Assembly Programming Manual

IN THIS DOCUMENT

The XMOS assembly language supports the formation of objects in the Executable and Linkable Format (ELF)[1] with DWARF 3[2] debugging information. Extensions to the ELF format are documented in the XMOS Application Binary Interface (see XM-000967-PC).

1 Lexical Conventions

There are six classes of tokens: symbol names, directives, constants, operators, instruction mnemonics and other separators. Blanks, tabs, formfeeds and comments are ignored except as they separate tokens.

1.1 Comments

The character # introduces a comment, which terminates with a newline. Comments do not occur within string literals.

1.2 Symbol Names

A symbol name begins with a letter or with one of the characters '.', '_' or '$', followed by an optional sequence of letters, digits, periods, underscores and dollar signs. Upper and lower case letters are different.

1.3 Directives

A directive begins with '.' followed by one or more letters. Directives instruct the assembler to perform some action (see §6).

[1] http://www.xmos.com/references/elf
[2] http://www.xmos.com/references/dwarf3

Publication Date: 2015/3/27 Document Number: X9432B

1.4 Constants

A constant is either an integer number, a character constant or a string literal.

- A binary integer is `0b` or `0B` followed by zero or more of the digits `01`.
- An octal integer is `0` followed by zero or more of the digits `01234567`.
- A decimal integer is a non-zero digit followed by zero or more of the digits `0123456789`.
- A hexadecimal integer is `0x` or `0X` followed by one or more of the digits and letters `0123456789abcdefABCDEF`.
- A character constant is a sequence of characters surrounded by single quotes.
- A string literal is a sequence of characters surrounded by double quotes.

The C escape sequences may be used to specify certain characters.

2 Sections and Relocations

Named ELF sections are specified using directives (see §6.13). In addition, there is a unique unnamed "absolute" section and a unique unnamed "undefined" section. The notation {*secname* X} refers to an "offset X into section *secname*."

The values of symbols in the absolute section are unaffected by relocations. For example, address {absolute 0} is "relocated" to run-time address 0. The values of symbols in the undefined section are not set.

The assembler keeps track of the current section. Initially the current section is set to the text section. Directives can be used to change the current section. Assembly instructions and directives which allocate storage are emitted in the current section. For each section, the assembler maintains a location counter which holds the current offset in the section. The *active location counter* refers to the location counter for the current section.

3 Symbols

Each symbol has exactly one name; each name in an assembly program refers to exactly one symbol. A local symbol is any symbol beginning with the characters ".L". A local symbol may be discarded by the linker when no longer required for linking.

3.1 Attributes

Each symbol has a *value*, an associated section and a *binding*. A symbol is assigned a value using the `set` or `linkset` directives (see §6.15), or through its use in a label (see §4). The default binding of symbols in the undefined section is *global*; for all other symbols the default binding is *local*.

4 Labels

A label is a symbol name immediately followed by a colon (:). The symbol's value is set to the current value of the active location counter. The symbol's section is set to the current section. A symbol name must not appear in more than one label.

5 Expressions

An expression specifies an address or value. The result of an expression must be an absolute number or an offset into a particular section. An expression is a *constant expression* if all of its symbols are defined and it evaluates to a constant. An expression is a simple expression if it is one of a constant expression, a symbol, or a symbol ± a constant. An expression may be encoded in the ELF-extended expression section and its value evaluated by the linker (see §6.15); the encoding scheme is determined by the ABI. The syntax of an expression is:

expression ::= *unary-exp*
| *expression infix-op unary-exp*
| *unary-exp* `?` *unary-exp* `$:` *unary-exp*
| *function-exp*

unary-exp ::= *argument*
| *prefix-op unary-exp*

argument ::= *symbol*
| *constant*
| `(` *expression* `)`

function-exp ::= `$overlay_region_ptr (` *symbol* `)`
| `$overlay_index (` *symbol* `)`
| `$overlay_physical_addr (` *symbol* `)`
| `$overlay_virtual_addr (` *symbol* `)`
| `$overlay_num_bytes (` *symbol* `)`

infix-op ::= *one of*
`+ - < > <= >= || << >> * $M $A & /`

prefix-op ::= *one of*
`- ~ $D`

Symbols are evaluated to {*section x*} where *section* is one of a named section, the absolute section or the undefined section, and *x* is a signed 2's complement 32-bit integer.

Infix operators have the same precedence and behavior as C, and operators with equal precedence are performed left to right. In addition, the `$M` operator has lowest precedence, and the `$A` operator has the highest precedence.

