XS1 Assembly Language Manual

(VERSION 8.7)

2008/07/23

Authors:

Douglas WATT
Richard OSBORNE
Martin YOUNG

1 Introduction

This manual describes the XS1 assembly language. The XS1 assembly language supports the formation of objects in the Executable and Linkable Format (ELF) [1] for the XS1 architecture [2, 3]. Extensions to the ELF format are documented in the XMOS ABI [4].

2 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.

2.1 Comments

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

2.2 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.

2.3 Directives

A directive begins with '**.**' followed by one or more letters. Directives instruct the assembler to perform some action; they are discussed in §7.

2.4 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 `01234-56789`.
- 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 [5, A2.5.2] may be used to specify certain characters.

3 Sections and Relocations

Named ELF sections are specified using directives (see §7.8). 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.

4 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 charac-

ters "`.L`". A local symbol may be discarded by the linker when no longer required for linking.

4.1 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 §7.9), or through its use in a label (see §5). The default binding of symbols in the undefined section is *global*; for all other symbols the default binding is *local*.

5 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.

6 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 $\pm$ a constant. An expression may be encoded in the ELF-extended expression section and its value evaluated by the linker (see §7.9); 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*

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

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}

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

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 behaviour as C , and operators with equal precedence are performed left to right. 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.

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.

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.

7 Assembler Directives

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

7.1 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.

7.2 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*
| *string*

7.3 byte, int, long, short

These directives emit, for each expression, a number that at run-time is the value of that expression. The byte order and bit size of the numbers are determined by the ABI. The expressions must be simple expressions.

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

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

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

7.4 file

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.

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

7.5 weak

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

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

7.6 globl, extern

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.

glob-directive ::= `.globl` *symbol* **,** $string_{opt}$
| `.extern` *symbol* **,** $string_{opt}$

If the optional string is provided then 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 then it is inserted.) The meaning of this string is determined by the ABI.

7.7 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.

7.8 section, pushsection and popsection

The section directives create ELF sections (see §3).

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 into the named section. The optional flags may contain any combination of the following characters:

`a`	section is allocatable
`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 then 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.

7.9 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 then the symbol is defined in the absolute section; if the expression is a symbol then 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.

7.10 assert

The assert directives are:

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 then a failure should be reported as a warning; if the constant is 1 it should be reported as an error. The string is a message for an assembler/linker to emit on failure.

7.11 Language Directives

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

xc-directive ::= *globdir symbol* , *symbol* , *string*
| .call *symbol* , *symbol*
| .par *symbol* , *symbol* , *string*

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

For each directive, the string is an error message for the assembler/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.

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.

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.

7.12 uleb128, sleb128

The following directives emit a value that encodes either an unsigned or signed DWARF little-endian base 128 number. The expression must be a simple expression

leb-directive ::= .uleb128 *expression*
| .sleb128 *expression*

7.13 space

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-directive ::= `.space` *expression*
| `.space` *expression* `,` *expression*

7.14 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`

7.15 size

The `size` directive specifies the size associated with a symbol. The expression must be a constant expression.

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

7.16 jmptable

The `jmptable` directive generates a table of 16-bit entries where each entry is an unconditional branch to the next label in the list.

jmptable-directive ::= `.jmptable` $jmptable\text{-}list_{opt}$

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

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

8 XCore XS1 Instructions

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

instruction	`::=`	*mnemonic instruction-args*$_{opt}$
instruction-args	`::=`	*instruction-args* `,` *instruction-arg*
	`|`	*instruction-arg*
instruction-arg	`::=`	*symbol* `[` *expression* `]`
	`|`	*symbol* `[` *expression* `]` `:` *symbol*

The assembly instructions for the XS1 architecture [2, 3] are given in tables in the following subsections. The instructions refer to the Rev-B architecture; see §8.1 for a description of differences from the Rev-A architecture.

