cadence®

Tensilica® Instruction Extension (TIE) Language

Reference Manual

For Cadence Xtensa® LX Processor Cores

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date:08/2017

RG-2017.7

PD-17-2001-10-06

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Figures

List of Tables

Preface

This document is written for Tensilica customers who have experience using design tools based on hardware description languages (such as Verilog-HDL or VHDL) and in system-level design.

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- `$` represents your shell prompt, in user-session examples.
- **literal_input** indicates literal command-line input.
- `variable` indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *... output ...* indicates unspecified program output.
- *[optional-*`variable`*]* indicates an optional parameter.
- *[optional-*`variable`*]** indicates zero or more instances of a parameter.
- **[**`variable`**]** indicates a parameter within literal square-braces.
- **{**`variable`**}** indicates a parameter within literal curly-braces.
- **(**`variable`**)** indicates a parameter within literal parentheses.
- | means *OR*.
- *(var1* | *var2)* indicates a required choice between one of multiple parameters.
- *[var1* | *var2]* indicates an optional choice between one of multiple parameters.
- `var1` *[,* `varn`*]** indicates a list of 1 or more parameters (0 or more repetitions).
- `::=` means *is equivalent to*.
- `4'b001x` is a 4-bit constant in binary representation.
- `12'o7x01` is a 12-bit constant in octal representation.
- `10'd483` is a 10-bit constant in decimal representation.
- `16'hffx2` or `16'HFFx2` is a 16-bit constant in hexadecimal representation.

The above notation includes a modified version of Backus-Naur Form (BNF). The modifications to generally accepted BNF notation include: (i) *italic font* rather than angle-braces identifies variables, (ii) `bold font` rather than bounding quotation marks identifies literal keywords, (iii) bold curly-braces, **{}**, are literal inputs rather than delimiters of

optional variables, (iv) square-braces are literal inputs when bold, **[]**, but delimiters of optional variables when non-bold italic, *[]*, and (v) parentheses are literal inputs when bold, **()**, but group-separators or delimiters of alternatives when non-bold italic, *()*.

Character Set

`variable_names` in the TIE language are case-sensitive and must:

- Begin with a letter.
- Optionally contain alphanumeric or underscore (_) characters after the first character.

Terms

- *b* means bit.
- *B* means byte.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

See the Index for pointers to definitions of additional terms.

Changes from the Previous Version

The following changes (denoted with change bars) were made to this document for the Cadence RG-2017.7 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Updated the default maximum width for the register file to 4,096 bits, see Section 5.6 “Implementation Restrictions” .
- Clarified the encoding restrictions for FLIX slotting for `instruction_group`, see Section 22.3 “Restrictions” .

The following changes (denoted with change bars) were made to this document for the Cadence RG-2017.5 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Made minor program code updates in various examples throughout the reference manual for accuracy.

The RG-2016.4 release is the first general release of Xtensa LX7 architecture and software. Xtensa software and tools are available to all users for software upgrade from previous releases.

The following changes (denoted with change bars) were made to this document for the Cadence RG-2016.4 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Switched the order of operations in the example so that the store operation is first then followed by the load operation, see Section 18.2 “Example” on page 156
- Added clarification for using the register file bypass construct, `regbypass`, see Chapter 23.

The following changes (denoted with change bars) were made to this document for the Cadence RG-2016.3 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Added clarification for `xfer_proto` when using a store instruction to store a user register value--the content in memory must be bit-by-bit identical to the content specified in the user register, see Section 17.1 “User Register Syntax” and Section 18.3 “Implementation Restrictions” .

The following changes (denoted with change bars) were made to this document for the Cadence RG-2015.2 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Added clarification for operator overloading, see Section 26.4 “Implementation *Restrictions*”

The following changes (denoted with change bars) were made to this document for the Cadence RG-2015.1 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Only void can be used as the ctype with memory, see Chapter 18
- Added information for `instruction_group`, see Chapter 22
- Added information for `regbypass`, see Chapter 23
- Added information for `regport`, see Chapter 24

The following changes were made to this document for the Cadence RG-2015.0 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

- Added description of new chapters in Chapter 1
- Added description for the `regfile` optional `no_bypass` attribute in Chapter 5
- Added description for the `xfer_proto` construct in Chapter 4 and Chapter 17
- Added new chapter for the `xfer_proto` construct, Chapter 18
- Added new chapter for the `instruction_group` construct, Chapter 22
- Added new chapter for the `regbypass` construct, Chapter 23
- Added new chapter for the `regport` construct, Chapter 24

1. Overview

The Tensilica Instruction Extension (TIE) language describes extensions to the core instruction set documented in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*. TIE instruction extensions can be useful for optimizing functionality or performance in specific applications. A TIE description specifies the syntax and semantics of new instructions. It also specifies any additional register files, processor states, or ports created by the designer.

1.1 Document Overview

This document is a reference manual for the TIE language, and provides specific implementation details about the TIE compiler. It provides a detailed description of every construct in the TIE language including its purpose, syntax, and usage examples. The reader is encouraged to be familiar with the *Tensilica Instruction Extension (TIE) Language User's Guide* before attempting to read this document. The user guide teaches how to write TIE code, starting with the basics and leading to the advanced concepts. This document, the *Tensilica Instruction Extension (TIE) Language Reference Manual* contains a complete, concise description of all the TIE language features.

This document is organized as follows:

- Chapter 1, "Overview" provides an overview of the TIE language and the structure of a TIE description.
- Chapters 2 through 9 describe the most common TIE constructs used in the creation of TIE descriptions.
- For LX cores Chapter 10, "Instruction Format (`format`) Sections"and Chapter 11, "Slot Opcode (`slot_opcodes`) Sections"describe the TIE constructs used to create FLIX TIE descriptions.
- For LX cores Chapters 12 through 14 describe the TIE constructs used to create new interfaces between the Xtensa microprocessor and other system components.
- Chapters 15 through 24 describe various advanced TIE constructs for the advanced user.
- Chapter 25, "Instruction Map (`imap`) Sections" provides information about a TIE construct that enables the XCC compiler to infer the use of TIE instructions from patterns of other TIE or Xtensa ISA instructions.
- Chapter 26, "Operator (operator) Sections" provides a TIE construct to map C/C++ built-in operator symbols to TIE protos.
- Chapter 27, "Cstub Swap (`cstub_swap`) Sections" describes a TIE construct only used in a native simulation environment.

- Chapter 28, “Computation Sections” and Chapter 29, “TIE Built-in Modules” specify the rules for describing instruction semantics and some built-in modules that can simplify the specification of these semantics.
- Chapter 30, “TIEprint” describes a TIE construct that can be used for debugging your TIE code when running it on the Xtensa Instruction Set Simulator.
- Chapter 31, “TIE Preprocessor”provides information about TIE preprocessing.
- Chapters 32 through 40 describe various TIE constructs that are not commonly used, but are available for those who want complete control over issues such as instruction encoding and field mappings.