For the + and - operators, the set of valid operations and results is given in Figure 1. For the `$D` operator, the argument must be a symbol; the result is 1 if the symbol is defined and 0 otherwise.

Figure 1: Valid operations for + and - operators

Op	Left Operand	Right Operand	Result
+	{*section* x}	{absolute y}	{*section* x+y}
+	{absolute x}	{*section* y}	{*section* x+y}
+	{absolute x}	{absolute y}	{absolute x+y}
-	{*section* x}	{*section* y}	{absolute x-y}
-	{*section* x}	{absolute y}	{*section* x-y}
-	{absolute x}	{absolute y}	{absolute x-y}

The ? operator is used to select between symbols: if the first operand is non-zero then the result is the second operand, otherwise the result is the third operand.

The operators `$overlay_region_ptr`, `$overlay_index`, `$overlay_physical_addr`, `$overlay_virtual_addr` and `$overlay_num_bytes` can be used to query properties of the overlay containing the overlay roots with the specified overlay key symbol (see §6.21). The set of results of these operators is given in Figure 2.

Figure 2: Operators for querying properties of overlays.

Operator	Result
$overlay_region_ptr	Virtual address of the overlay region containing the overlay.
$overlay_index	Index of the overlay in the overlay region.
$overlay_physical_addr	Physical address of the overlay.
$overlay_virtual_addr	Virtual (runtime) address of the overlay.
$overlay_num_bytes	Size of the overlay in bytes.

For all other operators, both arguments must be absolute and the result is absolute. The `$M` operator returns the maximum of the two operands and the `$A` operator returns the value of the first operand aligned to the second.

Wherever an absolute expression is required, if omitted then {absolute 0} is assumed.

6 Directives

Directives instruct the assembler to perform some action. The supported directives are given in this section.

6.1 add_to_set

The `add_to_set` directive adds an expression to a set of expressions associated with a key symbol. Its syntax is:

add-to-set-directive ::= `.add_to_set` *symbol* , *expression*
| `.add_to_set` *symbol* , *expression* , *symbol*

An optional predicate symbol may be specified as the 3rd argument. If this argument is specified the expression will only be added to the set if the predicate symbol is not eliminated from the linked object.

6.2 max_reduce, sum_reduce

The `max_reduce` directive computes the maximum of the values of the expressions in a set. The `sum_reduce` directive computes the sum of the values of the expressions in a set.

max-reduce-directive ::= `.max_reduce` *symbol* , *symbol* , *expression*

sum-reduce-directive ::= `.sum_reduce` *symbol* , *symbol* , *expression*

The first symbol is defined using the value computed by the directive. The second symbol is the key symbol identifying the set of expressions (see § 6.1). The expression specifies the initial value for the reduction operation.

6.3 align

The `align` directive pads the active location counter section to the specified storage boundary. Its syntax is:

align-directive ::= `.align` *expression*

The expression must be a constant expression; its value must be a power of 2. This value specifies the alignment required in bytes.

6.4 ascii, asciiz

The `ascii` directive assembles each string into consecutive addresses. The `asciiz` directive is the same, except that each string is followed by a null byte.

ascii-directive ::= `.ascii` *string-list*
| `.asciiz` *string-list*

string-list ::= *string-list* , *string*
| `.asciiz` *string*

6.5 byte, short, int, long, word

These directives emit, for each expression, a number that at run-time is the value of that expression. The byte order is determined by the endianness of the target architecture. The size of numbers emitted with the word directive is determined by the size of the natural word on the target architecture. The size of the numbers emitted using the other directives are determined by the sizes of corresponding types in the ABI.

value-directive ::= *value-size exp-list*

value-size ::= `.byte`
| `.short`
| `.int`
| `.long`
| `.word`

exp-list ::= *exp-list* , *expression*
| *expression*

6.6 file

The `file` directive has two forms.

file-directive ::= `.file` *string*
| `.file` *constant string*

When used with one argument, the `file` directive creates an ELF symbol table entry with type `STT_FILE` and the specified string value. This entry is guaranteed to be the first entry in the symbol table.

When used with two arguments the `file` directive adds an entry to the DWARF 3 `.debug_line` file names table. The first argument is a unique positive integer to use as the index of the entry in the table. The second argument is the name of the file.