There are two ways to specify an instruction: by writing its mnemonic in uppercase followed by a comma-separated list of its operands, or by using a lowercase notation that more clearly expresses its behaviour. The following notation is used:

b	denotes a bit-pattern, one of: bits per word, 1, 2, 3, 4, 5, 6, 7, 8, 16, 24, and 32 these are encoded using numbers 0...11
c	register used as a conditional operand
d, *e*	register used as a destination 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

Data Access

Mnemonic	Operands	Mnemonic	Operands	Size
LD16S	*d*, *b*, *i*	`ld16s`	`d,b[i]`	16
LD8U	*d*, *b*, *i*	`ld8u`	`d,b[i]`	16
LDA16B	*d*, *b*, *i*	`lda16`	`d,b[-i]`	32
LDA16F	*d*, *b*, *i*	`lda16`	`d,b[i]`	32
LDAPB	u_{20}	`ldap`	`r11,`u_{20}	16/32
LDAPF	u_{20}	`ldap`	`r11,`u_{20}	16/32
LDAWB	*d*, *b*, *i*	`ldaw`	`d,b[-i]`	32
LDAWBI	*d*, *b*, u_s	`ldaw`	`d,b[-`u_s`]`	32
LDAWCP	u_{16}	`ldaw`	`r11,cp[`u_{16}`]`	16/32
LDAWDP	*d*, u_{16}	`ldaw`	`d,dp[`u_{16}`]`	16/32
LDAWF	*d*, *b*, *i*	`ldaw`	`d,b[i]`	32
LDAWFI	*d*, *b*, *i*	`ldaw`	`d,b[i]`	32
LDAWSP	*d*, u_{16}	`ldaw`	`d,sp[`u_{16}`]`	16/32
LDC	*d*, u_{16}	`ldc`	`d,`u_{16}	16/32
LDET		`ldw`	`et,sp[4]`	16
LDSED		`ldw`	`sed,sp[3]`	16
LDSPC		`ldw`	`spc,sp[1]`	16
LDSSR		`ldw`	`ssr,sp[2]`	16
LDW	*d*, *b*, *i*	`ldw`	`d,b[i]`	16

(continued)

Mnemonic	Operands	Mnemonic	Operands	Size
LDWI	*d*, *b*, *i*	ldw	d, b[i]	16
LDWCP	*d*, u_{16}	ldw	d, cp[u_{16}]	16/32
LDWCPL	u_{20}	ldw	r11, cp[u_{20}]	16/32
LDWDP	*d*, u_{16}	ldw	d, dp[u_{16}]	16/32
LDWSP	*d*, u_{16}	ldw	d, sp[u_{16}]	16/32
MKMSK	*d*, *s*	mkmsk	d, s	16
MKMSKI	*d*, u_s	mkmsk	d, u_s	16
SETCP	*s*	set	cp, s	16
SETDP	*s*	set	dp, s	16
SETSP	*s*	set	sp, s	16
SEXT	*d*, *s*	sext	d, s	16
SEXTI	*d*, u_s	sext	d, u_s	16
ST16	*s*, *b*, *i*	st16	s, b[i]	32
ST8	*s*, *b*, *i*	st8	s, b[i]	32
STSED		stw	sed, sp[3]	16
STET		stw	et, sp[4]	16
STSPC		stw	spc, sp[1]	16
STSSR		stw	ssr, sp[2]	16
STW	*s*, *b*, *i*	stw	s, b[i]	32
STWI	*s*, *b*, *i*	stw	s, b[i]	16
STWDP	*s*, u_{16}	stw	s, dp[u_{16}]	16/32
STWSP	*s*, u_{16}	stw	s, sp[u_{16}]	16/32
ZEXT	*d*, *s*	zext	d, s	16
ZEXTI	*s*, u_s	zext	s, u_s	16

Branching, Jumping and Calling

Mnemonic	Operands	Mnemonic	Operands	Size
BAU	*s*	bau	s	16
BLA	*s*	bla	s	16
BLACP	u_{20}	blacp	u_{20}	16/32
BLAT	u_{16}	blat	u_{16}	16/32
BLRB	u_{20}	bl	u_{20}	16/32
BLRF	u_{20}	bl	u_{20}	16/32
BRBF	*c*, u_{16}	bf	c, u_{16}	16/32
BRBT	*c*, u_{16}	bt	c, u_{16}	16/32
BRBU	u_{16}	bu	u_{16}	16/32
BRFF	*c*, u_{16}	bf	c, u_{16}	16/32

(continued)