1.2 Using the TIE Language and TIE Compiler

TIE language descriptions are created in a text editor or the TIE editor of the Xtensa Xplorer development environment. There are two methods for customizing Xtensa software tools — the compiler, debugger, assembler, disassembler, Instruction Set Simulator (ISS), and so forth — to support new TIE instructions:

1. Submit a TIE description to the Xtensa Processor Generator, build the processor, download the customized software tools, and install the software tools.
2. Use the TIE compiler *(tc)* to generate a TIE Development Kit (TDK), then use the kit to customize previously installed Xtensa software tools.

The second method, using the TIE compiler, is usually a more efficient way to develop and debug TIE descriptions. The TIE compiler is a software tool provided by Cadence that translates TIE-language descriptions into additional software-tool components and hardware components that are consistent with a specific implementation of the Xtensa architecture. The TIE compiler generates the following:

- Dynamic (shared) libraries that can be used by previously installed Xtensa software tools to support new TIE instructions.
- Intrinsic C function declarations that allow new instructions to be coded as functions in the application code.
- Verilog descriptions for the new hardware needed to support new instructions. This is used to measure the hardware cost and performance for the new instructions.
- An estimate of the area or amount of logic needed to implement the TIE extensions in hardware.
- A Design-Compiler script that can be used to synthesize the actual hardware from the Verilog description to measure the area and speed impact of the new instructions.
- A report on various architecture and micro-architecture characteristics of the TIE extensions.

The only way to customize the processor hardware is to submit the TIE descriptions to the Xtensa Processor Generator. Details of this design process are described in the *Tensilica Instruction Extension (TIE) Language User's Guide*.

1.3 Description Sections

A TIE description file contains a list of *description sections*. Each section specifies a particular aspect of a new instruction, each has its own syntax, and each has an associated list of predefined or reserved symbols in the core instruction set. Table 1–1 shows the TIE language description sections.

Table 1–1. Description Sections

Section Name	Description	Reference
`immediate_range`	Defines a range of values to be used as immediate constants with instruction.	Chapter 2
`table`	Defines tables of constants.	Chapter 3
`state`	Defines new state registers created by the designer.	Chapter 4
`regfile`	Defines new register files created by the designer.	Chapter 5
`interface`	Defines what interface signals to use between the base Xtensa processor and TIE extensions.	Chapter 6
`operation`	Defines the instruction opcode, format and behavior in one compact section.	Chapter 7
`schedule`	Specifies pipeline schedule for multicycle instructions.	Chapter 8
`semantic`	Defines the behavior of instructions with specific emphasis on efficient hardware implementation.	Chapter 9
`format`	Defines a FLIX instruction format. For LX cores	Chapter 10
`slot_opcodes`	Defines the instructions available within a FLIX slot. For LX cores	Chapter 11
`import_wire`	Defines a new input interface (TIE port) to the Xtensa core. For LX cores	Chapter 12
`queue`	Defines a new interface to an external queue. For LX cores	Chapter 13
`lookup`	Defines a new interface to an external lookup table or other device. For LX cores	Chapter 14
`ctype`	Defines new C datatypes associated with register files.	Chapter 15
`proto`	Defines instruction prototypes.	Chapter 16
`user_register`	Provides direct read/write access to designer-defined states.	Chapter 17
`xfer_proto`	Provides direct load/store access for user registers from/to system memory (Do not need to go through AR registers.)	Chapter 18
`coprocessor`	Groups TIE states and register files into coprocessors.	Chapter 19
`function`	Creates a new designer-defined function.	Chapter 20
`property`	Adds property attributes to other TIE constructs	Chapter 21

Table 1–1. Description Sections (continued)

Section Name	Description	Reference
`instruction_group`	Defines a list of instructions using one identifier which designers can use in place of a sequence of instruction names.	Chapter 22
`regbypass`	Directs the TIE compiler to add back selected direct bypasses.	Chapter 23
`regport`	Forces all arguments to be allocated to the same port	Chapter 24
`imap`	Defines instruction maps, which enable the XCC compiler to infer the use of TIE instructions.	Chapter 25
`operator`	Map TIE protos to C/C++ operators	Chapter 26
`cstub_swap`	Swap the memory byte-order in cstub for native simulation only.	Chapter 27
`TIEprint`	Defines the format of printing out TIE statement variables in ISS simulation.	Chapter 30
`field`	Defines instruction fields that can be used to create opcodes and operands.	Chapter 32
`opcode`	Defines instruction opcodes and their encoding.	Chapter 33
`operand`	Defines instruction operands.	Chapter 34
`operand_map`	Defines a mapping from an operation argument to an operand	Chapter 36
`length`	Defines an instruction length. For LX cores	Chapter 37
`slot`	Defines an instruction slot within a FLIX format. For LX cores	Chapter 38
`iclass`	Defines classes of instructions that have the same assembly format and operand usage.	Chapter 39
`reference`	Defines the behavior of one instruction with an emphasis on readability.	Chapter 40

1.3.1 TIE Program Syntax

A TIE file has the following syntax:

```
TIE-program ::= section [section*]
section ::= (immediate_range | table |state | regfile | interface |
operation | schedule | semantic | length | format | slot | slot_opcodes
| import_wire | queue | lookup | ctype | proto | user_register |
coprocessor | test | function | field | opcode | operand | iclass |
reference | imap | cstub_swap | property | operator | operand_map)
```

The syntax for each type of description section is defined in the chapters that follow. Description sections start with a keyword and have optional descriptors. Description sections end before the keyword for the next description section. A single description section can span multiple lines. White spaces are used as delimiters.

1.3.2 Predefined and Reserved Symbols

Certain names and symbols are predefined or reserved by the Xtensa processor instruction set architecture. These symbols can be found in the chapter that covers instruction formats and opcodes in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*. Avoid using symbols that are reserved, and be aware of these symbols, as predefined symbols may be needed.

For example, in the case of opcode sections, two predefined opcode names (`CUST0` and `CUST1`) are intended specifically for TIE extensions. The mapping of these opcodes is described in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* chapter that covers instruction formats and `opcodes`.

The TIE compiler responds with an error message if symbols that are reserved for the instructions of the Xtensa architecture are redefined. For example, `op0` is a predefined field in the Xtensa ISA and cannot be redefined in TIE description. TIE language and C/C++ keywords, shown in Table 1–2, are also disallowed.

Table 1–2. TIE and C/C++ Language Keywords

Type	Keywords
TIE	All description names in Table 1–1, all names of built-in TIE modules in Chapter 29, all names of predefined interface signals in Table 6–3, add_read_write, assign, core, immediate, in, inout, Inst, InstBuf, MEM, out, rdy, wire, export, shared_or
C/C++ language	alignof, and, and_eq, asm, attribute, auto, bitand, bitor, bool, break, case, catch, char, class, compl, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, not, not_eq, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, typeof, union, unsigned, using, virtual, void, volatile, while, xor, xor_eq

1.3.3 Naming Convention and Restrictions

Certain naming restrictions have been imposed on the TIE language. This is to ensure that the hardware generated from TIE descriptions is compatible with the Verilog hardware description language, and the software tools generated are compatible with the C/C++ programming languages. These restrictions are:

- Names cannot begin or end with an underscore (‘ _ ‘).
- Names cannot begin with the prefix “TIE_” or “TIE” to avoid any duplication with internal signals generated by the TIE compiler.
- When the name of any TIE construct has a period (“.”) in it, it is translated to the underscore (“_“) character internal to the TIE compiler. This is to avoid conflict with the C language syntax of using the period character to access the components of a

struct. Thus an instruction named FOO.N will have a C intrinsic with the name FOO_N. This convention also implies that you cannot have two names in your TIE description that differ only in the dot and underscore character.