6.7 loc

The `.loc` directive adds a row to the DWARF 3 `.debug_line` line number matrix.

loc-directive ::= *constant constant constant*$_{\text{opt}}$
| *constant constant constant* ⟨*loc-option*⟩$*$

loc-option ::= `basic_block`
| `prologue_end`
| `epilogue_begin`
| `is_stmt` *constant*
| `isa` *constant*

The address register is set to active location counter. The first two arguments set the file and line registers respectively. The optional third argument sets the column register. Additional arguments set further registers in the `.debug_line` state machine.

`basic_block`
Sets *basic_block* to true.

`prologue_end`
Sets *prologue_end* to true.

`epilogue_begin`
Sets *epilogue_begin* to true.

`is_stmt`
Sets *is_stmt* to the specified value, which must be 0 or 1.

`isa`
Sets *isa* to the specified value.

6.8 weak

The `weak` directive sets the weak attribute on the specified symbol.

weak-directive ::= `.weak` *symbol*

6.9 globl, global, extern, locl, local

The `globl` directive makes the specified symbols visible to other objects during linking. The `extern` directive specifies that the symbol is defined in another object. The `locl` directive specifies a symbol has local binding.

visibility ::= `.globl`
| `.extern`
| `.locl`
| `.global`
| `.extern`
| `.local`

vis-directive ::= *visibility symbol*
| *visibility symbol* , *string*

If the optional string is provided, an `SHT_TYPEINFO` entry is created in the ELF-extended type section which contains the symbol and an index into the string table whose entry contains the specified string. (If the string does not already exist in the string table, it is inserted.) The meaning of this string is determined by the ABI.

The `global` and `local` directives are synonyms for the `globl` and `locl` directives. They are provided for compatibility with other assemblers.

6.10 globalresource

globalresource-directive ::= `.globalresource` *expression* , *string*
| `.globalresource` *expression* , *string* , *string*

The `globalresource` directive causes the assembler to add information to the binary to indicate that there was a global port or clock declaration. The first argument is the resource ID of the port. The second argument is the name of the variable. The optional third argument is the tile the port was declared on. For example:

```
.globalresource 0x10200, p, tile[0]
```

specifies that the port `p` was declared on `tile[0]` and initialized with the resource ID `0x10200`.

6.11 typestring

The `typestring` adds an `SHT_TYPEINFO` entry in the ELF-extended type section which contains the symbol and an index into the string table whose entry contains the specified string. (If the string does not already exist in the string table, it is inserted.) The meaning of this string is determined by the ABI.

typestring-directive ::= `.typestring` *symbol* , *string*

6.12 ident, core, corerev

Each of these directives creates an ELF note section named "`.xmos_note`."

info-directive ::= `.ident` *string*
| `.core` *string*
| `.corerev` *string*

The contents of this section is a (name, type, value) triplet: the name is `xmos`; the type is either `IDENT`, `CORE` or `COREREV`; and the value is the specified string.

6.13 section, pushsection, popsection

The section directives change the current ELF section (see §2).

section-directive ::= *sec-or-push name*
| *sec-or-push name* , *flags sec-type*$_{opt}$
| `.popsection`

sec-or-push ::= `.section`
| `.pushsection`

flags ::= *string*

sec-type ::= *type*
| *type* , *flag-args*

type ::= `@progbits`
| `@nobits`

flag-args ::= *string*

The code following a `section` or `pushsection` directive is assembled and appended to the named section. The optional flags may contain any combination of the following characters.

`a`	section is allocatable
`c`	section is placed in the global constant pool
`d`	section is placed in the global data region
`w`	section is writable
`x`	section is executable
`M`	section is mergeable
`S`	section contains zero terminated strings

The optional type argument `progbits` specifies that the section contains data; `nobits` specifies that it does not.

If the `M` symbol is specified as a flag, a type argument must be specified and an integer must be provided as a flag-specific argument. The flag-specific argument represents the entity size of data entries in the section. For example:

```
.section .cp.const4, "M", @progbits, 4
```

Sections with the `M` flag but not S flag must contain fixed-size constants, each *flag-args* bytes long. Sections with both the `M` and `S` flags must contain zero-terminated strings, each character *flag-args* bytes long. The linker may remove duplicates within sections with the same name, entity size and flags.

Each section with the same name must have the same type and flags. The `section` directive replaces the current section with the named section. The `pushsection` directive pushes the current section onto the top of a *section stack* and then replaces the current section with the named section. The `popsection` directive replaces the current section with the section on top of the section stack and then pops this section from the stack.