Mnemonic	Operands	Mnemonic	Operands	Size
BRFT	*c*, u_{16}	bt	c, u_{16}	16/32
BRFU	u_{16}	bu	u_{16}	16/32
BRU	*s*	bru	s	16
ENTSP	u_{16}	entsp	u_{16}	16/32
EXTDP	u_{16}	extdp	u_{16}	16/32
EXTSP	u_{16}	extsp	u_{16}	16/32
RETSP	u_{16}	retsp	u_{16}	16/32
SETCP	*s*	set	cp, s	16
SETDP	*s*	set	dp, s	16
SETSP	*s*	set	sp, s	16

Data Manipulation

Mnemonic	Operands	Mnemonic	Operands	Size
ADD	*d*, *x*, *y*	add	d, x, y	16
ADDI	*d*, *x*, u_s	add	d, x, u_s	16
AND	*d*, *x*, *y*	and	d, x, y	16
ANDNOT	*d*, *s*	andnot	d, s	16
ASHR	*d*, *x*, *y*	ashr	d, x, y	32
ASHRI	*d*, *x*, u_s	ashr	d, x, u_s	32
BITREV	*d*, *s*	bitrev	d, s	32
BYTEREV	*d*, *s*	byterev	d, s	32
CLZ	*d*, *s*	clz	d, s	32
CRC32	*d*, *x*, *p*	crc32	d, x, p	32
CRC8	*d*, *e*, *p*, *x*	crc8	x, d, e, p	32
DIVS	*d*, *x*, *y*	divs	d, x, y	32
DIVU	*d*, *x*, *y*	divu	d, x, y	32
EQ	*c*, *x*, *y*	eq	c, x, y	16
EQI	*c*, *x*, u_s	eq	c, x, u_s	16
LADD	*d*, *x*, *y*, *e*, *v*	ladd	e, d, x, y, v	32
LDIVU	*d*, *x*, *y*, *e*, *v*	ldivu	d, e, v, x, y	32
LMUL	*d*, *x*, *y*, *e*, *v*, *w*	lmul	d, e, x, y, v, w	32
LSS	*c*, *x*, *y*	lss	c, x, y	16
LSU	*c*, *x*, *y*	lsu	c, x, y	16
LSUB	*d*, *x*, *y*, *e*, *v*	lsub	e, d, x, y, v	32
MACCU	*d*, *x*, *y*, *e*	maccu	d, e, x, y	32
MACCS	*d*, *x*, *y*, *e*	maccs	d, e, x, y	32

(continued)

Mnemonic	Operands	Mnemonic	Operands	Size
MKMSK	*d*, *s*	`mkmsk`	`d, s`	16
MKMSKI	*d*, u_s	`mkmsk`	`d, u`$_s$	16
MUL	*d*, *x*, *y*	`mul`	`d, x, y`	32
NEG	*d*, *s*	`neg`	`d, s`	16
NOT	*d*, *s*	`not`	`d, s`	16
OR	*d*, *x*, *y*	`or`	`d, x, y`	16
REMS	*d*, *x*, *y*	`rems`	`d, x, y`	32
REMU	*d*, *x*, *y*	`remu`	`d, x, y`	32
SEXT	*d*, *s*	`sext`	`d, s`	16
SEXTI	*d*, u_s	`sext`	`d, u`$_s$	16
SHL	*d*, *x*, *y*	`shl`	`d, x, y`	16
SHLI	*d*, *x*, u_s	`shl`	`d, x, u`$_s$	16
SHR	*d*, *x*, *y*	`shr`	`d, x, y`	16
SHRI	*d*, *x*, u_s	`shr`	`d, x, u`$_s$	16
SUB	*d*, *x*, *y*	`sub`	`d, x, y`	16
SUBI	*d*, *x*, u_s	`sub`	`d, x, u`$_s$	16
XOR	*d*, *x*, *y*	`xor`	`d, x, y`	32
ZEXT	*d*, *s*	`zext`	`d, s`	16
ZEXTI	*s*, u_s	`zext`	`s, u`$_s$	16