- The TIE language is case-sensitive. However, various other tools that work with the output of the TIE compiler may or may not be case-sensitive. For example, the C programming language is case-sensitive, but the assembler is case-insensitive. Similarly, the Verilog hardware description language is case-sensitive, but the VHDL hardware description language is case-insensitive. As a result, even though the TIE language is case-sensitive, the names of various TIE constructs have to be unique in a case-insensitive manner. For example, if two instruction names differ only in case, the assembler cannot distinguish between them. The best way to avoid name conflicts is to write your TIE description such that you preserve the case, and not create multiple names that differ only in case. Specific cases of naming conflict that are detected by the TIE compiler and signaled as an error are listed in the next bullet.
- Do not use two TIE variables within the same scope that differ only in case. Names within the same scope are checked for duplications in a case-insensitive manner for the following cases:
  - Same TIE construct: Having two `regfiles` with names “XX” and “xx” is illegal.
  - Semantic variables: There are some predefined variables inside semantics, such as the instructions in the semantic. Names that conflict with these predefined variables are also illegal. For example:

```
semantic Y { X }
{
wire [31:0] x;
}
```

is illegal because the name “x” conflicts with the name of the instruction X, which is an implicit input to the semantic.

1.3.4 Specifying Vectors or Bit Range

All wires of bit-width greater than 1 must be declared as [high# : low#]. Declarations of the form [low# : high#] are not allowed. The same applies to specifying bit ranges.

1.3.5 Order of Description Sections

All description sections must be written in a define-before-use order. Thus, forward references are not allowed. The only exception to this rule is that a `format` declaration is allowed (and required) to list the `slots` that make up the format.

1.3.6 Comments

Two kinds of comments can be used:

- Block comments that start with `/*` and end with `*/`. These comments cannot be nested.
- Line comments that start with `//` and go to the end of the line.

2. Immediate Range (immediate_range) Sections

The immediate_range construct provides a way to define a range of immediate values that can be used as the operand of an instruction. It is an abstraction that represents the range of legal values that an immediate operand can take. After an immediate range has been declared, it can be used as the *type* of an *argument* in an operation statement. The TIE compiler automatically generates an operand to represent this immediate range, and also chooses an appropriately sized field to encode it in the instruction word. Refer to Chapter 7, “Instruction Operation (operation) Sections” on page 37 for an example of how an immediate_range gets used as an argument in an operation section. Alternately, use the immediate range to explicitly create an immediate operand using the operand construct as described in Section 34 “Instruction Operand (operand) Sections” on page 261.

2.1 Immediate Range Syntax

```
immediate_range-def ::= immediate_range name
                                low_value high_value step_size
name ::= a unique identifier
low_value ::= lowest numerical value in the range
high_value ::= highest numerical value in the range
step_size ::= difference between successive values in the range
```

name is a unique identifier that can later be used in an operation or operand section. *low_value* is a signed integer representing the lowest value that an operand of this type can have. *high_value* is a signed integer representing the highest value that an operand of this type can have. The *high_value* must be greater than the *low_value*. *step_size* is the amount by which successive values within this range increment, starting from *low_value* and going up to *high_value*. *step_size* needs to be a positive integer that is a power of 2. Both the *low_value* and the *high_value* must be a multiple of *step_size*.

In some situations, the number of bits needed to encode the values within the immediate range may allow additional values (outside the specified range) to be encoded. In such situations, the TIE compiler extends the range as appropriate, fully utilizing the encoding bits. However, this behavior is implementation dependent, and thus not guaranteed. Conversely, the TIE compiler may not be able to encode an immediate range that spans a very large number of values if it cannot find unused bits in the instruction word. If the TIE compiler is not able to perform the encoding, an error message is generated.

Use the immediate_range construct to create operands that are linearly increasing according to a fixed step size (where the step size is a power of 2). For information about only creating immediate operands that do not follow this restriction, refer to Chapter 3, “Constant Table (table) Sections” on page 11.

By definition, all immediate operands are assumed to be 32-bits wide. Each value of an immediate_range will be *sign extended* to 32 bits when used in any computation inside an operation block. It is illegal to define immediate operands that are wider than 32 bits.

2.2 Examples

The following example creates an immediate range representing the values {-8, -6, -4, -2, 0, 2, 4, 6}:

```
immediate_range imrange -8 6 2
```

The next example defines an immediate range representing the following values {0, 1, 2, 3, 4, 5, 6}. This range can be automatically extended by the TIE compiler to include the value 7, because values in the range [0, 7] can be encoded using the same number of bits as values in the range [0, 6]. However, this behavior is implementation dependent, and should not be relied upon. Always specify all the desired values in the immediate_range construct. If automatic range extension does occur, then the C compiler or assembler can use values from this extended range.

```
immediate_range imr0 0 6 1
```

In the following example, the specification of imr1 is incorrect because the *high_value* (7) is not a multiple of *step_size* (2). The specification of imr2 is incorrect because the *step_size* (3) is not a power of 2.

```
immediate_range imr1 0 7 2         // Error case

immediate_range imr2 30 60 3       // Error case
```

3. Constant Table (table) Sections

Constant tables are defined with table description sections. Constant tables allow the designer to define a hardware table that can be used in any computational expression. For example, it is possible to encode sets of arbitrary constants as immediate operands. In many cases, immediate operands can be extracted directly by either a signed or unsigned extension of an instruction field, but constant tables are more flexible.

Tables can be used as explicit operands of an instruction or as “globals” that can be used inside any function, operation, semantic, or reference section.

3.1 Table Syntax

```
table-def ::= table name width size value-list
name ::= a unique identifier
width ::= number of bits in each value
size ::= number of values in value-list
value-list ::= {constant [, constant]*}
```

name is a unique identifier that can later be used to reference the table. *size* specifies the number of values in the table. *value-list* is a list of one or more comma-delimited constant values. These constant values can be specified as either sized constants or integers. Unsized integers are implicitly converted to 32-bit constants. If the size of a constant is greater than *width*, the constant is truncated to *width* bits. If the constant has fewer than *width* bits, it is zero-extended. If the number of values in *value-list* is less than *size*, the remaining entries are assumed to be zero. If size is not an integer power of 2, indexing the table from size to the next power of 2 will return a zero. The number of bits required to index a table is the smallest integer greater than or equal to log2(size). Any additional bits used to index a table will be ignored.

If you list negative integers in the value-list of a table, it is recommended that you set the width of the table to 32. Because unsized integers are implicitly converted to 32-bit constants, a negative value in a table of width less than 32 may not be represented correctly in the design. The TIE compiler generates a warning message for tables of width less than 32 that list negative integers in the value-list.