6.14 text

The `text` directive changes the current ELF section to the `.text` section. The section type and attributes are determined by the ABI.

text-directive ::= `.text`

6.15 set, linkset

A symbol is assigned a value using the `set` or `linkset` directive.

set-directive ::= *set-type symbol* , *expression*

set-type ::= `.set`
| `.linkset`

The `set` directive defines the named symbol with the value of the expression. The expression must be either a constant or a symbol: if the expression is a constant, the symbol is defined in the absolute section; if the expression is a symbol, the defined symbol inherits its section information and other attributes from this symbol.

The `linkset` directive is the same, except that the expression is not evaluated; instead one or more `SHT_EXPR` entries are created in the ELF-extended expression section which together form a tree representation of the expression.

Any symbol used in the assembly code may be a target of an `SHT_EXPR` entry, in which case its value is computed by the linker by evaluating the expression once values for all other symbols in the expression are known. This may happen at any incremental link stage; once the value is known, it is assigned to the symbol as with set and the expression entry is eliminated from the linked object.

6.16 cc_top, cc_bottom

The `cc_top` and `cc_bottom` directives are used to mark the beginning and end of elimination blocks.

cc-top-directive ::= `.cc_top` *name* , *predicate*
| `.cc_top` *name*

cc-directive ::= *cc-top-directive*
| `.cc_bottom` *name*

name ::= *symbol*

predicate ::= *symbol*

`cc_top` and `cc_bottom` directives with the same name refer to the same elimination block. An elimination block must have precisely one `cc_top` directive and one `cc_bottom` directive. The top and bottom of an elimination block must be in the same section. The elimination block consists of the data and labels in this section between the `cc_top` and `cc_bottom` directives. Elimination blocks must be disjoint; it is illegal for elimination blocks to overlap.

An elimination block is retained in final executable if one of the following is true:

- A label inside the elimination block is referenced from a location outside an elimination block.
- A label inside the elimination block is referenced from an elimination block which is not eliminated
- The predicate symbol is defined outside an elimination block or is contained in an elimination block which is not eliminated.

If none of these conditions are true the elimination block is removed from the final executable.

6.17 scheduling

The `scheduling` directive enables or disables instruction scheduling. When scheduling is enabled, the assembler may reorder instructions to minimize the number of FNOPs. The default scheduling mode is determined by the command-line option `-fschedule` (see XM-000927-PC).

scheduling-directive ::= `.scheduling` *scheduling-mode*

scheduling-mode ::= `on`
| `off`
| `default`

6.18 issue_mode

The `issue_mode` directive changes the current issue mode assumed by the assembler. See §7 for details of how the issue mode affects how instructions are assembled.

issue-mode-directive ::= `issue_mode` *issue-mode*

issue-mode ::= `single`
| `dual`

6.19 syntax

The `syntax` directive changes the current syntax mode. See §7 for details of how assembly instructions are specified in each mode.

syntax-directive ::= `.syntax` *syntax*

syntax ::= `default`
| `architectural`

6.20 assert

assert-directive ::= `.assert` *constant* , *symbol* , *string*

The `assert` directive requires an assertion to be tested prior to generating an executable object: the assertion fails if the symbol has a non-zero value. If the constant is 0, a failure should be reported as a warning; if the constant is 1, a failure should be reported as an error. The string is a message for an assembler or linker to emit on failure.

6.21 Overlay Directives

The overlay directives control how code and data is partitioned into overlays that are loaded on demand at runtime.

overlay-directive ::= `.overlay_reference` *symbol* , *symbol*
| `.overlay_root` *symbol* , *symbol*
| `.overlay_root` *symbol*
| `.overlay_subgraph_conflict` *sym-list*

sym-list ::= *sym-list* , *symbol*
| *symbol*

- The `overlay_root` directive specifies that the first symbol should be treated as an overlay root. The optional second symbols specifies a overlay key symbol. If no overlay key symbol is explictly specified the overlay root symbol is used as the key symbol. Specifying the same overlay key symbol for multiple overlay roots forces the overlay roots into the same overlay.

- The `overlay_reference` directive specifies that linker should assume that there is a reference from the first symbol to the second symbol when it partitions the program into overlays.
- The `overlay_subgraph_conflict` directive specifies that linker should not place any code or data reachable from one the symbols into an overlay that is mapped an overlay region that contains another overlay containing code or data reachable from one of the other symbols.

6.22 Language Directives

The language directives create entries in the ELF-extended expression section; the encoding is determined by the ABI.

xc-directive ::= *globdir symbol* , *string*
| *globdir symbol* , *symbol* , *range-args* , *string*
| `.globpassesref` *symbol* , *symbol* , *string*
| `.call` *symbol* , *symbol*
| `.par` *symbol* , *symbol* , *string*

range-args ::= *expression* , *expression*

globdir ::= `.globread`
| `.globwrite`
| `.parwrite`
| `.globpassesref`

For each directive, the string is an error message for the assembler or linker to display on encountering an error attributed to the directive.