Concurrency and Thread Synchronisation

Mnemonic	Operands	Mnemonic	Operands	Size
FREET		`freet`		16
GETID		`get`	`r11, id`	16
GETST	*d*, *r*	`getst`	`d, res[r]`	16
MJOIN	*r*	`mjoin`	`res[r]`	16
MSYNC	*r*	`msync`	`res[r]`	16
SSYNC		`ssync`		16
TINITCP	*s*, *r*	`init`	`t[r] : cp, s`	16
TINITDP	*s*, *r*	`init`	`t[r] : dp, s`	16
TINITLR	*s*, *r*	`init`	`t[r] : lr, s`	32
TINITPC	*s*, *r*	`init`	`t[r] : pc, s`	16
TINITSP	*s*, *r*	`init`	`t[r] : sp, s`	16
TSETMR	*d*, *s*	`tsetmr`	`d, s`	16
TSETR	*d*, *s*, *r*	`set`	`t[r] : d, s`	16
TSTART	*r*	`start`	`t[r]`	16

Communication

Mnemonic	Operands	Mnemonic	Operands	Size
CHKCT	*r*, *s*	chkct	res[r], s	16
CHKCTI	*r*, u_s	chkct	res[r], u_s	16
GETN	*d*, *r*	getn	d, res[r]	32
IN	*d*, *r*	in	d, res[r]	16
INCT	*d*, *r*	inct	d, res[r]	16
INT	*d*, *r*	int	d, res[r]	16
OUT	*s*, *r*	out	res[r], s	16
OUTCT	*r*, *s*	outct	res[r], s	16
OUTCTI	*r*, u_s	outct	res[r], u_s	16
OUTT	*s*, *r*	outt	res[r], s	16
SETN	*s*, *r*	setn	res[r], s	32
TESTLCL	*d*, *r*	testlcl	d, res[r]	32
TESTCT	*d*, *r*	testct	d, res[r]	16

Resource Operations

Mnemonic	Operands	Mnemonic	Operands	Size
CLRPT	*r*	clrpt	res[r]	16
ENDIN	*d*, *r*	endin	d, res[r]	16
FREER	*r*	freer	res[r]	16
GETD	*d*, *r*	getd	d, res[r]	32
GETR	*d*, u_s	getr	d, u_s	16
GETTS	*d*, *r*	getts	d, res[r]	16
IN	*d*, *r*	in	d, res[r]	16
INPW	*d*, *r*, u_s	inpw	d, res[r], u_s	32
INSHR	*d*, *r*	inshr	d, res[r]	16
OUT	*s*, *r*	out	res[r], s	16
OUTPW	*s*, *r*, u_s	outpw	res[r], s, u_s	32
OUTSHR	*s*, *r*	outshr	res[r], s	16
PEEK	*d*, *r*	peek	d, res[r]	16
SETC	*r*, u_{16}	setc	res[r], u_{16}	16/32
SETC	*r*, *s*	setc	res[r], s	32
SETCLK	*s*, *r*	setclk	res[r], s	32
SETD	*s*, *r*	setd	res[r], s	16

(continued)

Mnemonic	Operands	Mnemonic	Operands	Size
SETEV	*r*	`setev`	`res[r], r11`	16
SETPSC	*s*, *r*	`setpsc`	`res[r], s`	16
SETPT	*s*, *r*	`setpt`	`res[r], s`	16
SETRDY	*s*, *r*	`setrdy`	`res[r], s`	32
SETTW	*s*, *r*	`settw`	`res[r], s`	32
SETV	*r*	`setv`	`res[r], r11`	16
SYNCR	*r*	`syncr`	`res[r]`	16
TESTWCT	*d*, *r*	`testwct`	`d, res[r]`	16

Event Handling

Mnemonic	Operands	Mnemonic	Operands	Size
CLRE		`clre`		16
CLRSR	u_{16}	`clrsr`	u_{16}	16/32
EDU	*r*	`edu`	`res[r]`	16
EEF	*d*, *r*	`eef`	`d, res[r]`	16
EET	*d*, *r*	`eet`	`d, res[r]`	16
EEU	*r*	`eeu`	`res[r]`	16
GETSR	u_{16}	`getsr`	`r11,` u_{16}	16/32
SETSR	u_{16}	`setsr`	u_{16}	16/32
WAITEF	*c*	`waitef`	`c`	16
WAITET	*c*	`waitet`	`c`	16
WAITEU		`waiteu`		16