3.2 Examples

The following TIE code defines an 8-bit wide table with 16 entries:

```
table prime 8 16 {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53
}
```

The following illustrates two different ways for an instruction to use this table. The instruction ADDP uses the table as an explicit immediate operand, while the instruction SUBP uses it as a “global”.

```
operation ADDP {out AR sum, in AR a, in prime p} {} {
     assign sum = a + p;
}

immediate_range IR0 0 15 1
operation SUBP {out AR diff, in AR a, in IR0 i} {} {
     assign diff = a - prime[i];
}
```

The instruction ADDP takes two input operands, one of which is an AR regfile operand and the other is a value from the table prime. The assembly (or C) language syntax of this instruction specifies the table value (and not the index) as an immediate operand. The table index is encoded in the instruction word by the assembler, and the hardware translates the index into the table value when executing the instruction. Thus the argument “p” of the operation refers to the table value specified by the instruction.

The instruction SUBP also has two input operands: an AR regfile operand, and an immediate value in the range 0 to 15. The immediate is used to index into the table to generate the value that gets subtracted from the AR regfile value.

The following are some legal and illegal examples of the instructions ADDP and SUBP:

```
ADDP a2, a0, 13     // a2 = a0 + 13
ADDP a2, a0, 4      // Error: 4 is not a valid table entry.
                    // Use values from the table, not the table index
                    // Thus, use 11 if you want the 4th table entry.

SUBP a3, a4, 5      // a3 = a4 - 13. Because prime[5] = 13.
SUBP a5, a3, 53     // Error: 53 is not a valid immediate
                    // Use 15 as the operand to refer to the table
                    // value of 53.
```

3.3 Implementation Restrictions

The implementation restrictions for the `table` sections are:

- The *width* of a table is limited to a maximum of 32 bits.
- The index of a table cannot itself be indexed by another constant or variable. To get this behavior, create a temporary wire of the appropriate size and use it to index the table.

4. Processor State (state) Sections

Processor states (or registers) are defined with state description sections. States behave like variables that hold a value prior to the execution of an instruction. This value can be used for a computation performed by the instruction. An instruction can also assign a value to a state, which is then updated with this new value after the execution of the instruction. Thus, *designer-defined states* can be thought of as registers or flip-flops that hold a value written to them, and can be used for processing by an instruction. The TIE compiler ensures that designer-defined states are implemented in a way consistent with the processor's micro architecture. All logic necessary for handling interrupts, exceptions, and pipeline stalls is generated automatically.

Designer-defined states can be used in operation, semantic and reference description sections. They cannot be used in operand description sections.

4.1 State Syntax

```
state-spec ::= state name width [<reset_value>] [shared_or]
                  [export] [add_read_write]
name ::= a unique identifier
width ::= an integer equal to the bit-width of the state
reset_value ::= a sized constant
shared_or ::= an optional keyword For LX cores
export ::= an optional keyword For LX cores
add_read_write ::= an optional keyword
```

name is a unique identifier that can later be used to reference the state. *width* is the number of bits in the state. The bit-index of designer-defined states is always from *width–1* down to *0*. *reset_value* specifies the initial value of the state on reset. *shared_or* is an optional keyword that indicates that multiple operations in a FLIX instruction can write to the state in the same cycle, and the result is the or of all operations. *export* is another optional keyword that indicates that the state is visible on a TIE port outside the processor. *add_read_write* is an optional and deprecated keyword, which indicates that a general purpose mechanism to read and write this state is desired. The xfer_proto construct can be used to realize this feature. For more information, refer to Chapter 18.

4.2 General Purpose Software Access to States

While any designer-defined instruction can access a state, it is useful to have specific instructions that provide a well-defined, but general purpose way to read and write states. The generation of these read/write instructions is optional, but it is important to understand the consequences of not having them–corresponding states are not accessible in a hardware debug environment, and may not be possible to save and restore these states in interrupt or exception handlers, and context switching tasks. In some situations this functionality may not be needed.

You can generate and use these instructions using one of the following methods:

- Write your own instructions and associate them with xfer_proto, which is the preferred method. For details refer to Section 18 “Transfer Prototype (xfer_proto) Sections” on page 155.
- Let the TC generate these instructions for you by specifying the add_read_write optional keyword, which is a deprecated method with limitations. For details refer to Chapter 17, “User Register (user_register) Sections” on page 149.

4.3 Exporting a State For LX cores

The keyword *export* in the description of a state makes the state a primary output of the Xtensa processor. The TIE_<*name*> port is automatically created, and has the current architectural value of this state. Use this feature to have the status of the processor visible to an external device or another processor.

The externally visible value on the port changes only when the architectural value of the state changes; speculative writes to the state are not reflected on the port.

It is important to understand the concepts of synchronization and ordering when designing a system using an exported state. The exact cycle in which the port is updated (with the value of a recent write to the state) is implementation dependent. Also, there is no output signal that indicates when the port is updated with the new value. It is therefore important to take care in designing systems in which an exported state controls an external device that interacts with the processor. If an instruction that writes to such a state is followed by an instruction that reads from the device, it is important to make sure that the new state value is seen by the device before the read is executed. Without adequate synchronization, it is possible for the read to occur before the write is seen by the external device. The Xtensa ISA provides a synchronization instruction, EXTW (external wait), that can be useful in this situation. EXTW ensures that all externally visible actions from earlier instructions from the processor prior to the EXTW instruction are executed before the pipeline can proceed to the next instruction. By inserting an EXTW instruction between the write to the state and the read from the device, you can ensure that the new value of the state will be visible on the corresponding port before the read is initiated. Note that the use of EXTW only guarantees that at the boundary of the Xtensa processor

and the external logic, all interfaces are ordered. If there are buffering or other delays in the external logic, additional synchronization steps may be necessary for the system to work as expected.

The behavior described in the previous paragraph implies that an instruction cannot write to an exported state and read back a value (from an external device) dependent on the exported state.

In general, the relative ordering between state export and memory requests, `import_wire` reads, TIE queue and lookup accesses, as written in a program is not guaranteed to be preserved by the Xtensa C compiler (XCC) by default. Thus a write to an exported state followed by a memory load in the C program may be reordered by XCC and will be seen in reverse order by external logic. Again, an `EXTW` instruction after the write to the state and before the load instruction would guarantee the correct order at the Xtensa processor boundary. To achieve this, add the appropriate synchronization pragma in the C program. This subject is also covered in Section 12.4 “Synchronization and Ordering” on page 88 and Section 13.5 “Synchronization and Ordering” on page 97.

4.4 ORing a State For LX cores

The keyword *shared_or* in the description of a state allows the state to be written by multiple operations in a single FLIX instruction and defines the result to be the bitwise OR of the individual results. Multiple operations can be executed in the same cycle using a FLIX instruction. Refer to Chapter 10, “Instruction Format (`format`) Sections” on page 69 for details on FLIX instructions. Use the *shared_or* attribute if two operations write the same state in the same FLIX instruction, otherwise the assembler will generate an error.

Shared_or is normally used in conjunction with the *ignore_state_output* property (see Chapter 21, “Property (property) Sections” on page 177). Without the property, the compiler will not bundle together multiple operations that write a state (in which case, you must use hand-bundled assembly code for bundling).

