric labels:

```
1:          / define numeric label "1"
one:        / define symbolic label "one"

/ ... assembler code ...

jmp   1f    / jump to first numeric label "1" defined
            / after this instruction
            / (this reference is equivalent to label "two")

jmp   1b    / jump to last numeric label "1" defined
            / before this instruction
            / (this reference is equivalent to label "one")

1:          / redefine label "1"
two:        / define symbolic label "two"

jmp   1b    / jump to last numeric label "1" defined
            / before this instruction
            / (this reference is equivalent to label "two")
```

Tokens

There are five classes of tokens:

- Identifiers (symbols)
- Keywords
- Numerical constants
- String Constants
- Operators

Identifiers

An *identifier* is an arbitrarily-long sequence of letters and digits. The first character must be a letter; the underscore (_) (ASCII 0x5F) and the period (.) (ASCII 0x2E) are considered to be letters. Case is significant: uppercase and lowercase letters are different.

Keywords

Keywords such as x86 instruction mnemonics ("opcodes") and assembler directives are reserved for the assembler and should not be used as identifiers. See Chapter 3, "Instruction Set Mapping," for a list of the Solaris x86 mnemonics. See "Assembler Directives" on page 19 for the list of `as` assembler directives.

Numerical Constants

Numbers in the x86 architecture can be *integers* or *floating point*. Integers can be *signed* or *unsigned*, with signed integers represented in two's complement representation. Floating-point numbers can be: single-precision floating-point; double-precision floating-point; and double-extended precision floating-point.

Integer Constants

Integers can be expressed in several bases:

- **Decimal.** Decimal integers begin with a non-zero digit followed by zero or more decimal digits (0–9).
- **Binary.** Binary integers begin with "0b" or "0B" followed by zero or more binary digits (0, 1).
- **Octal.** Octal integers begin with zero (0) followed by zero or more octal digits (0–7).
- **Hexadecimal.** Hexadecimal integers begin with "0x" or "0X" followed by one or more hexadecimal digits (0–9, A–F). Hexadecimal digits can be either uppercase or lowercase.

Floating Point Constants

Floating point constants have the following format:

- **Sign** (optional) – either plus (+) or minus (–)
- **Integer** (optional) – zero or more decimal digits (0–9)
- **Fraction** (optional) – decimal point (.) followed by zero or more decimal digits
- **Exponent** (optional) – the letter "e" or "E", followed by an optional sign (plus or minus), followed by one or more decimal digits (0–9)

A valid floating point constant must have either an integer part or a fractional part.

String Constants

A *string* constant consists of a sequence of characters enclosed in double quotes (") (ASCII 0x22). To include a double-quote character ("), single-quote character ('), or backslash character (\) within a string, precede the character with a backslash (\) (ASCII 0x5C). A character can be expressed in a string as its ASCII value in octal preceded by a backslash (for example, the letter "J" could be expressed as "\112"). The assembler accepts the following escape sequences in strings:

Escape Sequence	Character Name	ASCII Value (hex)
`\n`	newline	0A
`\r`	carriage return	0D
`\b`	backspace	08
`\t`	horizontal tab	09
`\f`	form feed	0C
`\v`	vertical tab	0B

Operators

The assembler supports the following operators for use in expressions. Operators have no assigned precedence. Expressions can be grouped in square brackets (`[]`) to establish precedence.

`+` Addition

`-` Subtraction

`\*` Multiplication

`\/` Division

`&` Bitwise logical AND

	Bitwise logical OR
>>	Shift right
<<	Shift left
\%	Remainder
!	Bitwise logical AND NOT
^	Bitwise logical XOR

Note – The asterisk (*), slash (/), and percent sign (%) characters are overloaded. When used as operators in an expression, these characters must be preceded by the backslash character (\).

Instructions, Operands, and Addressing

Instructions are operations performed by the CPU. *Operands* are entities operated upon by the instruction. *Addresses* are the locations in memory of specified data.

Instructions

An *instruction* is a statement that is executed at runtime. An x86 instruction statement can consist of four parts:

- Label (optional)
- Instruction (required)
- Operands (instruction specific)
- Comment (optional)

See "Statements" on page 13 for the description of labels and comments.

The terms *instruction* and *mnemonic* are used interchangeably in this document to refer to the names of x86 instructions. Although the term *opcode* is sometimes used as a synonym for *instruction*, this document reserves the term *opcode* for the hexadecimal representation of the instruction value.

For most instructions, the Solaris x86 assembler mnemonics are the same as the Intel or AMD mnemonics. However, the Solaris x86 mnemonics might appear to be different because the Solaris mnemonics are suffixed with a one-character modifier that specifies the size of the instruction operands. That is, the Solaris assembler derives its operand type information from the instruction name and the suffix. If a mnemonic is specified with no type suffix, the operand type defaults to `long`. Possible operand types and their instruction suffixes are:

`b` Byte (8–bit)

`w` Word (16–bit)

`l` Long (32–bit) (default)

`q` Quadword (64–bit)

The assembler recognizes the following suffixes for x87 floating-point instructions:

[no suffix] Instruction operands are registers only

`l` (“long”) Instruction operands are 64–bit

`s` (“short”) Instruction operands are 32–bit

See Chapter 3, “Instruction Set Mapping,” for a mapping between Solaris x86 assembly language mnemonics and the equivalent Intel or AMD mnemonics.

Operands

An x86 instruction can have zero to three operands. Operands are separated by commas (`,`) (ASCII 0x2C). For instructions with two operands, the first (lefthand) operand is the *source* operand, and the second (righthand) operand is the *destination* operand (that is, *source*→*destination*).

Note – The Intel assembler uses the opposite order (*destination*←*source*) for operands.

Operands can be *immediate* (that is, constant expressions that evaluate to an inline value), *register* (a value in the processor number registers), or *memory* (a value stored in memory). An *indirect* operand contains the address of the actual operand value. Indirect operands are specified by prefixing the operand with an asterisk (`*`) (ASCII 0x2A). Only jump and call instructions can use indirect operands.

- *Immediate* operands are prefixed with a dollar sign (`$`) (ASCII 0x24)
- *Register* names are prefixed with a percent sign (`%`) (ASCII 0x25)
- *Memory* operands are specified either by the name of a variable or by a register that contains the address of a variable. A variable name implies the address of a variable and instructs the computer to reference the contents of memory at that address. Memory references have the following syntax:
  segment`:`*offset*`(`*base*`,` *index*`,` *scale*`)`.
  - *Segment* is any of the x86 architecture segment registers. *Segment* is optional: if specified, it must be separated from *offset* by a colon (`:`). If *segment* is omitted, the value of `%ds` (the default segment register) is assumed.

- *Offset* is the displacement from *segment* of the desired memory value. *Offset* is optional.
- *Base* and *index* can be any of the general 32–bit number registers.
- *Scale* is a factor by which *index* is to be multipled before being added to *base* to specify the address of the operand. *Scale* can have the value of 1, 2, 4, or 8. If *scale* is not specified, the default value is 1.

Some examples of memory addresses are:

`movl var, %eax`	Move the contents of memory location `var` into number register `%eax`.
`movl %cs:var, %eax`	Move the contents of memory location `var` in the code segment (register `%cs`) into number register `%eax`.
`movl $var, %eax`	Move the address of `var` into number register `%eax`.
`movl array_base(%esi), %eax`	Add the address of memory location `array_base` to the contents of number register `%esi` to determine an address in memory. Move the contents of this address into number register `%eax`.
`movl (%ebx, %esi, 4), %eax`	Multiply the contents of number register `%esi` by 4 and add the result to the contents of number register `%ebx` to produce a memory reference. Move the contents of this memory location into number register `%eax`.
`movl struct_base(%ebx, %esi, 4), %eax`	Multiply the contents of number register `%esi` by 4, add the result to the contents of number register `%ebx`, and add the result to the address of `struct_base` to produce an address. Move the contents of this address into number register `%eax`.

Assembler Directives

Directives are commands that are part of the assembler syntax but are not related to the x86 processor instruction set. All assembler directives begin with a period (.) (ASCII 0x2E).

`.align` *integer*, *pad*
The `.align` directive causes the next data generated to be aligned modulo *integer* bytes. *Integer* must be a positive integer expression and must be a power of 2. If specified, *pad* is an integer bye value used for padding. The default value of *pad* for the `text` section is 0x90 (nop); for other sections, the default value of *pad* is zero (0).

`.ascii` "*string*"
The `.ascii` directive places the characters in *string* into the object module at the current location but does *not* terminate the string with a null byte (\0). *String* must be enclosed in double quotes (") (ASCII 0x22). The `.ascii` directive is not valid for the `.bss` section.

`.bcd` *integer*
The `.bcd` directive generates a packed decimal (80-bit) value into the current section. The `.bcd` directive is not valid for the `.bss` section.

`.bss`
The `.bss` directive changes the current section to `.bss`.

`.bss` *symbol*, *integer*
Define *symbol* in the `.bss` section and add *integer* bytes to the value of the location counter for `.bss`. When issued with arguments, the `.bss` directive does not change the current section to `.bss`. *Integer* must be positive.