`call`
: Both symbols must have function type. This directive sets the property that the first function may make a call to the second function.

`par`
: Both symbols must have function type. This directive sets the property that the first function is invoked in parallel with the second function.

`globread`
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the function may read the object. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is read and the second expression is the size of the read in bytes.

`globwrite`
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the function may write the object. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is written and the second expression is the size of the write in bytes.

`parwrite`
: The first symbol must have function type and the second directive must have object type. This directive set the property that the function is called in an expression which writes to the object where the order of evalulation of the write and the function call is undefined. When a range is specified, the first expression is the offset from the start of the variable in bytes of the address which is written and the second expression is the size of the write in bytes.

`globpassesref`
: The first symbol must have function type and the second directive must have object type. This directive sets the property that the object may be passed by reference to the function.

6.23 XMOS Timing Analyzer Directives

The XMOS Timing Analyzer directives add timing metadata to ELF sections.

xta-directive ::= `.xtabranch` *exp-list*$_{opt}$
| `.xtaendpoint` *string* , *source-location*
| `.xtacall` *string* , *source-location*
| `.xtalabel` *string* , *source-location*
| `.xtathreadstart`
| `.xtathreadstop`
| `.xtaloop` *constant*
| *.xtacommand string* , *source-location*

source-location ::= *string* , *string* , *constant*

The first string of a source location is the compilation directory. The second string is the path to the file. The path may be specified as either a relative path from the compilation directory or as an absolute path. The third argument is the line number.

- `xtabranch` specifies a comma-separated list of locations that may be branched to from the current location.
- `xtaendpoint` marks the current location as an endpoint with the specified label.
- `xtacall` marks the current location as a function call with the specified label.
- `xtalabel` marks the current location using the specified label.
- `xtathreadstart` apecifies that a thread may be initialized to start executing at the current location.
- `xtathreadstop` specifies that a thread executing the instruction at the current location will not execute any further instructions.
- `xtaloop` specifies that the innermost loop containing the current location executes the specified number of times.

- `xtacommand` specifies an XTA command to be executed when analyzing the executable.

6.24 uleb128, sleb128

The following directives emit, for each expression in the comma-separated list of expressions, a value that encodes either an unsigned or signed DWARF little-endian base 128 number.

leb-directive ::= `.uleb128` *exp-list*
| `.sleb128` *exp-list*

6.25 space, skip

The `space` directive emits a sequence of bytes, specified by the first expression, each with the fill value specified by the second expression. Both expressions must be constant expressions.

space-or-skip ::= `.space`
| `.skip`

space-directive ::= *space-or-skip expression*
| *space-or-skip expression* , *expression*

The `skip` directive is a synonym for the space directive. It is provided for compatibility with other assemblers.

6.26 type

The `type` directive specifies the type of a symbol to be either a function symbol or an object symbol.

type-directive ::= `.type` *symbol* , *symbol-type*

symbol-type ::= `@function`
| `@object`

6.27 size

The `size` directive specifies the size associated with a symbol.

size-directive ::= `.size` *symbol* , *expression*

6.28 jmptable, jmptable32

The `jmptable` and `jmptable32` directives generate a table of unconditional branch instructions. The target of each branch instruction is the next label in the list. The size of the each branch instruction is 16 bits for the `jmptable` directive and 32 bits for the `jmptable32` directive.

jmptable-directive ::= `.jmptable` *jmptable-list*$_{\text{opt}}$
| `.jmptable32` *jmptable-list*$_{\text{opt}}$

jmptable-list ::= *symbol*
| *jmptable-list symbol*

Each symbol must be a label. A maximum of 32 labels maybe specified. If the unconditional branch distance does not fit into a 16-bit branch instruction, a branch is made to a trampoline at the end of the table, which performs the branch to the target label.