An example use of a *shared_or* state is to implement an *overflow* bit. If an operation generates an overflow, the overflow state is set to one, otherwise its previous value is preserved. Using the *shared_or* state, multiple operations that can potentially overflow can be scheduled in the same instruction, and the overflow bit is set if any of those operations do overflow.

4.5 Reset Behavior of States

States that specify a reset value in their definition will be initialized to that value upon reset. If no reset value is specified, and the state is not mapped to a user register, it is initialized to 0 upon reset. States that are mapped to user registers and do not specify a reset value are not initialized at reset time. The programmer must initialize these states (using the WUR.<*name*> instruction) to a known value before an instruction uses them. You must specify a reset value for exported states.

4.6 Examples

The following example declares four states of different sizes, none of which have the read/write instructions automatically generated.

```
state DATA 64
state KEYC 32
state KEYD 32
state TIME 16
```

In the following example, state S1 is a 20-bit wide state and state ROUND is a 50-bit wide state. Both are defined with the optional *add_read_write* argument. Thus, the TIE compiler automatically generates the instructions RUR.S1 to read state S1 and WUR.S1 to write state S1. Similarly, the TIE compiler generates the instruction RUR.ROUND_0 to read bits [31:0] of the state ROUND, and the instruction RUR.ROUND_1 to read bits [49:32]. The instructions WUR.ROUND_0 and WUR.ROUND_1 write the corresponding bits.

```
state S1 20 add_read_write
state ROUND 50 add_read_write
```

In the following example, state FOO is a 16-bit state with a reset value of 0xFFFF. State BAR is an 8-bit state that is not mapped to any user registers, and thus has a reset value of 0x00. State FOOBAR is a 10-bit state that has an undefined value coming out of reset.

```
state FOO 16 16'hFFFF add_read_write
state BAR 8
state FOOBAR 10 add_read_write
```

For LX cores In the following example, state COUNT is a 32-bit state that is exported. This results in a 32-bit output port, TIE_COUNT, coming out of the Xtensa processor.

```
state COUNT 32 32'h00001111 add_read_write export
```

For LX cores In the following example, state OVERFLOW is a 1-bit state that has shared_or attribute. This allows multiple operations accessing the state to be in one FLIX instruction.

```
state OVERFLOW 1 add_read_write shared_or
```

4.7 Implementation Restrictions

Following are the implementation restrictions for the processor `state` sections:

- The *width* of a state register is limited to a maximum of 1024 bits.
- The total number of states in an Xtensa configuration is limited to a maximum of 1024. This limit includes states defined by the base Xtensa processor of your configuration, as well as states in designer-defined TIE code.
- For LX cores States that are exported must have a reset value defined by the designer.
- The earliest use stage of a state must be less than or equal to its latest `def` stage. In other words, it is illegal to have the first use stage of a state be greater than its last `def` stage.
- The identifier *name* is restricted to a length of 63 characters or less.
- For LX cores Operations that access states with *shared_or* attribute cannot access any interface.
- For LX cores States with *shared_or* attribute cannot be written conditionally. Detailed description on conditional write can be found in Chapter 7, “Instruction Operation (`operation`) Sections” on page 37
- For LX cores All operations in a FLIX instruction that access states with *shared_or* attribute must access the state (read and write) in the same schedule cycle. This rule applies to all FLIX instructions across all formats with multiple slots. It does not apply if the FLIX instruction only contains one slot. Detailed description on schedule cycle can be found in Chapter 8, “Schedule (`schedule`) Sections” on page 47

5. Register File (regfile) Sections

The designer can create additional register files with regfile description sections and use these designer-defined register files as operation arguments of TIE instructions. The TIE compiler ensures that designer-defined register files are implemented in a way consistent with the processor's microarchitecture. All logic necessary for handling interrupts, exceptions, and pipeline stalls is generated automatically.

5.1 Register File Syntax

```
regfile-spec ::= regfile name width depth short-name [no_bypass]
[groups]
name ::= a unique identifier
width ::= bit-width of the register file
depth ::= number of entries in the register file
short-name ::= short name used as the register operand prefix
no_bypass ::= an optional keyword
groups ::= an optional keyword, one-group[ one-group] [port | view]
one-group ::= group-name = [2|4]
port ::= the register file group feature is implemented using multiple
ports
view ::= the register file group feature is implemented using multiple
widths
```

name is a unique identifier that can be used later to reference the register file. *width* is the number of bits in each entry of the register file. *depth* is the number of entries of the register file[1]. *short-name*, typically a single letter, is the prefix in the register operand associated with the register file in assembly instructions. For example, if a short name for register file VR is v, v3 would be the register operand for VR[3]. By default, the register file is fully bypassed, which means that data written to the register file in one cycle is available for consumption in the next cycle. When the optional no_bypass attribute is specified, the register file is generated without the direct bypass hardware; hence, the data written to the register file is available for consumption two cycles later. The register file may optionally contain one or two groups. *group-name* is a variation of the register file that can access two or four consecutive entries together. The variation can be implemented in two ways. One is to utilize multiple ports to access the grouped data, which is specified by the port keyword. The other is to vary the width of a register file port to access the grouped data, which is specified by the view keyword.

1. Before release RE-2013.0 of the TIE compiler, the number of entries of a register file must be power of two.

5.2 Using Register Files in C/C++ Programs

The ability to create designer-defined register files and to use them as arguments for instructions provides a very powerful set of tools for TIE development. The TIE compiler can automatically generate a new data type corresponding to a designer-defined register file. This data type has the same name as the register file name, and can be used in C/C++ programs as the *type* of variables that reside in the register file. The TIE compiler can also generate a set of load/store instructions to move data between the register file and memory, and a move instruction to move data between two registers of this register file. These instructions are then used to automatically generate *prototypes* that are used by the C/C++ compiler to enable it to perform register allocation for the designer-defined register file. Note that if a register file or a ctype is wider than the memory access width of the Xtensa processor, the designer must manually create ctypes, protos, and load and store instructions; the TIE compiler will not automatically generate them.

The TIE language also allows the designer to explicitly specify all the above information corresponding to their register file. Refer to Chapter 15, “C Datatype (`ctype`) Sections” on page 121 for information about declaring additional data types that reside in designer-defined register files. Chapter 6, “Interface Signals (`interface`) Sections” on page 29 describes how to create load/store instructions for a designer-defined register file. Chapter 16, “Prototype (`proto`) Sections” on page 127 describes how to teach the C/C++ compiler to perform register allocation for the designer-defined register files.

When the TIE compiler automatically generates load, store, and move instructions for a register file, it uses a set of stylized names for these instructions. The names of the load/store/move instructions generated by the TIE compiler are `ld.`<*regfile*>, `st.`<*regfile*>, and `mv.`<*regfile*> respectively, where <*regfile*> is the name of the register file. Instructions with the above names should only be used to load, store, or move register file entries. They should be used in <*ctype*>`_loadi`, <*ctype*>`_storei`, and <*ctype*>`_move` protos, where <*ctype*> is a ctype of the register. If you explicitly provide the load, store, and move instructions and the associated prototypes for a register file, you need not follow the above naming convention.