`.byte` *byte1*,*byte2*,...,*byteN*
The `.byte` directive generates initialized bytes into the current section. The `.byte` directive is not valid for the `.bss` section. Each *byte* must be an 8-bit value.

`.2byte` *expression1*, *expression2*, ..., *expressionN*
Refer to the description of the `.value` directive.

`.4byte` *expression1*, *expression2*, ..., *expressionN*
Refer to the description of the `.long` directive.

`.8byte` *expression1*, *expression2*, ..., *expressionN*
Refer to the description of the `.quad` directive.

`.comm` *name*, *size*, *alignment*
The `.comm` directive allocates storage in the `data` section. The storage is referenced by the identifier *name*. *Size* is measured in bytes and must be a positive integer. *Name* cannot be predefined. *Alignment* is optional. If *alignment* is specified, the address of *name* is aligned to a multiple of *alignment*.

`.data`
The `.data` directive changes the current section to `.data`.

`.double` *float*
The `.double` directive generates a double-precision floating-point constant into the current section. The `.double` directive is not valid for the `.bss` section.

`.even`
The `.even` directive aligns the current program counter (.) to an even boundary.

`.ext` *expression1*`,` *expression2*`, ...,` *expressionN*
The `.ext` directive generates an 80387 80–bit floating point constant for each *expression* into the current section. The `.ext` directive is not valid for the `.bss` section.

`.file` "*string*"
The `.file` directive creates a symbol table entry where *string* is the symbol name and `STT_FILE` is the symbol table type. *String* specifies the name of the source file associated with the object file.

`.float` *float*
The `.float` directive generates a single-precision floating-point constant into the current section. The `.float` directive is not valid in the `.bss` section.

`.globl` *symbol1*`,` *symbol2*`, ...,` *symbolN*
The `.globl` directive declares each *symbol* in the list to be *global*. Each symbol is either defined externally or defined in the input file and accessible in other files. Default bindings for the symbol are overridden. A global symbol definition in one file satisfies an undefined reference to the same global symbol in another file. Multiple definitions of a defined global symbol are not allowed. If a defined global symbol has more than one definition, an error occurs. The `.globl` directive only declares the symbol to be global in scope, it does not define the symbol.

`.group` *group*`,` *section*`, #comdat`
The `.group` directive adds *section* to a COMDAT *group*. Refer to "COMDAT Section" in *Linker and Libraries Guide* for additional information about COMDAT.

`.hidden` *symbol1*`,` *symbol2*`, ...,` *symbolN*
The `.hidden` directive declares each *symbol* in the list to have *hidden* linker scoping. All references to *symbol* within a dynamic module bind to the definition within that module. *Symbol* is not visible outside of the module.

`.ident` "*string*"
The `.ident` directive creates an entry in the `.comment` section containing *string*. *String* is any sequence of characters, not including the double quote ("). To include the double quote character within a string, precede the double quote character with a backslash (\) (ASCII 0x5C).

`.lcomm` *name*`,` *size*`,` *alignment*
The `.lcomm` directive allocates storage in the `.bss` section. The storage is referenced by the symbol *name*, and has a size of *size* bytes. *Name* cannot be predefined, and *size* must be a positive integer. If *alignment* is specified, the address of *name* is aligned to a multiple of *alignment* bytes. If *alignment* is not specified, the default alignment is 4 bytes.

`.local` *symbol1*`,` *symbol2*`, ...,` *symbolN*
The `.local` directive declares each *symbol* in the list to be *local*. Each symbol is defined in the input file and not accessible to other files. Default bindings for the symbols are overridden. Symbols declared with the `.local` directive take precedence over *weak* and *global* symbols. (See "Symbol Table Section" in *Linker and Libraries Guide* for a description of global and

weak symbols.) Because local symbols are not accessible to other files, local symbols of the same name may exist in multiple files. The `.local` directive only declares the symbol to be local in scope, it does not define the symbol.

`.long` *expression1*`,` *expression2*`, ...,` *expressionN*
The `.long` directive generates a long integer (32-bit, two's complement value) for each *expression* into the current section. Each *expression* must be a 32–bit value and must evaluate to an integer value. The `.long` directive is not valid for the `.bss` section.

`.popsection`
The `.popsection` directive pops the top of the section stack and continues processing of the popped section.

`.previous`
The `.previous` directive continues processing of the previous section.

`.pushsection` *section*
The `.pushsection` directive pushes the specified section onto the section stack and switches to another section.

`.quad` *expression1*`,` *expression2*`, ...,` *expressionN*
The `.quad` directive generates an initialized word (64-bit, two's complement value) for each *expression* into the current section. Each *expression* must be a 64-bit value, and must evaluate to an integer value. The `.quad` directive is not valid for the `.bss` section.

`.rel` *symbol*`@` *type*
The `.rel` directive generates the specified relocation entry *type* for the specified *symbol*. The `.lit` directive supports TLS (thread-local storage). Refer to Chapter 8, "Thread-Local Storage," in *Linker and Libraries Guide* for additional information about TLS.

`.section` *section*`,` *attributes*
The `.section` directive makes *section* the current section. If *section* does not exist, a new section with the specified name and attributes is created. If *section* is a non-reserved section, *attributes* must be included the first time *section* is specified by the `.section` directive.

`.set` *symbol*`,` *expression*
The `.set` directive assigns the value of *expression* to *symbol*. *Expression* can be any legal expression that evaluates to a numerical value.

`.skip` *integer*, *value*
While generating values for any data section, the `.skip` directive causes *integer* bytes to be skipped over, or, optionally, filled with the specified *value*.

`.sleb128` *expression*
The `.sleb128` directive generates a signed, little-endian, base 128 number from *expression*.

`.string` "*string*"
The `.string` directive places the characters in *string* into the object module at the current location and terminates the string with a null byte (\0). *String* must be enclosed in double quotes (") (ASCII 0x22). The `.string` directive is not valid for the `.bss` section.

`.symbolic` *symbol1*`,` *symbol2*`, ...,` *symbolN*
The `.symbolic` directive declares each *symbol* in the list to have*symbolic* linker scoping. All references to *symbol* within a dynamic module bind to the definition within that module. Outside of the module, *symbol* is treated as global.

`.tbss`
The `.tbss` directive changes the current section to `.tbss`. The `.tbss` section contains uninitialized TLS data objects that will be initialized to zero by the runtime linker.

`.tcomm`
The `.tcomm` directive defines a TLS common block.

`.tdata`
The `.tdata` directive changes the current section to `.tdata`. The `.tdata` section contains the initialization image for initialized TLS data objects.

`.text`
The `.text` directive defines the current section as `.text`.

`.uleb128` *expression*
The `.uleb128` directive generates an unsigned, little-endian, base 128 number from *expression*.

`.value` *expression1*`,` *expression2*`, ...,` *expressionN*
The `.value` directive generates an initialized word (16-bit, two's complement value) for each *expression* into the current section. Each *expression* must be a 16-bit integer value. The `.value` directive is not valid for the `.bss` section.

`.weak` *symbol1*`,` *symbol2*`, ...,` *symbolN*
The `.weak` directive declares each *symbol* in the argument list to be defined either externally or in the input file and accessible to other files. Default bindings of the symbol are overridden by the `.weak` directive. A *weak* symbol definition in one file satisfies an undefined reference to a global symbol of the same name in another file. Unresolved *weak* symbols have a default value of zero. The link editor does not resolve these symbols. If a *weak* symbol has the same name as a defined *global* symbol, the weak symbol is ignored and no error results. The `.weak` directive does not define the symbol.

`.zero` *expression*
While filling a data section, the `.zero` directive fills the number of bytes specified by *expression* with zero (0).

◆ ◆ ◆ CHAPTER 3

Instruction Set Mapping

This chapter provides a general mapping between the Solaris x86 assembly language mnemonics and the Intel or Advanced Micro Devices (AMD) mnemonics.

Instruction Overview

It is beyond the scope of this manual to document the x86 architecture instruction set. This chapter provides a general mapping between the Solaris x86 assembly language mnemonics and the Intel or AMD mnemonics to enable you to refer to your vendor's documentation for detailed information about a specific instruction. Instructions are grouped by functionality in tables with the following sections:

- Solaris mnemonic
- Intel/AMD mnemonic
- Description (short)
- Notes

For certain Solaris mnemonics, the allowed data type suffixes for that mnemonic are indicated in braces ({}) following the mnemonic. For example, `bswap{lq}` indicates that the following mnemonics are valid: `bswap`, `bswapl` (which is the default and equivalent to `bswap`), and `bswapq`. See "Instructions" on page 17 for information on data type suffixes.

To locate a specific Solaris x86 mnemonic, look up the mnemonic in the index.

General-Purpose Instructions

The general-purpose instructions perform basic data movement, memory addressing, arithmetic and logical operations, program flow control, input/output, and string operations on integer, pointer, and BCD data types.

Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers, and perform operations such as conditional moves, stack access, and data conversion.