7 Instructions

Assembly instructions are normally inserted into an ELF text section. The syntax of an instruction is:

instruction ::= *mnemonic instruction-args*$_{\text{opt}}$

instruction-args ::= *instruction-args* , *instruction-arg*
| *instruction-arg*

instruction-arg ::= *symbol* [*expression*]
| *symbol* [*expression*] : *symbol*
| *expression*

To target the dual issue execution mode of xCORE-200 devices, instructions may be put in bundles:

separator ::= *newline*
| ;

instruction-bundle ::= { ⟨*separator*⟩* *bundle-contents* ⟨*separator*⟩* }

bundle-contents ::= *instruction* ⟨*separator*⟩+ *instruction*
| *instruction*

The current issue mode, as specifed by the issue_mode directive (see §6.18), affects how the assembler assembles instructions. Initially the current issue mode is single and instruction bundles cannot be used. If the current issue mode is changed to dual then:

- Instruction bundles can be specified.
- 16-bit instructions not in an instruction bundle are implicitly placed in an instruction bundle alongside a NOP instruction.
- The encoding of some operands may change. For example the assembler applies a different scaling factor to the immediate operand of relative branch instructions to match the different scaling factor that the processor uses at runtime when the instruction is executed in dual issue mode.

The order in which instructions are listed in an instruction bundle is not significant. The assembler may reorder the instructions in the bundle to satisfy architectural constraints.

The assembly instructions are summarized below using the default assembly syntax. The architecture manual (see X7879) documents the architectural syntax of the instructions. The syntax directive is used to switch the syntax mode.

The following notation is used:

bitp	one of: 1, 2, 3, 4, 5, 6, 7, 8, 16, 24 and 32
b	register used as a base address
c	register used as a conditional operand
d, e	register used as a destination operand
i	register used as a index operand
r	register used as a resource identifier
s	register used as a source operand
t	register used as a thread identifier
u_s	small unsigned constant in the range 0...11
u_x	unsigned constant in the range 0...(2^x-1)
v, w, x, y	registers used for two or more source operands

A register is one of: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, sp, dp, cp and lr. The instruction determines which of these registers are permitted.

Where there is choice of instruction formats, the assembler chooses the format with the smallest size. To force a specific format, specify a mnemonic of the form INSTRUCTION_format where the instruction and format names are as described in the architecture manual. For example the LDWCP_ru6 mnemonic specifies the ru6 format of the LDWCP instruction.

7.1 Data Access

Mnemonic	Operands	Meaning
ld16s	d, b[i]	Load signed 16 bits
ld8u	d, b[i]	Load unsigned 8 bits
lda16	d, b[i]	Add to 16-bit address
lda16	d, b[-i]	Subtract from 16-bit address
ldap	r11, u_{20}	Load pc-relative address
ldap	r11, -u_{20}	Load pc-relative address
ldaw	d, b[i]	Add to a word address
ldaw	d, b[-i]	Subtract from a word address
ldaw	d, b[u_s]	Add to a word address immediate
ldaw	d, b[-u_s]	Subtract from a word address immediate
ldaw	r11, cp[u_{16}]	Load address of word in constant pool
ldaw	d, dp[u_{16}]	Load address of word in data pool
ldaw	d, sp[u_{16}]	Load address of word on stack
ldd	e, d, b[i]	Load double word (xCORE-200 only)
ldd	e, d, b[u_s]	Load double word immediate (xCORE-200 only)
ldd	e, d, sp[u_s]	Load double from the stack (xCORE-200 only)
ldw	et, sp[4]	Load ET from the stack
ldw	sed, sp[3]	Load SED from the stack
ldw	spc, sp[1]	Load SPC from the stack
ldw	ssr, sp[2]	Load SSR from the stack
ldw	d, b[i]	Load word
ldw	d, b[u_s]	Load word immediate
ldw	d, cp[u_{16}]	Load word from constant pool
ldw	r11, cp[u_{20}]	Load word from constant pool
ldw	d, dp[u_{16}]	Load word from data pool
ldw	d, sp[u_{16}]	Load word from stack
set	cp, s	Set constant pool
set	dp, s	Set data pointer
set	sp, s	Set the stack pointer
st16	s, b[i]	16-bit store
st8	s, b[i]	8-bit store
std	e, d, b[i]	Store double word (xCORE-200 only)
std	e, d, b[u_s]	Store double word immediate (xCORE-200 only)
std	y, x, sp[u_s]	Store double word on the stack (xCORE-200 only)
stw	sed, sp[3]	Store SED on the stack
stw	et, sp[4]	Store ET on the stack
stw	spc, sp[1]	Store SPC on the stack
stw	ssr, sp[2]	Store SSR on the stack
stw	s, b[i]	Store word
stw	s, b[u_s]	Store word immediate
stw	s, dp[u_{16}]	Store word in data pool
stw	s, sp[u_{16}]	Store word on stack