Each designer-defined register file is required to have at least one ctype that spans the entire register file, as described in Chapter 15. Furthermore, there needs to be <*ctype*>`_loadi`, <*ctype*>`_storei`, and <*ctype*>`_move` prototypes for this ctype as described in Chapter 16. Your may explicitly define these constructs in your TIE file, or rely on the TIE compiler’s ability to automatically generate them. If you specify the appropriate ctype and prototype declarations for a register file in the TIE description, the TIE compiler turns off the auto-generation of these constructs for that register file.

Designer-defined register files can have multiple read and write ports. An operation can read multiple values from a register file, and the TIE compiler automatically determines the number of read ports required for each designer-defined register file. Similarly, an operation can write multiple values to a register file[1].

5.3 Example

The following statement declares a 128-bit wide, 16 entry register file called VR, with the short name v. As a result of this declaration, a new data type VR is generated for use in C/C++ programs, to declare variables that reside in this register file. In assembly language programs, the registers of this register file are referred using the symbols v0, v1, v2 ... v14, v15.

```
regfile VR 128 16 v
```

If this register file is defined in an Xtensa processor configuration with 128-bit or higher data memory access width, the TIE compiler can automatically generate the load, store, and move instructions and the associated prototypes for this register file. If a narrower data memory access width (64 or 32) is chosen, then the designer must define the instructions (and the associated prototypes) for loading, storing, and moving entries of this register file.

5.4 Register Group

A register file operation argument usually refers to one entry of the register file. It is viewed as a base register file. With register group, an operation argument can refer to two or four consecutive entries of the register file. The description of operation can be found in Chapter 7, “Instruction Operation (operation) Sections” on page 37. Thus, the operation argument width is two or four times the width of a register file entry. The number of bits to encode the operation argument is one bit or two bits less than the number of bits required to encode the operation arguments on the base register file. To utilize the feature, the operation argument needs to specify the group name as the argument type.

The register group is viewed as a separate register file implemented on the base register file. When two entries are grouped together, the register group contains half of the number of entries of the base register file. Each entry is twice the width of the base register file. The lower half of the entry is mapped to the even entry of the base register file, while the upper half of the entry is mapped to the odd entry of the base register file. When four entries are grouped together, the register group contains a quarter of the number of entries but each entry is four times wide. The lowest quarter of the register group entry is mapped to the base register file entry with lowest index. Similarly, the highest quarter is mapped to the base register file entry with the highest index (lowest index plus three). The register group also has its own ctypes as well as the default ctype. The ctypes defined on the register group need to be structure ctypes of the ctypes on the base register file. The first field in the struct ctype is mapped to the base

1. Before release RC-2009.0 of the TIE compiler, one operation can only write to one entry of a register file. Before release RE-2013.0 of the TIE compiler, one operation can only read or write three entries of the AR register file.

register file entry with the highest index, while the last field is mapped to the entry with the lowest index. The ctypes on the register group have their load, store and move protos defined. The description of ctype can be found in Chapter 15, “C Datatype (ctype) Sections” on page 121. The description of proto can be found in Chapter 16, “Prototype (proto) Sections” on page 127.

5.4.1 Example

The following example illustrates the usage of the register group.

```
regfile XR 16 16 xr XR2=2
ctype XR 16 16 XR default
ctype XR2 32 32 XR2 {XR a, XR b}

immediate_range immed2 0 60 4

operation XR2_add {out XR2 a, in XR2 b, in XR2 c} {} {
       assign a = b + c;
}

/* a[15:0] is loaded to the an even entry of XR, say n'th entry
   a[31:16] is loaded to an odd entry (n+1) of XR
   Suppose the load address is p:
     On a little endian processor,
       value saved in {p+1, p} is loaded to the even entry of XR
       value saved in {p+3, p+2} is loaded to the odd entry of XR
     On a big endian processor,
       value saved in {p, p+1} is loaded to the odd entry of XR
       value saved in {p+2, p+3} is loaded to the even entry of XR
*/
operation ld.XR2 {out XR2 a, in XR2 *b, in immed2 imm} {out VAddr, in
MemDataIn32} {
       assign VAddr = b + imm;
       assign a = MemDataIn32;
}

operation st.XR2 {in XR2 a, in XR2 *b, in immed2 imm} {out VAddr, out
MemDataOut32} {
       assign VAddr = b + imm;
       assign MemDataOut32 = a;
}

operation mv.XR2 {out XR2 a, in XR2 b} {} {
       assign a = b;
}

proto XR2_loadi {out XR2 a, in XR2 *b, in immediate imm} {} {
```

```
      ld.XR2 a, b, imm;
}

proto XR2_storei{in XR2 a, in XR2 *b, in immediate imm} {} {
      st.XR2 a, b, imm;
}

proto XR2_move{out XR2 a, in XR2 b} {} {
      mv.XR2 a, b;
}

operation extract_a {out XR a, in XR2 b} {} {
       assign a = b[31:16];
}
proto extract_a {out XR a, in XR2 b} {} {
          extract_a a, b;
}

proto extract_a_prime {out XR a, in XR2 b} {} {
     mv.XR a, b->a;
}
```

In this example, register file XR contains 16 entries, each 16-bit wide. Register group XR2 is defined on the XR register file. It can be viewed as an 8-entry, 32-bit wide register file implemented on the XR register file. The structure ctype XR2 is defined on the register group XR2, whose fields are the base register file XR. The load, store and move protos of the ctype XR2 are also specified. The load and store instructions are endian dependent, because the memory interface width (32) is more than the base register file entry width (16). The protos `extract_a` and `extract_a_prime` are identical, as both extracts the first field of the struct ctype XR2.

5.4.2 Implementation Choices

If you add a register group to your regfile statement, you can choose to implement the register file in one of two ways by specifying an implementation keyword ("port" or "view"). Regardless of whether a register file is implemented as "port" or "view", the functionality of the register file will be the same. However, the area and timing of the resultant hardware implementation could be quite different depending on the implementation choice, the TIE operations that use that register file, and the FLIX formats and slots containing those operations.

For the port keyword, the TIE compiler implements the register file as a single register bank with multiple ports. In this implementation, reading a register pair or quad uses 2 or 4 ports of the register file to access 2 or 4 adjacent registers in the single bank.

For the view keyword, the TIE compiler implements the register file with wide ports and multiple register banks, where each bank contains half or a quarter of the number of entries of the register file. In this implementation, a register pair or quad will be read from the same register index of the multiple banks.

In general, the TIE compiler uses a complicated algorithm to assign operands from all operations in all FLIX slots to register file ports in the most efficient way. That is, to reduce the total number of ports required for that register file. If you don't specify a register group implementation keyword for your user-defined regfile, the TIE compiler will analyze both options and choose the implementation that requires the least number of register file ports. You can find out which choice it made by reading the TDK report file.

The TIE compiler's area estimator includes the cost in area due to the register group implementation choice. However, to see the timing impact, you need to build the full Xtensa hardware including your TIE extensions on the Xtensa Processor Generator, then synthesize the RTL for both implementation choices.

5.4.3 Boolean Register

The boolean register BR in the boolean option is a register file containing 16 1-bit entries. Several register groups are defined on the BR register file. BR2 contains eight 2-bit entries. BR4 contains four 4-bit entries. BR8 contains two 8-bit entries. BR16 contains one 16-bit entry. The register groups are implemented using the view implementation.