TABLE 3–1 Data Transfer Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`bswap{lq}`	`BSWAP`	byte swap	`bswapq` valid only under `-xarch=amd64`
`cbtw`	`CBW`	convert byte to word	
`cltd`	`CDQ`	convert doubleword to quadword	`%eax` → `%edx:%eax`
`cltq`	`CDQE`	convert doubleword to quadword	`%eax` → `%rax` `cltq` valid only under `-xarch=amd64`
`cmova{wlq}`, `cmov{wlq}.a`	`CMOVA`	conditional move if above	`cmovaq` valid only under `-xarch=amd64`
`cmovae{wlq}`, `cmov{wlq}.ae`	`CMOVAE`	conditional move if above or equal	`cmovaeq` valid only under `-xarch=amd64`
`cmovb{wlq}`, `cmov{wlq}.b`	`CMOVB`	conditional move if below	`cmovbq` valid only under `-xarch=amd64`
`cmovbe{wlq}`, `cmov{wlq}.be`	`CMOVBE`	conditional move if below or equal	`cmovbeq` valid only under `-xarch=amd64`
`cmovc{wlq}`, `cmov{wlq}.c`	`CMOVC`	conditional move if carry	`cmovcq` valid only under `-xarch=amd64`
`cmove{wlq}`, `cmov{wlq}.e`	`CMOVE`	conditional move if equal	`cmoveq` valid only under `-xarch=amd64`
`cmovg{wlq}`, `cmov{wlq}.g`	`CMOVG`	conditional move if greater	`cmovgq` valid only under `-xarch=amd64`

TABLE 3–1 Data Transfer Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cmovge{wlq}`, `cmov{wlq}.ge`	`CMOVGE`	conditional move if greater or equal	`cmovgeq` valid only under `-xarch=amd64`
`cmovl{wlq}`, `cmov{wlq}.l`	`CMOVL`	conditional move if less	`cmovlq` valid only under `-xarch=amd64`
`cmovle{wlq}`, `cmov{wlq}.le`	`COMVLE`	conditional move if less or equal	`cmovleq` valid only under `-xarch=amd64`
`cmovna{wlq}`, `cmov{wlq}.na`	`CMOVNA`	conditional move if not above	`cmovnaq` valid only under `-xarch=amd64`
`cmovnae{wlq}`, `cmov{wlq}.nae`	`CMOVNAE`	conditional move if not above or equal	`cmovnaeq` valid only under `-xarch=amd64`
`cmovnb{wlq}`, `cmov{wlq}.nb`	`CMOVNB`	conditional move if not below	`cmovnbq` valid only under `-xarch=amd64`
`cmovnbe{wlq}`, `cmov{wlq}.nbe`	`CMOVNBE`	conditional move if not below or equal	`cmovnbeq` valid only under `-xarch=amd64`
`cmovnc{wlq}`, `cmov{wlq}.nc`	`CMOVNC`	conditional move if not carry	`cmovncq` valid only under `-xarch=amd64`
`cmovne{wlq}`, `cmov{wlq}.ne`	`CMOVNE`	conditional move if not equal	`cmovneq` valid only under `-xarch=amd64`
`cmovng{wlq}`, `cmov{wlq}.ng`	`CMOVNG`	conditional move if greater	`cmovngq` valid only under `-xarch=amd64`
`cmovnge{wlq}`, `cmov{wlq}.nge`	`CMOVNGE`	conditional move if not greater or equal	`cmovngeq` valid only under `-xarch=amd64`
`cmovnl{wlq}`, `cmov{wlq}.nl`	`CMOVNL`	conditional move if not less	`cmovnlq` valid only under `-xarch=amd64`
`cmovnle{wlq}`, `cmov{wlq}.nle`	`CMOVNLE`	conditional move if not above or equal	`cmovnleq` valid only under `-xarch=amd64`
`cmovno{wlq}`, `cmov{wlq}.no`	`CMOVNO`	conditional move if not overflow	`cmovnoq` valid only under `-xarch=amd64`
`cmovnp{wlq}`, `cmov{wlq}.np`	`CMOVNP`	conditional move if not parity	`cmovnpq` valid only under `-xarch=amd64`
`cmovns{wlq}`, `cmov{wlq}.ns`	`CMOVNS`	conditional move if not sign (non-negative)	`cmovnsq` valid only under `-xarch=amd64`
`cmovnz{wlq}`, `cmov{wlq}.nz`	`CMOVNZ`	conditional move if not zero	`cmovnzq` valid only under `-xarch=amd64`

TABLE 3–1 Data Transfer Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cmovo{wlq}`, `cmov{wlq}.o`	`CMOVO`	conditional move if overflow	`cmovoq` valid only under `-xarch=amd64`
`cmovp{wlq}`, `cmov{wlq}.p`	`CMOVP`	conditional move if parity	`cmovpq` valid only under `-xarch=amd64`
`cmovpe{wlq}`, `cmov{wlq}.pe`	`CMOVPE`	conditional move if parity even	`cmovpeq` valid only under `-xarch=amd64`
`cmovpo{wlq}`, `cmov{wlq}.po`	`CMOVPO`	conditional move if parity odd	`cmovpoq` valid only under `-xarch=amd64`
`cmovs{wlq}`, `cmov{wlq}.s`	`CMOVS`	conditional move if sign (negative)	`cmovsq` valid only under `-xarch=amd64`
`cmovz{wlq}`, `cmov{wlq}.z`	`CMOVZ`	conditional move if zero	`cmovzq` valid only under `-xarch=amd64`
`cmpxchg{bwlq}`	`CMPXCHG`	compare and exchange	`cmpxchgq` valid only under `-xarch=amd64`
`cmpxchg8b`	`CMPXCHG8B`	compare and exchange 8 bytes	
`cqtd`	`CQO`	convert quadword to octword	`%rax` → `%rdx:%rax` `cqtd` valid only under `-xarch=amd64`
`cqto`	`CQO`	convert quadword to octword	`%rax` → `%rdx:%rax` `cqto` valid only under `-xarch=amd64`
`cwtd`	`CWD`	convert word to doubleword	`%ax` → `%dx:%ax`
`cwtl`	`CWDE`	convert word to doubleword in `%eax` register	`%ax` → `%eax`
`mov{bwlq}`	`MOV`	move data between immediate values, general purpose registers, segment registers, and memory	`movq` valid only under `-xarch=amd64`
`movabs{bwlq}`	`MOVABS`	move immediate value to register	`movabs` valid only under `-xarch=amd64`

TABLE 3–1 Data Transfer Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`movabs{bwlq}A`	`MOVABS`	move immediate value to register {AL, AX, GAX, RAX}	`movabs` valid only under `-xarch=amd64`
`movsb{wlq}, movsw{lq}`	`MOVSX`	move and sign extend	`movsbq` and `movswq` valid only under `-xarch=amd64`
`movzb{wlq}, movzw{lq}`	`MOVZX`	move and zero extend	`movzbq` and `movzwq` valid only under `-xarch=amd64`
`pop{wlq}`	`POP`	pop stack	`popq` valid only under `-xarch=amd64`
`popaw`	`POPA`	pop general-purpose registers from stack	`popaw` invalid under `-xarch=amd64`
`popal, popa`	`POPAD`	pop general-purpose registers from stack	invalid under `-xarch=amd64`
`push{wlq}`	`PUSH`	push onto stack	`pushq` valid only under `-xarch=amd64`
`pushaw`	`PUSHA`	push general-purpose registers onto stack	`pushaw` invalid under `-xarch=amd64`
`pushal, pusha`	`PUSHAD`	push general-purpose registers onto stack	invalid under `-xarch=amd64`
`xadd{bwlq}`	`XADD`	exchange and add	`xaddq` valid only under `-xarch=amd64`
`xchg{bwlq}`	`XCHG`	exchange	`xchgq` valid only under `-xarch=amd64`
`xchg{bwlq}A`	`XCHG`	exchange	`xchgqA` valid only under `-xarch=amd64`

Binary Arithmetic Instructions

The binary arithmetic instructions perform basic integer computions on operands in memory or the general-purpose registers.

TABLE 3–2 Binary Arithmetic Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`adc{bwlq}`	`ADC`	add with carry	`adcq` valid only under `-xarch=amd64`

TABLE 3–2 Binary Arithmetic Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`add{bwlq}`	`ADD`	integer add	`addq` valid only under `-xarch=amd64`
`cmp{bwlq}`	`CMP`	compare	`cmpq` valid only under `-xarch=amd64`
`dec{bwlq}`	`DEC`	decrement	`decq` valid only under `-xarch=amd64`
`div{bwlq}`	`DIV`	divide (unsigned)	`divq` valid only under `-xarch=amd64`
`idiv{bwlq}`	`IDIV`	divide (signed)	`idivq` valid only under `-xarch=amd64`
`imul{bwlq}`	`IMUL`	multiply (signed)	`imulq` valid only under `-xarch=amd64`
`inc{bwlq}`	`INC`	increment	`incq` valid only under `-xarch=amd64`
`mul{bwlq}`	`MUL`	multiply (unsigned)	`mulq` valid only under `-xarch=amd64`
`neg{bwlq}`	`NEG`	negate	`negq` valid only under `-xarch=amd64`
`sbb{bwlq}`	`SBB`	subtract with borrow	`sbbq` valid only under `-xarch=amd64`
`sub{bwlq}`	`SUB`	subtract	`subq` valid only under `-xarch=amd64`

Decimal Arithmetic Instructions

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

TABLE 3–3 Decimal Arithmetic Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`aaa`	`AAA`	ASCII adjust after addition	invalid under `-xarch=amd64`
`aad`	`AAD`	ASCII adjust before division	invalid under `-xarch=amd64`

TABLE 3–3 Decimal Arithmetic Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`aam`	`AAM`	ASCII adjust after multiplication	invalid under `-xarch=amd64`
`aas`	`AAS`	ASCII adjust after subtraction	invalid under `-xarch=amd64`
`daa`	`DAA`	decimal adjust after addition	invalid under `-xarch=amd64`
`das`	`DAS`	decimal adjust after subtraction	invalid under `-xarch=amd64`

Logical Instructions

The logical instructions perform basic logical operations on their operands.