7.2 Branching, Jumping and Calling

Mnemonic	Operands	Meaning
bau	s	Branch absolute unconditional
bf	c, u_{16}	Branch relative if false
bf	c, -u_{16}	Branch relative if false
bl	u_{20}	Branch and link relative
bl	-u_{20}	Branch and link relative
bla	s	Branch and link absolute via register
bla	cp[u_{20}]	Branch and link absolute via CP
blat	u_{16}	Branch and link absolute via table
bru	s	Branch relative unconditional via register
bt	c, u_{16}	Branch relative if true
bt	c, -u_{16}	Branch relative if true
bu	u_{16}	Branch relative unconditional
bu	-u_{16}	Branch relative unconditional
dualentsp	u_{16}	Adjust stack, save link register and enable dual issue (xCORE-200 only)
entsp	u_{16}	Adjust stack and save link register and enable single issue
extdp	u_{16}	Extend data pointer
extsp	u_{16}	Extend stack pointer
retsp	u_{16}	Return

7.3 Data Manipulation

Mnemonic	Operands	Meaning
add	d, x, y	Add
add	d, x, u_s	Add immediate
and	d, x, y	Bitwise and
andnot	d, s	And not
ashr	d, x, y	Arithmetic shift right
ashr	d, x, bitp	Arithmetic shift right immediate
bitrev	d, s	Bit reverse
byterev	d, s	Byte reverse
clz	d, s	Count leading zeros
crc32	d, r, p	Word CRC
crc32_inc	d, e, x, y, bitp	Word CRC with address increment (xCORE-200 only)
crc8	r, o, d, p	8-step CRC
crcn	d, x, p, n	Variable step CRC (xCORE-200 only)
divs	d, x, y	Signed division
divu	d, x, y	Unsigned division
eq	c, x, y	Equal
eq	c, x, u_s	Equal immediate
ladd	e, d, x, y, v	Long unsigned add with carry
ldc	d, u_{16}	Load constant
ldivu	d, e, v, x, y	Long unsigned divide
lextract	d, x, y, u, bitp	Bitfield extraction from register pair (xCORE-200 only)

(continued)

Mnemonic	Operands	Meaning
linsert	d, e, x, y, bitp	Inserts a bitfield into a pair of registers (xCORE-200 only)
lmul	d, e, x, y, v, w	Long multiply
lsats	d, x, y	Saturate signed (xCORE-200 only)
lss	c, x, y	Less than signed
lsu	c, x, y	Less than unsigned
lsub	e, d, x, y, v	Long unsigned subtract
maccs	d, e, x, y	Mulitply and accumulate signed
maccu	d, e, x, y	Multiply and accumulate unsigned
mkmsk	d, s	Make mask
mkmsk	d, bitp	Make mask immediate
mul	d, x, y	Multiply
neg	d, s	Two's complement negate
not	d, s	Bitwise not
or	d, x, y	Bitwise or
rems	d, x, y	Signed remainder
remu	d, x, y	Unsigned remainder
sext	d, s	Sign extend
sext	d, bitp	Sign extend immediate
shl	d, x, y	Shift left
shl	d, x, bitp	Shift left immediate
shr	d, x, y	Shift right
shr	d, x, bitp	Shift right immediate
sub	d, x, y	Subtract
sub	d, x, u_s	Subtract immediate
unzip	d, e, x	Unzips a pair of registers (xCORE-200 only)
xor	d, x, y	Bitwise exclusive or
xor4	d, e, x, y, v	Bitwise exclusive or of four words (xCORE-200 only)
zext	d, s	Zero extend
zext	s, bitp	Zero extend immediate
zip	d, e, x	Zips together a pair of registers (xCORE-200 only)

7.4 Concurrency and Thread Synchronization

Mnemonic	Operands	Meaning
freet		Free unsynchronized thread
get	r11, id	Get thread ID
getst	d, res[r]	Get synchronized thread
mjoin	res[r]	Master synchronize and join
msync	res[r]	Master synchronize
ssync		Slave synchronize
init	t[r]:cp, s	Initialize thread's CP
init	t[r]:dp, s	Initialize thread's DP
init	t[r]:lr, s	Initialize thread's LR
init	t[r]:pc, s	Initialize thread's PC
init	t[r]:sp, s	Initialize thread's SP
set	t[r]:d, s	Set register in thread

(continued)

Mnemonic	Operands	Meaning
start	t[r]	Start thread
tsetmr	d, s	Set register in master thread