Users can use the register groups to access 2, 4, 8, or 16 bits of the boolean register in user-defined instructions. Each register group has a `ctype`, which is described in Section 15.2 on page 122.

5.5 Register File without Direct Bypass

When an instruction produces data to be written to a register file, the data enters the write pipeline of a write port on the register file. This data moves along the write pipeline and is written back to the actual register file a few cycles later. Subsequent instructions that need this data do not need to wait until the register file is updated. Hardware bypass paths on the write pipeline are used to forward the data before it is written back to the register file itself.

By default, register files are fully bypassed. That is, they have direct bypass hardware that forwards data entering the write pipelines of the register file to subsequent instructions immediately in the next cycle. This effectively creates the minimum read-after-write delay or latency between two data-dependent instructions. The direct bypass hardware may restrict the maximum attainable processor frequency because data generated in one instruction must be routed to all destinations in the same cycle.

When the register file has the no_bypass attribute, the register file is generated without this direct bypass hardware. The data entering the write pipelines of the register file is forwarded to subsequent instructions two cycles later. While this may increase the maximum attainable processor frequency, it may impact software performance because the minimum read-after-write delay between two data-dependent instructions is increased to two cycles. If a register file with register groups has the attribute no_bypass specified, the register groups do not have the direct bypasses either.

Designers can also specify the regbypass construct to selectively specify certain direct bypasses to reduce the read-after-write delay to one cycle on certain performance critical instruction sequences. Details of the regbypass construct can be found in Chapter 23, “Register File Bypass (regbypass) Sections” on page 195.

5.6 Implementation Restrictions

Following are the implementation restrictions for the regfile sections:

- The earliest use stage on every read port of a register file must be less than or equal to the latest def stage on the write ports.
- The names AR, BR, and MR are used for various Xtensa processor register files, and cannot be used for a designer-defined register file. Additional naming restrictions may be imposed in configurations that include various coprocessor packages.
- The following short names are not allowed for a designer-defined register file. Additional naming restrictions may be imposed in configurations that include various coprocessor packages.

```
a, b, i, l, L, m, r, sp, fp
```

- The register file name FR, and short name f, cannot be used in a configuration that includes the Floating Point Coprocessor option.
- The maximum number of entries supported in a single register file is 128. If more entries are desired, please contact the Cadence support team.
- The *width* of the register file is limited to a maximum of 4096 bits.
- If a register file contains register groups or two or four, the number of entries of the register file must be divisible by two or four.
- An Xtensa configuration can contain a maximum of 24 register files. This includes the predefined AR register file that exists in every Xtensa processor configuration, register files added as a result of user-selected configuration options (such as the Floating Point Unit Option, Boolean Register Option, Vectra DSP Engine Option or HiFi2 Audio Engine Option), and designer-defined register files.

- The total number of read ports of any register file cannot exceed 32. The total number of write ports cannot exceed 16. In addition, the number of read ports and write ports of a register file cannot exceed the number of entries of the register file.
- The identifier *name* is restricted to a length of 63 characters or less.
- The identifier *short-name* is restricted to a length of six characters or less.

6. Interface Signals (interface) Sections

Designer-defined TIE extensions can interact with the base Xtensa processor through a set of predefined interface signals. An interface signal can be directly used in `operation`, `iclass`, `reference`, or `semantic` sections without being previously declared[1].

6.1 Interface Signals Syntax

```
interface-def ::= interface name width core direction
name ::= one of the predefined interface signal names
width ::= number of bits in the interface signal
direction ::= direction of the interface signal
```

The complete list of predefined interface signals is shown in Table 6–3.

Table 6–3. Predefined Interface Signals

Name	Width	Direction[1]	Purpose
VAddr	32	out	Addresses for load/store instructions
MemDataIn512	512	In	The 512-bit load data
MemDataIn256	256	In	The 256-bit load data
MemDataIn128	128	in	The 128-bit load data
MemDataIn64	64	in	The 64-bit load data
MemDataIn32	32	in	The 32-bit load data
MemDataIn16	16	in	The 16-bit load data
MemDataIn8	8	in	The 8-bit load data
MemDataOut512	512	out	The 512-bit store data
MemDataOut256	256	out	The 256-bit store data
MemDataOut128	128	out	The 128-bit store data
MemDataOut64	64	out	The 64-bit store data
MemDataOut32	32	out	The 32-bit store data
MemDataOut16	16	out	The 16-bit store data
MemDataOut8	8	out	The 8-bit store data

1. “in” signals go from the base Xtensa processor to the TIE logic; “out” signals go from the TIE logic to the base Xtensa processor.

2. If the memory data width is more than 128 bits, the width of the byte disable is the memory data width divided by 8. For example, if the memory data width is 512 bits, the byte disable width is 64 bits.

1. An earlier version of the TIE compiler requires that all interfaces are declared before they can be used in operations.

Table 6–3. Predefined Interface Signals

Name	Width	Direction[1]	Purpose
LoadByteDisable	16[2]	out	The 16-bit byte disable signal for conditional load
StoreByteDisable	16[2]	out	The 16-bit byte disable signal for conditional stores
PIFAttribute	4	out	The 4-bit user defined control bits passed to PIF. It requires the “PIF Request Attributes” option to be selected in Xplorer.

1. “in” signals go from the base Xtensa processor to the TIE logic; “out” signals go from the TIE logic to the base Xtensa processor.

2. If the memory data width is more than 128 bits, the width of the byte disable is the memory data width divided by 8. For example, if the memory data width is 512 bits, the byte disable width is 64 bits.

6.2 Example

The following example declares the complete set of interface signals required for describing 32-bit load/store semantics. The *Tensilica Instruction Extension (TIE) Language User’s Guide* contains detailed examples of using these interface signals for creating load/store instructions.

```
interface VAddr 32 core out
interface MemDataIn32 32 core in
interface MemDataOut32 32 core out
```

6.3 Definition of Interface Signals

Interface signals between designer-defined TIE extensions and the base Xtensa processor are used in very specific ways. Each interface signal is associated with a specific pipeline stage of the Xtensa processor. The current implementation of the Xtensa processor executes instructions in five or seven pipeline stages. The pipeline stages are as follows:

1. Instructions are fetched in the I stage for a 5-stage pipeline. For a 7-stage pipeline, the instruction fetch operation is spread over the H and I stages.
2. Instructions are decoded in the R stage.
3. The E stage is when computations are performed, including the computation of the virtual address for load/store instructions.
4. Memories are read or written in the M stage. For a 7-stage pipeline, the memory access operation is spread over the L and M stages.
5. Results are written back in the W stage.

As noted above, the instruction fetch and data memory read/write stage get an extra cycle each for a seven stage pipeline For LX cores. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more information on the processor pipeline.

Because interface signals communicate information between the TIE extensions and the base Xtensa processor, it is important to understand the pipeline stage with which each signal is associated. This chapter contains a detailed definition of each interface signal, including the pipeline stage with which it is associated.

6.3.1 VAddr

VAddr is the virtual address for the designer-defined load/store instruction, and is associated with the E stage of the processor pipeline. In the absence of a memory management unit (MMU) in the Xtensa processor, this is also the physical address.