TABLE 3–4 Logical Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`and{bwlq}`	`AND`	bitwise logical AND	`andq` valid only under `-xarch=amd64`
`not{bwlq}`	`NOT`	bitwise logical NOT	`notq` valid only under `-xarch=amd64`
`or{bwlq}`	`OR`	bitwise logical OR	`orq` valid only under `-xarch=amd64`
`xor{bwlq}`	`XOR`	bitwise logical exclusive OR	`xorq` valid only under `-xarch=amd64`

Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in their operands.

TABLE 3–5 Shift and Rotate Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`rcl{bwlq}`	`RCL`	rotate through carry left	`rclq` valid only under `-xarch=amd64`
`rcr{bwlq}`	`RCR`	rotate through carry right	`rcrq` valid only under `-xarch=amd64`

TABLE 3–5 Shift and Rotate Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`rol{bwlq}`	ROL	rotate left	`rolq` valid only under `-xarch=amd64`
`ror{bwlq}`	ROR	rotate right	`rorq` valid only under `-xarch=amd64`
`sal{bwlq}`	SAL	shift arithmetic left	`salq` valid only under `-xarch=amd64`
`sar{bwlq}`	SAR	shift arithmetic right	`sarq` valid only under `-xarch=amd64`
`shl{bwlq}`	SHL	shift logical left	`shlq` valid only under `-xarch=amd64`
`shld{bwlq}`	SHLD	shift left double	`shldq` valid only under `-xarch=amd64`
`shr{bwlq}`	SHR	shift logical right	`shrq` valid only under `-xarch=amd64`
`shrd{bwlq}`	SHRD	shift right double	`shrdq` valid only under `-xarch=amd64`

Bit and Byte Instructions

The bit instructions test and modify individual bits in operands. The byte instructions set the value of a byte operand to indicate the status of flags in the `%eflags` register.

TABLE 3–6 Bit and Byte Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`bsf{wlq}`	BSF	bit scan forward	`bsfq` valid only under `-xarch=amd64`
`bsr{wlq}`	BSR	bit scan reverse	`bsrq` valid only under `-xarch=amd64`
`bt{wlq}`	BT	bit test	`btq` valid only under `-xarch=amd64`
`btc{wlq}`	BTC	bit test and complement	`btcq` valid only under `-xarch=amd64`
`btr{wlq}`	BTR	bit test and reset	`btrq` valid only under `-xarch=amd64`

TABLE 3–6 Bit and Byte Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`bts{wlq}`	`BTS`	bit test and set	`btsq` valid only under `-xarch=amd64`
`seta`	`SETA`	set byte if above	
`setae`	`SETAE`	set byte if above or equal	
`setb`	`SETB`	set byte if below	
`setbe`	`SETBE`	set byte if below or equal	
`setc`	`SETC`	set byte if carry	
`sete`	`SETE`	set byte if equal	
`setg`	`SETG`	set byte if greater	
`setge`	`SETGE`	set byte if greater or equal	
`setl`	`SETL`	set byte if less	
`setle`	`SETLE`	set byte if less or equal	
`setna`	`SETNA`	set byte if not above	
`setnae`	`SETNAE`	set byte if not above or equal	
`setnb`	`SETNB`	set byte if not below	
`setnbe`	`SETNBE`	set byte if not below or equal	
`setnc`	`SETNC`	set byte if not carry	
`setne`	`SETNE`	set byte if not equal	
`setng`	`SETNG`	set byte if not greater	
`setnge`	`SETNGE`	set byte if not greater or equal	
`setnl`	`SETNL`	set byte if not less	
`setnle`	`SETNLE`	set byte if not less or equal	
`setno`	`SETNO`	set byte if not overflow	
`setnp`	`SETNP`	set byte if not parity	
`setns`	`SETNS`	set byte if not sign (non-negative)	
`setnz`	`SETNZ`	set byte if not zero	

TABLE 3–6 Bit and Byte Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`seto`	`SETO`	set byte if overflow	
`setp`	`SETP`	set byte if parity	
`setpe`	`SETPE`	set byte if parity even	
`setpo`	`SETPO`	set byte if parity odd	
`sets`	`SETS`	set byte if sign (negative)	
`setz`	`SETZ`	set byte if zero	
`test{bwlq}`	`TEST`	logical compare	`testq` valid only under `-xarch=amd64`

Control Transfer Instructions

The control transfer instructions control the flow of program execution.

TABLE 3–7 Control Transfer Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`bound{wl}`	`BOUND`	detect value out of range	`boundw` invalid under `-xarch=amd64`
`call`	`CALL`	call procedure	
`enter`	`ENTER`	high-level procedure entry	
`int`	`INT`	software interrupt	
`into`	`INTO`	interrupt on overflow	invalid under `-xarch=amd64`
`iret`	`IRET`	return from interrupt	
`ja`	`JA`	jump if above	
`jae`	`JAE`	jump if above or equal	
`jb`	`JB`	jump if below	
`jbe`	`JBE`	jump if below or equal	
`jc`	`JC`	jump if carry	
`jcxz`	`JCXZ`	jump register `%cx` zero	
`je`	`JE`	jump if equal	

TABLE 3–7 Control Transfer Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`jecxz`	`JECXZ`	jump register `%ecx` zero	invalid under `-xarch=amd64`
`jg`	`JG`	jump if greater	
`jge`	`JGE`	jump if greater or equal	
`jl`	`JL`	jump if less	
`jle`	`JLE`	jump if less or equal	
`jmp`	`JMP`	jump	
`jnae`	`JNAE`	jump if not above or equal	
`jnb`	`JNB`	jump if not below	
`jnbe`	`JNBE`	jump if not below or equal	
`jnc`	`JNC`	jump if not carry	
`jne`	`JNE`	jump if not equal	
`jng`	`JNG`	jump if not greater	
`jnge`	`JNGE`	jump if not greater or equal	
`jnl`	`JNL`	jump if not less	
`jnle`	`JNLE`	jump if not less or equal	
`jno`	`JNO`	jump if not overflow	
`jnp`	`JNP`	jump if not parity	
`jns`	`JNS`	jump if not sign (non-negative)	
`jnz`	`JNZ`	jump if not zero	
`jo`	`JO`	jump if overflow	
`jp`	`JP`	jump if parity	
`jpe`	`JPE`	jump if parity even	
`jpo`	`JPO`	jump if parity odd	
`js`	`JS`	jump if sign (negative)	
`jz`	`JZ`	jump if zero	

TABLE 3–7 Control Transfer Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`lcall`	`CALL`	call far procedure	valid as indirect only for `-xarg=amd64`
`leave`	`LEAVE`	high-level procedure exit	
`loop`	`LOOP`	loop with `%ecx` counter	
`loope`	`LOOPE`	loop with `%ecx` and equal	
`loopne`	`LOOPNE`	loop with `%ecx` and not equal	
`loopnz`	`LOOPNZ`	loop with `%ecx` and not zero	
`loopz`	`LOOPZ`	loop with `%ecx` and zero	
`lret`	`RET`	return from far procedure	valid as indirect only for `-xarg=amd64`
`ret`	`RET`	return	

String Instructions

The string instructions operate on strings of bytes. Operations include storing strings in memory, loading strings from memory, comparing strings, and scanning strings for substrings.

Note – The Solaris mnemonics for certain instructions differ slightly from the Intel/AMD mnemonics. Alphabetization of the table below is by the Solaris mnemonic. All string operations default to long (doubleword).