7.5 Communication

Mnemonic	Operands	Meaning
chkct	res[r], s	Test for control token
chkct	res[r], u_s	Test for control token immediate
getn	d, res[r]	Get network
in	d, res[r]	Input data
inct	d, res[r]	Input control token
int	d, res[r]	Input token of data
out	res[r], s	Output data
outct	res[r], s	Output control token
outct	res[r], u_s	Output control token immediate
outt	res[r], s	Output token of data
setn	res[r], s	Set network
testlcl	d, res[r]	Test local
testct	d, res[r]	Test for control token
testwct	d, res[r]	Test for position of control token

7.6 Resource Operations

Mnemonic	Operands	Meaning
clrpt	res[r]	Clear port time
elate	s	Throw exception if too late (xCORE-200 only)
endin	d, res[r]	End a current input
freer	res[r]	Free a resource
getd	d, res[r]	Get resource data
getr	d, u_s	Allocate resource
gettime	d	Get the reference time (xCORE-200 only)
getts	d, res[r]	Get port timestamp
in	d, res[r]	Input data
inpw	d, res[r], bitp	Input a part word
inshr	d, res[r]	Input and shift right
out	res[r], s	Output data
outpw	res[r], s, bitp	Output a part word
outpw	res[r], s, w	Output a part word immediate (xCORE-200 only)
outshr	res[r], s	Output data and shift
peek	d, res[r]	Peek at port data
setc	res[r], s	Set resource control bits
setc	res[r], u_{16}	Set resource control bits immediate
setclk	res[r], s	Set clock for a resource
setd	res[r], s	Set data
setev	res[r], r11	Set environment vector
setpsc	res[r], s	Set the port shift count

(continued)

Mnemonic	Operands	Meaning
setpt	res[r], s	Set the port time
setrdy	res[r], s	Set ready input for a port
settw	res[r], s	Set transfer width for a port
setv	res[r], r11	Set event vector
syncr	res[r]	Synchronize a resource

7.7 Event Handling

Mnemonic	Operands	Meaning
clre		Clear all events
clrsr	u_{16}	Clear bits in SR
edu	res[r]	Disable events
eef	d, res[r]	Enable events if false
eet	d, res[r]	Enable events if true
eeu	res[r]	Enable events
getsr	r11, u_{16}	Get bits from SR
setsr	u_{16}	Set bits in SR
waitef	c	Wait for event if false
waitet	c	Wait for event if true
waiteu		Wait for event

7.8 Interrupts, Exceptions and Kernel Calls

Mnemonic	Operands	Meaning
clrsr	u_{16}	Clear bits in SR
ecallf	c	Raise exception if false
ecallt	c	Raise exception if true
get	r11, ed	Get ED into r11
get	r11, et	Get ET into r11
get	r11, kep	Get the kernel entry point
get	r11, ksp	Get the kernel stack pointer
getsr	r11, u_{16}	Get bits from SR
kcall	s	Kernel call
kcall	u_{16}	Kernel call immediate
kentsp	u_{16}	Switch to kernel stack
krestsp	u_{16}	Restore stack pointer from kernel stack
kret		Kernel return
set	kep, r11	Set the kernel entry point
setsr	u_{16}	Set bits in SR

7.9 Debugging

Mnemonic	Operands	Meaning
dcall		Cause a debug interrupt

(continued)

Mnemonic	Operands	Meaning
dentsp		Save and modify stack pointer for debug
dgetreg	s	Debug read of another thread's register
drestsp		Restore non debug stack pointer
dret		Return from debug interrupt
get	d, ps[r]	Get processor state
set	ps[r], s	Set processor state

7.10 Pseudo Instructions

In the default syntax mode, the assembler supports a small set of pseudo instructions. These instructions do not exist on the processor, but are translated by the assembler into xCORE instructions.

Mnemonic	Operands	Definition
mov	d, s	add d, s, 0
nop		r0, r0, 0

8 Assembly Program

An assembly program consists of a sequence of statements.

program ::= ⟨*statement*⟩*

statement ::= *label-list*$_{\text{opt}}$ *dir-or-inst*$_{\text{opt}}$ *separator*

label-list ::= *label*
| *label-list label*

dir-or-inst ::= *directive*
| *instruction*
| *instruction-bundle*

directive ::= *align-directive*
| *ascii-directive*
| *value-directive*
| *file-directive*
| *loc-directive*
| *weak-directive*
| *vis-directive*
| *text-directive*
| *set-directive*
| *cc-directive*
| *scheduling-directive*
| *syntax-directive*
| *assert-directive*
| *xc-directive*
| *xta-directive*
| *space-directive*
| *type-directive*
| *size-directive*
| *jmptable-directive*
| *globalresource-directive*

XMOS®