Interrupts, Exceptions and Kernel Calls

Mnemonic	Operands	Mnemonic	Operands	Size
CLRSR	u_{16}	`clrsr`	u_{16}	16/32
ECALLF	*c*	`ecallf`	`c`	16
ECALLT	*c*	`ecallt`	`c`	16
GETED		`get`	`r11, ed`	16
GETET		`get`	`r11, et`	16
GETKEP		`get`	`r11, kep`	16
GETKSP		`get`	`r11, ksp`	16
GETSR	u_{16}	`getsr`	`r11,` u_{16}	16/32

(continued)

Mnemonic	Operands	Mnemonic	Operands	Size
KCALL	u_{16}	`kcall`	$\mathtt{u}_{16}$	16/32
KCALL	*s*	`kcall`	`s`	16
KENTSP	u_{16}	`kentsp`	$\mathtt{u}_{16}$	16/32
KRESTSP	u_{16}	`krestsp`	$\mathtt{u}_{16}$	16/32
KRET		`kret`		16
SETKEP		`set`	`kep, r11`	16
SETSR	u_{16}	`setsr`	$\mathtt{u}_{16}$	16/32

Debugging

Mnemonic	Operands	Mnemonic	Operands	Size
DCALL		`dcall`		16
DENTSP		`dentsp`		16
DGETREG	*s*	`dgetreg`	`s`	16
DRESTSP		`drestsp`		16
DRET		`dret`		16
GETPS	*d*, *r*	`get`	`d, ps[r]`	32
SETPS	*s*, *r*	`set`	`ps[r], s`	32

8.1 Rev-A Architecture

In the Rev-A architecture, the instructions GETN, SETN, LDAWCP, LADD, LSUB, LDIVU, MACCU and MACCS are not supported; operands of type *b* have one of {1-9, 16, 24, 32}; and the following additional instruction is provided:

Mnemonic	Operands	Mnemonic	Operands	Size
MACC	*d*, *e*, *v*, *w*, *x*, *y*	`macc`	`d, e, v, w, x, y`	32

9 Assembly Program

An assembly program consists of a sequence of statements:

program **::=** ⟨*statement*⟩*

statement **::=** *label-list*$_{opt}$ *dir-or-inst next-on-line*$_{opt}$

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

next-on-line **::=** **;** *statement*$_{opt}$

A statement must be terminated by a newline (or a comment terminated by a newline).

10 Grammar

Below is a recapitulation of the grammar. The grammar has undefined terminal symbols constant, string and symbol; the `typewriter` style words and symbols are terminals given literally.

program **::=** ⟨*statement*⟩*

statement **::=** *label-list*$_{opt}$ *dir-or-inst next-on-line*$_{opt}$

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

next-on-line **::=** **;** *statement*$_{opt}$

directive **::=** *align-directive*
| *ascii-directive*
| *value-directive*
| *file-directive*
| *weak-directive*
| *glob-directive*
| *section-directive*
| *set-directive*
| *assert-directive*
| *xc-directive*

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

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

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

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

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

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

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

glob-directive ::= `.globl` *symbol* `,` $string_{opt}$
| `.extern` *symbol* `,` $string_{opt}$

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

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

flags ::= *string*

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

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

flag-args ::= *string*

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

set-type ::= .set
| .linkset

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

xc-directive ::= *globdir symbol* , *symbol* , *string*
| .call *symbol* , *symbol*
| .par *symbol* , *symbol* , *string*

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

leb-directive ::= .uleb128 *expression*
| .sleb128 *expression*

space-directive ::= .space *expression*
| .space *expression* , *expression*

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

symbol-type ::= @function
| @object

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

jmptable-directive ::= .jmptable *jmptable-list*$_{opt}$

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

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

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

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

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

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

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

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

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

string-list ::= *string-list* , *string*
| *string*

References

[1] The Santa Cruz Operation. System V Application Binary Interface, Edition 4.1. Website, 1997. http://www.caldera.com/developers/devspecs/gabi41.pdf.

[2] David May. XMOS XS1 Architecture. Website, 2008. http://www.xmos.com/published/xs1-87.

[3] David May and Henk Muller. XMOS XS1 Instruction Set Architecture. Website, 2008. http://www.xmos.com/published/xs1inst87.

[4] Douglas Watt and Richard Osborne and Martin Young. XMOS XS1 32-Bit Application Binary Interface. Website, 2008. http://www.xmos.com/published/abi87.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.