TABLE 3–8 String Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cmps{q}`	`CMPS`	compare string	`cmpsq` valid only under `-xarch=amd64`
`cmpsb`	`CMPSB`	compare byte string	
`cmpsl`	`CMPSD`	compare doubleword string	
`cmpsw`	`CMPSW`	compare word string	
`lods{q}`	`LODS`	load string	`lodsq` valid only under `-xarch=amd64`

TABLE 3–8 String Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`lodsb`	`LODSB`	load byte string	
`lodsl`	`LODSD`	load doubleword string	
`lodsw`	`LODSW`	load word string	
`movs{q}`	`MOVS`	move string	`movsq` valid only under `-xarch=amd64`
`movsb`	`MOVSB`	move byte string	`movsb` is not `movsb{wlq}`. See Table 3–1
`movsl`, `smovl`	`MOVSD`	move doubleword string	
`movsw`, `smovw`	`MOVSW`	move word string	`movsw` is not `movsw{lq}`. See Table 3–1
`rep`	`REP`	repeat while `%ecx` not zero	
`repnz`	`REPNE`	repeat while not equal	
`repnz`	`REPNZ`	repeat while not zero	
`repz`	`REPE`	repeat while equal	
`repz`	`REPZ`	repeat while zero	
`scas{q}`	`SCAS`	scan string	`scasq` valid only under `-xarch=amd64`
`scasb`	`SCASB`	scan byte string	
`scasl`	`SCASD`	scan doubleword string	
`scasw`	`SCASW`	scan word string	
`stos{q}`	`STOS`	store string	`stosq` valid only under `-xarch=amd64`
`stosb`	`STOSB`	store byte string	
`stosl`	`STOSD`	store doubleword string	
`stosw`	`STOSW`	store word string	

I/O Instructions

The input/output instructions transfer data between the processor's I/O ports, registers, and memory.

TABLE 3–9 I/O Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`in`	`IN`	read from a port	
`ins`	`INS`	input string from a port	
`insb`	`INSB`	input byte string from port	
`insl`	`INSD`	input doubleword string from port	
`insw`	`INSW`	input word string from port	
`out`	`OUT`	write to a port	
`outs`	`OUTS`	output string to port	
`outsb`	`OUTSB`	output byte string to port	
`outsl`	`OUTSD`	output doubleword string to port	
`outsw`	`OUTSW`	output word string to port	

Flag Control (EFLAG) Instructions

The status flag control instructions operate on the bits in the `%eflags` register.

TABLE 3–10 Flag Control Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`clc`	`CLC`	clear carry flag	
`cld`	`CLD`	clear direction flag	
`cli`	`CLI`	clear interrupt flag	
`cmc`	`CMC`	complement carry flag	
`lahf`	`LAHF`	load flags into `%ah` register	
`popfw`	`POPF`	pop `%eflags` from stack	
`popf{lq}`	`POPFL`	pop `%eflags` from stack	`popfq` valid only under `-xarch=amd64`
`pushfw`	`PUSHF`	push `%eflags` onto stack	

TABLE 3–10 Flag Control Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`pushf{lq}`	`PUSHFL`	push `%eflags` onto stack	`pushfq` valid only under `-xarch=amd64`
`sahf`	`SAHF`	store `%ah` register into flags	
`stc`	`STC`	set carry flag	
`std`	`STD`	set direction flag	
`sti`	`STI`	set interrupt flag	

Segment Register Instructions

The segment register instructions load far pointers (segment addresses) into the segment registers.

TABLE 3–11 Segment Register Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`lds{wl}`	`LDS`	load far pointer using `%ds`	`ldsl` and `ldsw` invalid under `-xarch=amd64`
`les{wl}`	`LES`	load far pointer using `%es`	`lesl` and `lesw` invalid under `-xarch=amd64`
`lfs{wl}`	`LFS`	load far pointer using `%fs`	
`lgs{wl}`	`LGS`	load far pointer using `%gs`	
`lss{wl}`	`LSS`	load far pointer using `%ss`	

Miscellaneous Instructions

The instructions documented in this section provide a number of useful functions.

TABLE 3–12 Miscellaneous Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cpuid`	`CPUID`	processor identification	
`lea{wlq}`	`LEA`	load effective address	`leaq` valid only under `-xarch=amd64`
`nop`	`NOP`	no operation	

TABLE 3–12 Miscellaneous Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
ud2	UD2	undefined instruction	
xlat	XLAT	table lookup translation	
xlatb	XLATB	table lookup translation	

Floating-Point Instructions

The floating point instructions operate on floating-point, integer, and binary coded decimal (BCD) operands.

Data Transfer Instructions (Floating Point)

The data transfer instructions move floating-point, integer, and BCD values between memory and the floating point registers.

TABLE 3–13 Data Transfer Instructions (Floating-Point)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
fbld	FBLD	load BCD	
fbstp	FBSTP	store BCD and pop	
fcmovb	FCMOVB	floating-point conditional move if below	
fcmovbe	FCMOVBE	floating-point conditional move if below or equal	
fcmove	FCMOVE	floating-point conditional move if equal	
fcmovnb	FCMOVNB	floating-point conditional move if not below	
fcmovnbe	FCMOVNBE	floating-point conditional move if not below or equal	
fcmovne	FCMOVNE	floating-point conditional move if not equal	
fcmovnu	FCMOVNU	floating-point conditional move if unordered	

TABLE 3–13 Data Transfer Instructions (Floating-Point) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
fcmovu	FCMOVU	floating-point conditional move if unordered	
fild	FILD	load integer	
fist	FIST	store integer	
fistp	FISTP	store integer and pop	
fld	FLD	load floating-point value	
fst	FST	store floating-point value	
fstp	FSTP	store floating-point value and pop	
fxch	FXCH	exchange registers	

Basic Arithmetic Instructions (Floating-Point)

The basic arithmetic instructions perform basic arithmetic operations on floating-point and integer operands.

TABLE 3–14 Basic Arithmetic Instructions (Floating-Point)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
fabs	FABS	absolute value	
fadd	FADD	add floating-point	
faddp	FADDP	add floating-point and pop	
fchs	FCHS	change sign	
fdiv	FDIV	divide floating-point	
fdivp	FDIVP	divide floating-point and pop	
fdivr	FDIVR	divide floating-point reverse	
fdivrp	FDIVRP	divide floating-point reverse and pop	
fiadd	FIADD	add integer	
fidiv	FIDIV	divide integer	

TABLE 3–14 Basic Arithmetic Instructions (Floating-Point) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fidivr`	`FIDIVR`	divide integer reverse	
`fimul`	`FIMUL`	multiply integer	
`fisub`	`FISUB`	subtract integer	
`fisubr`	`FISUBR`	subtract integer reverse	
`fmul`	`FMUL`	multiply floating-point	
`fmulp`	`FMULP`	multiply floating-point and pop	
`fprem`	`FPREM`	partial remainder	
`fprem1`	`FPREM1`	IEEE partial remainder	
`frndint`	`FRNDINT`	round to integer	
`fscale`	`FSCALE`	scale by power of two	
`fsqrt`	`FSQRT`	square root	
`fsub`	`FSUB`	subtract floating-point	
`fsubp`	`FSUBP`	subtract floating-point and pop	
`fsubr`	`FSUBR`	subtract floating-point reverse	
`fsubrp`	`FSUBRP`	subtract floating-point reverse and pop	
`fxtract`	`FXTRACT`	extract exponent and significand	

Comparison Instructions (Floating-Point)

The floating-point comparison instructions operate on floating-point or integer operands.

TABLE 3–15 Comparison Instructions (Floating-Point)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fcom`	`FCOM`	compare floating-point	
`fcomi`	`FCOMI`	compare floating-point and set `%eflags`	

TABLE 3–15 Comparison Instructions (Floating-Point) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fcomip`	`FCOMIP`	compare floating-point, set `%eflags`, and pop	
`fcomp`	`FCOMP`	compare floating-point and pop	
`fcompp`	`FCOMPP`	compare floating-point and pop twice	
`ficom`	`FICOM`	compare integer	
`ficomp`	`FICOMP`	compare integer and pop	
`ftst`	`FTST`	test floating-point (compare with 0.0)	
`fucom`	`FUCOM`	unordered compare floating-point	
`fucomi`	`FUCOMI`	unordered compare floating-point and set `%eflags`	
`fucomip`	`FUCOMIP`	unordered compare floating-point, set `%eflags`, and pop	
`fucomp`	`FUCOMP`	unordered compare floating-point and pop	
`fucompp`	`FUCOMPP`	compare floating-point and pop twice	
`fxam`	`FXAM`	examine floating-point	

Transcendental Instructions (Floating-Point)

The transcendental instructions perform trigonometric and logarithmic operations on floating-point operands.

TABLE 3–16 Transcendental Instructions (Floating-Point)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`f2xm1`	`F2XM1`	computes 2^x-1	
`fcos`	`FCOS`	cosine	
`fpatan`	`FPATAN`	partial arctangent	

TABLE 3–16 Transcendental Instructions (Floating-Point) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fptan`	`FPTAN`	partial tangent	
`fsin`	`FSIN`	sine	
`fsincos`	`FSINCOS`	sine and cosine	
`fyl2x`	`FYL2X`	computes y * $\log_2 x$	
`fyl2xp1`	`FYL2XP1`	computes y * $\log_2(x+1)$	

Load Constants (Floating-Point) Instructions

The load constants instructions load common constants, such as π, into the floating-point registers.

TABLE 3–17 Load Constants Instructions (Floating-Point)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fld1`	`FLD1`	load +1.0	
`fldl2e`	`FLDL2E`	load $\log_2 e$	
`fldl2t`	`FLDL2T`	load $\log_2 10$	
`fldlg2`	`FLDLG2`	load $\log_{10} 2$	
`fldln2`	`FLDLN2`	load $\log_e 2$	
`fldpi`	`FLDPI`	load π	
`fldz`	`FLDZ`	load +0.0	

Control Instructions (Floating-Point)

The floating-point control instructions operate on the floating-point register stack and save and restore the floating-point state.

TABLE 3–18 Control Instructions (Floating-Point)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fclex`	`FCLEX`	clear floating-point exception flags after checking for error conditions	

TABLE 3–18 Control Instructions (Floating-Point) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fdecstp`	`FDECSTP`	decrement floating-point register stack pointer	
`ffree`	`FFREE`	free floating-point register	
`fincstp`	`FINCSTP`	increment floating-point register stack pointer	
`finit`	`FINIT`	initialize floating-point unit after checking error conditions	
`fldcw`	`FLDCW`	load floating-point unit control word	
`fldenv`	`FLDENV`	load floating-point unit environment	
`fnclex`	`FNCLEX`	clear floating-point exception flags without checking for error conditions	
`fninit`	`FNINIT`	initialize floating-point unit without checking error conditions	
`fnop`	`FNOP`	floating-point no operation	
`fnsave`	`FNSAVE`	save floating-point unit state without checking error conditions	
`fnstcw`	`FNSTCW`	store floating-point unit control word without checking error conditions	
`fnstenv`	`FNSTENV`	store floating-point unit environment without checking error conditions	
`fnstsw`	`FNSTSW`	store floating-point unit status word without checking error conditions	
`frstor`	`FRSTOR`	restore floating-point unit state	

TABLE 3–18 Control Instructions (Floating-Point) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fsave`	`FSAVE`	save floating-point unit state after checking error conditions	
`fstcw`	`FSTCW`	store floating-point unit control word after checking error conditions	
`fstenv`	`FSTENV`	store floating-point unit environment after checking error conditions	
`fstsw`	`FSTSW`	store floating-point unit status word after checking error conditions	
`fwait`	`FWAIT`	wait for floating-point unit	
`wait`	`WAIT`	wait for floating-point unit	

SIMD State Management Instructions

The `fxsave` and `fxrstor` instructions save and restore the state of the floating-point unit and the MMX, XMM, and MXCSR registers.

TABLE 3–19 SIMD State Management Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`fxrstor`	`FXRSTOR`	restore floating-point unit and SIMD state	
`fxsave`	`FXSAVE`	save floating-point unit and SIMD state	

MMX Instructions

The MMX instructions enable x86 processors to perform single-instruction, multiple-data(SIMD) operations on packed byte, word, doubleword, or quadword integer operands contained in memory, in MMX registers, or in general-purpose registers.

Data Transfer Instructions (MMX)

The data transfer instructions move doubleword and quadword operands between MMX registers and between MMX registers and memory.

TABLE 3–20 Data Transfer Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`movd`	`MOVD`	move doubleword	`movdq` valid only under `-xarch=amd64`
`movq`	`MOVQ`	move quadword	valid only under `-xarch=amd64`

Conversion Instructions (MMX)

The conversion instructions pack and unpack bytes, words, and doublewords.

TABLE 3–21 Conversion Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`packssdw`	`PACKSSDW`	pack doublewords into words with signed saturation	
`packsswb`	`PACKSSWB`	pack words into bytes with signed saturation	
`packuswb`	`PACKUSWB`	pack words into bytes with unsigned saturation	
`punpckhbw`	`PUNPCKHBW`	unpack high-order bytes	
`punpckhdq`	`PUNPCKHDQ`	unpack high-order doublewords	
`punpckhwd`	`PUNPCKHWD`	unpack high-order words	
`punpcklbw`	`PUNPCKLBW`	unpack low-order bytes	
`punpckldq`	`PUNPCKLDQ`	unpack low-order doublewords	
`punpcklwd`	`PUNPCKLWD`	unpack low-order words	

Packed Arithmetic Instructions (MMX)

The packed arithmetic instructions perform packed integer arithmetic on packed byte, word, and doubleword integers.

TABLE 3–22 Packed Arithmetic Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
paddb	PADDB	add packed byte integers	
paddd	PADDD	add packed doubleword integers	
paddsb	PADDSB	add packed signed byte integers with signed saturation	
paddsw	PADDSW	add packed signed word integers with signed saturation	
paddusb	PADDUSB	add packed unsigned byte integers with unsigned saturation	
paddusw	PADDUSW	add packed unsigned word integers with unsigned saturation	
paddw	PADDW	add packed word integers	
pmaddwd	PMADDWD	multiply and add packed word integers	
pmulhw	PMULHW	multiply packed signed word integers and store high result	
pmullw	PMULLW	multiply packed signed word integers and store low result	
psubb	PSUBB	subtract packed byte integers	
psubd	PSUBD	subtract packed doubleword integers	
psubsb	PSUBSB	subtract packed signed byte integers with signed saturation	

TABLE 3–22 Packed Arithmetic Instructions (MMX) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
psubsw	PSUBSW	subtract packed signed word integers with signed saturation	
psubusb	PSUBUSB	subtract packed unsigned byte integers with unsigned saturation	
psubusw	PSUBUSW	subtract packed unsigned word integers with unsigned saturation	
psubw	PSUBW	subtract packed word integers	

Comparison Instructions (MMX)

The compare instructions compare packed bytes, words, or doublewords.

TABLE 3–23 Comparison Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pcmpeqb	PCMPEQB	compare packed bytes for equal	
pcmpeqd	PCMPEQD	compare packed doublewords for equal	
pcmpeqw	PCMPEQW	compare packed words for equal	
pcmpgtb	PCMPGTB	compare packed signed byte integers for greater than	
pcmpgtd	PCMPGTD	compare packed signed doubleword integers for greater than	
pcmpgtw	PCMPGTW	compare packed signed word integers for greater than	

Logical Instructions (MMX)

The logical instructions perform logical operations on quadword operands.

TABLE 3–24 Logical Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pand	PAND	bitwise logical AND	
pandn	PANDN	bitwise logical AND NOT	
por	POR	bitwise logical OR	
pxor	PXOR	bitwise logical XOR	

Shift and Rotate Instructions (MMX)

The shift and rotate instructions operate on packed bytes, words, doublewords, or quadwords in 64–bit operands.

TABLE 3–25 Shift and Rotate Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pslld	PSLLD	shift packed doublewords left logical	
psllq	PSLLQ	shift packed quadword left logical	
psllw	PSLLW	shift packed words left logical	
psrad	PSRAD	shift packed doublewords right arithmetic	
psraw	PSRAW	shift packed words right arithmetic	
psrld	PSRLD	shift packed doublewords right logical	
psrlq	PSRLQ	shift packed quadword right logical	
psrlw	PSRLW	shift packed words right logical	

State Management Instructions (MMX)

The emms (EMMS) instruction clears the MMX state from the MMX registers.

TABLE 3–26 State Management Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`emms`	`EMMS`	empty MMX state	

SSE Instructions

SSE instructions are an extension of the SIMD execution model introduced with the MMX technology. SSE instructions are divided into four subgroups:

- SIMD single-precision floating-point instructions that operate on the XMM registers
- MXSCR state management instructions
- 64–bit SIMD integer instructions that operate on the MMX registers
- Instructions that provide cache control, prefetch, and instruction ordering functionality

SIMD Single-Precision Floating-Point Instructions (SSE)

The SSE SIMD instructions operate on packed and scalar single-precision floating-point values located in the XMM registers or memory.

Data Transfer Instructions (SSE)

The SSE data transfer instructions move packed and scalar single-precision floating-point operands between XMM registers and between XMM registers and memory.

TABLE 3–27 Data Transfer Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`movaps`	`MOVAPS`	move four aligned packed single-precision floating-point values between XMM registers or memory	
`movhlps`	`MOVHLPS`	move two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register	

TABLE 3–27 Data Transfer Instructions (SSE) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`movhps`	`MOVHPS`	move two packed single-precision floating-point values to or from the high quadword of an XMM register or memory	
`movlhps`	`MOVLHPS`	move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register	
`movlps`	`MOVLPS`	move two packed single-precision floating-point values to or from the low quadword of an XMM register or memory	
`movmskps`	`MOVMSKPS`	extract sign mask from four packed single-precision floating-point values	
`movss`	`MOVSS`	move scalar single-precision floating-point value between XMM registers or memory	
`movups`	`MOVUPS`	move four unaligned packed single-precision floating-point values between XMM registers or memory	

Packed Arithmetic Instructions (SSE)

SSE packed arithmetic instructions perform packed and scalar arithmetic operations on packed and scalar single-precision floating-point operands.

TABLE 3–28 Packed Arithmetic Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
addps	ADDPS	add packed single-precision floating-point values	
addss	ADDSS	add scalar single-precision floating-point values	
divps	DIVPS	divide packed single-precision floating-point values	
divss	DIVSS	divide scalar single-precision floating-point values	
maxps	MAXPS	return maximum packed single-precision floating-point values	
maxss	MAXSS	return maximum scalar single-precision floating-point values	
minps	MINPS	return minimum packed single-precision floating-point values	
minss	MINSS	return minimum scalar single-precision floating-point values.	
mulps	MULPS	multiply packed single-precision floating-point values	
mulss	MULSS	multiply scalar single-precision floating-point values	
rcpps	RCPPS	compute reciprocals of packed single-precision floating-point values	
rcpss	RCPSS	compute reciprocal of scalar single-precision floating-point values	

TABLE 3–28 Packed Arithmetic Instructions (SSE) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
rsqrtps	RSQRTPS	compute reciprocals of square roots of packed single-precision floating-point values	
rsqrtss	RSQRTSS	compute reciprocal of square root of scalar single-precision floating-point values	
sqrtps	SQRTPS	compute square roots of packed single-precision floating-point values	
sqrtss	SQRTSS	compute square root of scalar single-precision floating-point values	
subps	SUBPS	subtract packed single-precision floating-point values	
subss	SUBSS	subtract scalar single-precision floating-point values	

Comparison Instructions (SSE)

The SEE compare instructions compare packed and scalar single-precision floating-point operands.

TABLE 3–29 Comparison Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
cmpps	CMPPS	compare packed single-precision floating-point values	
cmpss	CMPSS	compare scalar single-precision floating-point values	

TABLE 3–29 Comparison Instructions (SSE) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
comiss	COMISS	perform ordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register	
ucomiss	UCOMISS	perform unordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register	

Logical Instructions (SSE)

The SSE logical instructions perform bitwise AND, AND NOT, OR, and XOR operations on packed single-precision floating-point operands.

TABLE 3–30 Logical Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
andnps	ANDNPS	perform bitwise logical AND NOT of packed single-precision floating-point values	
andps	ANDPS	perform bitwise logical AND of packed single-precision floating-point values	
orps	ORPS	perform bitwise logical OR of packed single-precision floating-point values	
xorps	XORPS	perform bitwise logical XOR of packed single-precision floating-point values	

Shuffle and Unpack Instructions (SSE)

The SSE shuffle and unpack instructions shuffle or interleave single-precision floating-point values in packed single-precision floating-point operands.

TABLE 3–31 Shuffle and Unpack Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`shufps`	`SHUFPS`	shuffles values in packed single-precision floating-point operands	
`unpckhps`	`UNPCKHPS`	unpacks and interleaves the two high-order values from two single-precision floating-point operands	
`unpcklps`	`UNPCKLPS`	unpacks and interleaves the two low-order values from two single-precision floating-point operands	

Conversion Instructions (SSE)

The SSE conversion instructions convert packed and individual doubleword integers into packed and scalar single-precision floating-point values.

TABLE 3–32 Conversion Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cvtpi2ps`	`CVTPI2PS`	convert packed doubleword integers to packed single-precision floating-point values	
`cvtps2pi`	`CVTPS2PI`	convert packed single-precision floating-point values to packed doubleword integers	
`cvtsi2ss`	`CVTSI2SS`	convert doubleword integer to scalar single-precision floating-point value	
`cvtss2si`	`CVTSS2SI`	convert scalar single-precision floating-point value to a doubleword integer	

TABLE 3–32 Conversion Instructions (SSE) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cvttps2pi`	`CVTTPS2PI`	convert with truncation packed single-precision floating-point values to packed doubleword integers	
`cvttss2si`	`CVTTSS2SI`	convert with truncation scalar single-precision floating-point value to scalar doubleword integer	

MXCSR State Management Instructions (SSE)

The MXCSR state management instructions save and restore the state of the MXCSR control and status register.

TABLE 3–33 MXCSR State Management Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`ldmxcsr`	`LDMXCSR`	load `%mxcsr` register	
`stmxcsr`	`STMXCSR`	save `%mxcsr` register state	

64–Bit SIMD Integer Instructions (SSE)

The SSE 64–bit SIMD integer instructions perform operations on packed bytes, words, or doublewords in MMX registers.

TABLE 3–34 64–Bit SIMD Integer Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`pavgb`	`PAVGB`	compute average of packed unsigned byte integers	
`pavgw`	`PAVGW`	compute average of packed unsigned byte integers	
`pextrw`	`PEXTRW`	extract word	
`pinsrw`	`PINSRW`	insert word	

TABLE 3–34 64–Bit SIMD Integer Instructions (SSE) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pmaxsw	PMAXSW	maximum of packed signed word integers	
pmaxub	PMAXUB	maximum of packed unsigned byte integers	
pminsw	PMINSW	minimum of packed signed word integers	
pminub	PMINUB	minimum of packed unsigned byte integers	
pmovmskb	PMOVMSKB	move byte mask	
pmulhuw	PMULHUW	multiply packed unsigned integers and store high result	
psadbw	PSADBW	compute sum of absolute differences	
pshufw	PSHUFW	shuffle packed integer word in MMX register	

Miscellaneous Instructions (SSE)

The following instructions control caching, prefetching, and instruction ordering.

TABLE 3–35 Miscellaneous Instructions (SSE)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
maskmovq	MASKMOVQ	non-temporal store of selected bytes from an MMX register into memory	
movntps	MOVNTPS	non-temporal store of four packed single-precision floating-point values from an XMM register into memory	
movntq	MOVNTQ	non-temporal store of quadword from an MMX register into memory	

TABLE 3–35 Miscellaneous Instructions (SSE) *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`prefetchnta`	`PREFETCHNTA`	prefetch data into non-temporal cache structure and into a location close to the processor	
`prefetcht0`	`PREFETCHT0`	prefetch data into all levels of the cache hierarchy	
`prefetcht1`	`PREFETCHT1`	prefetch data into level 2 cache and higher	
`prefetcht2`	`PREFETCHT2`	prefetch data into level 2 cache and higher	
`sfence`	`SFENCE`	serialize store operations	

SSE2 Instructions

SSE2 instructions are an extension of the SIMD execution model introduced with the MMX technology and the SSE extensions. SSE2 instructions are divided into four subgroups:

- Packed and scalar double-precision floating-point instructions
- Packed single-precision floating-point conversion instructions
- 128–bit SIMD integer instructions
- Instructions that provide cache control and instruction ordering functionality

SSE2 Packed and Scalar Double-Precision Floating-Point Instructions

The SSE2 packed and scalar double-precision floating-point instructions operate on double-precision floating-point operands.

SSE2 Data Movement Instructions

The SSE2 data movement instructions move double-precision floating-point data between XMM registers and memory.

TABLE 3–36 SSE2 Data Movement Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`movapd`	`MOVAPD`	move two aligned packed double-precision floating-point values between XMM registers and memory	
`movhpd`	`MOVHPD`	move high packed double-precision floating-point value to or from the high quadword of an XMM register and memory	
`movlpd`	`MOVLPD`	move low packed single-precision floating-point value to or from the low quadword of an XMM register and memory	
`movmskpd`	`MOVMSKPD`	extract sign mask from two packed double-precision floating-point values	
`movsd`	`MOVSD`	move scalar double-precision floating-point value between XMM registers and memory.	
`movupd`	`MOVUPD`	move two unaligned packed double-precision floating-point values between XMM registers and memory	

SSE2 Packed Arithmetic Instructions

The SSE2 arithmetic instructions operate on packed and scalar double-precision floating-point operands.

TABLE 3–37 SSE2 Packed Arithmetic Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
addpd	ADDPD	add packed double-precision floating-point values	
addsd	ADDSD	add scalar double-precision floating-point values	
divpd	DIVPD	divide packed double-precision floating-point values	
divsd	DIVSD	divide scalar double-precision floating-point values	
maxpd	MAXPD	return maximum packed double-precision floating-point values	
maxsd	MAXSD	return maximum scalar double-precision floating-point value	
minpd	MINPD	return minimum packed double-precision floating-point values	
minsd	MINSD	return minimum scalar double-precision floating-point value	
mulpd	MULPD	multiply packed double-precision floating-point values	
mulsd	MULSD	multiply scalar double-precision floating-point values	
sqrtpd	SQRTPD	compute packed square roots of packed double-precision floating-point values	
sqrtsd	SQRTSD	compute scalar square root of scalar double-precision floating-point value	

TABLE 3–37 SSE2 Packed Arithmetic Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
subpd	SUBPD	subtract packed double-precision floating-point values	
subsd	SUBSD	subtract scalar double-precision floating-point values	

SSE2 Logical Instructions

The SSE2 logical instructions operate on packed double-precision floating-point values.

TABLE 3–38 SSE2 Logical Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
andnpd	ANDNPD	perform bitwise logical AND NOT of packed double-precision floating-point values	
andpd	ANDPD	perform bitwise logical AND of packed double-precision floating-point values	
orpd	ORPD	perform bitwise logical OR of packed double-precision floating-point values	
xorpd	XORPD	perform bitwise logical XOR of packed double-precision floating-point values	

SSE2 Compare Instructions

The SSE2 compare instructions compare packed and scalar double-precision floating-point values and return the results of the comparison to either the destination operand or to the EFLAGS register.

TABLE 3–39 SSE2 Compare Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
cmppd	CMPPD	compare packed double-precision floating-point values	
cmpsd	CMPSD	compare scalar double-precision floating-point values	
comisd	COMISD	perform ordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register	
ucomisd	UCOMISD	perform unordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register	

SSE2 Shuffle and Unpack Instructions

The SSE2 shuffle and unpack instructions operate on packed double-precision floating-point operands.

TABLE 3–40 SSE2 Shuffle and Unpack Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
shufpd	SHUFPD	shuffle values in packed double-precision floating-point operands	
unpckhpd	UNPCKHPD	unpack and interleave the high values from two packed double-precision floating-point operands	
unpcklpd	UNPCKLPD	unpack and interleave the low values from two packed double-precision floating-point operands	

SSE2 Conversion Instructions

The SSE2 conversion instructions convert packed and individual doubleword integers into packed and scalar double-precision floating-point values (and vice versa). These instructions also convert between packed and scalar single-precision and double-precision floating-point values.

TABLE 3–41 SSE2 Conversion Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cvtdq2pd`	`CVTDQ2PD`	convert packed doubleword integers to packed double-precision floating-point values	
`cvtpd2dq`	`CVTPD2DQ`	convert packed double-precision floating-point values to packed doubleword integers	
`cvtpd2pi`	`CVTPD2PI`	convert packed double-precision floating-point values to packed doubleword integers	
`cvtpd2ps`	`CVTPD2PS`	convert packed double-precision floating-point values to packed single-precision floating-point values	
`cvtpi2pd`	`CVTPI2PD`	convert packed doubleword integers to packed double-precision floating-point values	
`cvtps2pd`	`CVTPS2PD`	convert packed single-precision floating-point values to packed double-precision floating-point values	
`cvtsd2si`	`CVTSD2SI`	convert scalar double-precision floating-point values to a doubleword integer	

TABLE 3–41 SSE2 Conversion Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`cvtsd2ss`	`CVTSD2SS`	convert scalar double-precision floating-point values to scalar single-precision floating-point values	
`cvtsi2sd`	`CVTSI2SD`	convert doubleword integer to scalar double-precision floating-point value	
`cvtss2sd`	`CVTSS2SD`	convert scalar single-precision floating-point values to scalar double-precision floating-point values	
`cvttpd2dq`	`CVTTPD2DQ`	convert with truncation packed double-precision floating-point values to packed doubleword integers	
`cvttpd2pi`	`CVTTPD2PI`	convert with truncation packed double-precision floating-point values to packed doubleword integers	
`cvttsd2si`	`CVTTSD2SI`	convert with truncation scalar double-precision floating-point values to scalar doubleword integers	

SSE2 Packed Single-Precision Floating-Point Instructions

The SSE2 packed single-precision floating-point instructions operate on single-precision floating-point and integer operands.

TABLE 3–42 SSE2 Packed Single-Precision Floating-Point Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
cvtdq2ps	CVTDQ2PS	convert packed doubleword integers to packed single-precision floating-point values	
cvtps2dq	CVTPS2DQ	convert packed single-precision floating-point values to packed doubleword integers	
cvttps2dq	CVTTPS2DQ	convert with truncation packed single-precision floating-point values to packed doubleword integers	

SSE2 128–Bit SIMD Integer Instructions

The SSE2 SIMD integer instructions operate on packed words, doublewords, and quadwords contained in XMM and MMX registers.

TABLE 3–43 SSE2 128–Bit SIMD Integer Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
movdq2q	MOVDQ2Q	move quadword integer from XMM to MMX registers	
movdqa	MOVDQA	move aligned double quadword	
movdqu	MOVDQU	move unaligned double quadword	
movq2dq	MOVQ2DQ	move quadword integer from MMX to XMM registers	
paddq	PADDQ	add packed quadword integers	
pmuludq	PMULUDQ	multiply packed unsigned doubleword integers	

TABLE 3–43 SSE2 128–Bit SIMD Integer Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pshufd	PSHUFD	shuffle packed doublewords	
pshufhw	PSHUFHW	shuffle packed high words	
pshuflw	PSHUFLW	shuffle packed low words	
pslldq	PSLLDQ	shift double quadword left logical	
psrldq	PSRLDQ	shift double quadword right logical	
psubq	PSUBQ	subtract packed quadword integers	
punpckhqdq	PUNPCKHQDQ	unpack high quadwords	
punpcklqdq	PUNPCKLQDQ	unpack low quadwords	

SSE2 Miscellaneous Instructions

The SSE2 instructions described below provide additional functionality for caching non-temporal data when storing data from XMM registers to memory, and provide additional control of instruction ordering on store operations.

TABLE 3–44 SSE2 Miscellaneous Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
clflush	CLFLUSH	flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy	
lfence	LFENCE	serializes load operations	
maskmovdqu	MASKMOVDQU	non-temporal store of selected bytes from an XMM register into memory	
mfence	MFENCE	serializes load and store operations	

TABLE 3–44 SSE2 Miscellaneous Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`movntdq`	`MOVNTDQ`	non-temporal store of double quadword from an XMM register into memory	
`movnti`	`MOVNTI`	non-temporal store of a doubleword from a general-purpose register into memory	`movntiq` valid only under `-xarch=amd64`
`movntpd`	`MOVNTPD`	non-temporal store of two packed double-precision floating-point values from an XMM register into memory	
`pause`	`PAUSE`	improves the performance of spin-wait loops	

Operating System Support Instructions

The operating system support instructions provide functionality for process management, performance monitoring, debugging, and other systems tasks.

TABLE 3–45 Operating System Support Instructions

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`arpl`	`ARPL`	adjust requested privilege level	
`clts`	`CLTS`	clear the task-switched flag	
`hlt`	`HLT`	halt processor	
`invd`	`INVD`	invalidate cache, no writeback	
`invlpg`	`INVLPG`	invalidate TLB entry	
`lar`	`LAR`	load access rights	`larq` valid only under `-xarch=amd64`
`lgdt`	`LGDT`	load global descriptor table (GDT) register	

TABLE 3–45 Operating System Support Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`lidt`	`LIDT`	load interrupt descriptor table (IDT) register	
`lldt`	`LLDT`	load local descriptor table (LDT) register	
`lmsw`	`LMSW`	load machine status word	
`lock`	`LOCK`	lock bus	
`lsl`	`LSL`	load segment limit	`lslq` valid only under `-xarch=amd64`
`ltr`	`LTR`	load task register	
`rdmsr`	`RDMSR`	read model-specific register	
`rdpmc`	`RDPMC`	read performance monitoring counters	
`rdtsc`	`RDTSC`	read time stamp counter	
`rsm`	`RSM`	return from system management mode (SMM)	
`sgdt`	`SGDT`	store global descriptor table (GDT) register	
`sidt`	`SIDT`	store interrupt descriptor table (IDT) register	
`sldt`	`SLDT`	store local descriptor table (LDT) register	`sldtq` valid only under `-xarch=amd64`
`smsw`	`SMSW`	store machine status word	`smswq` valid only under `-xarch=amd64`
`str`	`STR`	store task register	`strq` valid only under `-xarch=amd64`
`sysenter`	`SYSENTER`	fast system call, transfers to a flat protected model kernel at CPL=0	
`sysexit`	`SYSEXIT`	fast system call, transfers to a flat protected mode kernal at CPL=3	
`verr`	`VERR`	verify segment for reading	

TABLE 3–45 Operating System Support Instructions *(Continued)*

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
`verw`	`VERW`	verify segment for writing	
`wbinvd`	`WBINVD`	invalidate cache, with writeback	
`wrmsr`	`WRMSR`	write model-specific register	

64–Bit AMD Opteron Considerations

To assemble code for the AMD Opteron CPU, invoke the assembler with the `-xarch=amd64` command line option. See the `as(1)` man page for additional information.

The following Solaris mnemonics are only valid when the `-xarch=amd64` command line option is specified:

```
adcq        cmovneq     leaq
addq        cmovngeq    lodsq
andq        cmovngq     lslq
bsfq        cmovnleq    movabs
bsrq        cmovnlq     movdq
bswapq      cmovnoq     movntiq
btcq        cmovnpq     movq
btq         cmovnsq     movsq
btrq        cmovnzq     movswq
btsq        cmovoq      movzwq
cltq        cmovpeq     mulq
cmovaeq     cmovpoq     negq
cmovaq      cmovpq      notq
cmovbeq     cmovsq      orq
cmovbq      cmovzq      popfq
cmovcq      cmpq        popq
cmoveq      cmpsq       pushfq
cmovgeq     cmpxchgq    pushq
cmovgq      cqtd        rclq
cmovleq     cqto        rcrq
cmovlq      decq        rolq
cmovnaeq    divq        rorq
cmovnaq     idivq       salq
cmovnbeq    imulq       sarq
cmovnbq     incq        sbbq
cmovncq     larq        scasq
```

```
shldq                   smswq                   xaddq
shlq                    stosq                   xchgq
shrdq                   strq                    xchgqA
shrq                    subq                    xorq
sldtq                   testq
```

The following Solaris mnemonics are *not* valid when the `-xarch=amd64` command line option is specified:

```
aaa                     daa                     lesw
aad                     das                     popa
aam                     into                    popaw
aas                     jecxz                   pusha
boundw                  ldsw                    pushaw
```

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z