The Xtensa processor offers an option to generate an exception on any load or store instruction with an unaligned address. When this option is enabled, the value of VAddr must be aligned to the data size of the memory access for the load or store instruction to be successfully executed. Thus the address must be a multiple of 4 (bytes) for a 32-bit memory access, a multiple of 8 for a 64-bit memory access and so on. If the address is not aligned, the instruction will take an unaligned load/store exception. This option is the only allowed option for configurations with 256- and 512-bit memory accesses. Note that this behavior is different from the T1050 release in which TIE instructions never took an unaligned load/store exception. In the T1050 release, only the predefined Xtensa processor instructions took an exception on unaligned load/store addresses.

On an Xtensa processor configuration with the option to generate an exception on unaligned load/store disabled, the value of VAddr has no special alignment requirements. Note however, the special behavior of unaligned TIE loads under these conditions are described in Section 6.3.2. Note also the fact that under these conditions, TIE store instructions ignore an appropriate number of the least significant bits of VAddr such that the address is aligned to the write data width, as described in Section 6.3.3.

In general, any arbitrary expression using register, immediate, and state operands can be used to compute the value of VAddr. However, if the expression to generate this address follows certain rules, the TIE compiler recognizes the special form and marks the instruction with attributes that help the Xtensa C/C++ compiler use these instructions for certain optimizations. These restrictions are:

- The base address comes from an AR register
- The offset comes from any of the following:
  - another AR register, for an indexed load
  - an immediate operand or a field of the instruction

Following are the forms of expressions that satisfy the above conditions:

```
VAddr = ars + <imm>;// ars is an AR regfile operand
VAddr = ars - <imm>;
VAddr = ars + art;  // art is an AR regfile operand, used as a
```

```
// signed 32-bit offset
```

Load/store instructions used for various prototypes for designer-defined register files are required to use the restricted form of addressing described above. When the TIE compiler generates these prototypes, it adheres to this restriction. If you provide your own prototypes, you must use the restricted addressing modes for instructions used by these prototypes. Prototypes are explained in detail in Chapter 16, “Prototype (proto) Sections”.

6.3.2 MemDataIn<n>

This is the load data coming from the Xtensa processor load unit for data size <*n*>, and is available in the M stage of the processor pipeline. Any assignment using MemDataIn<*n*> must be scheduled to def 2 or greater for a 5-stage pipeline and def 3 or greater for a 7-stage pipeline. The variable <*n*> can take the values 8, 16, 32, 64, 128, 256, or 512, resulting in signals MemDataIn8, MemDataIn16, MemDataIn32, MemDataIn64, MemDataIn128, MemDataIn256, and MemDataIn512, respectively.

The load data returned on the MemDataIn<n> interface depends on certain Xtensa processor configuration options and the alignment of VAddr with the load data size as described below.

If the Xtensa processor configuration contains the option to generate an unaligned load/store exception on an unaligned load address, the load unit checks whether the value of VAddr aligns at the load data width. If the value is not aligned, an exception is generated. For example, if the load data interface is MemDataIn128, the load unit checks whether the least significant 4 bits of VAddr are zero. This option is the default configuration option for handling unaligned load/store operations in the Xtensa Processor Generator. If the maximum load data width is 256 or 512 bits, the only allowed configuration options are unaligned load/store exception and unaligned load/store handled by hardware.

If the "Handled by hardware" alignment option is selected, the load unit always loads data consisting of the next *n* bytes (which may span multiple memory locations) starting with the data at the (byte) address of the load. On certain unaligned VAddr addresses, the load unit may issue two memory requests.

If the "Align address" alignment option is selected, the behavior of the load unit is more complex. Conceptually, it takes the following steps to assign a value to this interface signal:

1. Compute the actual address from VAddr by ignoring a number of the least significant bits to align the actual address to the read width. For example, the 4 least significant bits are ignored for a 128-bit read width.

2. Load into a buffer *b* bytes from the aligned memory location where *b* is the read width in bytes.
3. Rotate the buffer contents to the right (left, for big-endian memory order) by the number of bytes specified by the VAddr bits ignored in step 1. In other words, rotate so that the byte specified by VAddr is first in the buffer.
4. Assign the rotated result to MemDataIn<*n*>.

This definition may seem to be more complex than necessary, but it is chosen to make loading unaligned data more efficient. If all the input data is aligned to the read width[1], ignore the above descriptions. MemDataIn<*n*> always has the right data. Note that this behavior of TIE load instructions is different than the load instructions in the core Xtensa instruction set: the core instructions do not perform the rotation described in step 3 above.

As an example, consider a little-endian Xtensa processor loading a 64-bit value from memory. Assume that the data at byte address 0 is 0xfedcba9876543210. A 64-bit load from byte address 3 would return the value 0x543210_fedcba9876. The data at address 0 is read from the memory, and then rotated right 3 bytes so that the data byte at address 3 would become the least significant byte of MemDataIn64.

As another example, consider a big-endian Xtensa processor loading a 32-bit value from memory. Assume that the data at byte address 0 is 0x01234567. A 32-bit load from byte address 1 would return the value 0x234567_01. The data at address 0 is read from memory, and then rotated left by 1 byte so that the data byte at address 1 would become the least significant byte of MemDataIn32.

6.3.3 MemDataOut<n>

MemDataOut<*n*> is the store data to the Xtensa processor store unit for data size <*n*>, and is required in the M stage of the processor pipeline. The variable <*n*> can take the values 8, 16, 32, 64, 128, 256, or 512, resulting in signals MemDataOut8, MemDataOut16, MemDataOut32, MemDataOut64, MemDataOut128, MemDataOut256, and MemDataOut512, respectively. The address of the store is specified by the interface signal VAddr as described in Section 6.3.1.

For Xtensa processor configurations with exception on unaligned store enabled, the processor will take an unaligned load/store exception if VAddr is not aligned to the write data width. For Xtensa processor configurations with the "Handled by hardware" option enabled, all the bits of VAddr are significant, and the hardware will store the data starting at the specified byte address, even if it means making multiple memory access. For

1. Aligning to the data memory access width is not necessary.

Xtensa processor configurations with "Align address" option enabled, the actual address of the store is the value of VAddr with a number of the least significant bits ignored such that the address is aligned to the write data width.[1]

6.3.4 LoadByteDisable and StoreByteDisable

LoadByteDisable specifies a byte mask to prevent the loading of certain bytes of MemDataIn<*n*> during memory read. StoreByteDisable specifies a byte mask to prevent the storing of certain bytes of MemDataOut<*n*> during memory write. Only the least significant <*n*>/8 bits of LoadByteDisable and StoreByteDisable are used, where n is the size of the load or store. The default value of both these interfaces is 0. LoadByteDisable is required in the E stage of the processor pipeline, while StoreByteDisable is required in the M stage.

On any data memory interface (including XLMI), these signal values are presented as byte enable signals with their polarity inverted. For load instructions, the data on the disabled bytes of the MemDataIn bus is always zero, irrespective of the data returned by the external memory or device.

It is legal to assign any arbitrary value to the LoadByteDisable and StoreByteDisable interface signals in a TIE description. However, not all byte enable values are supported by the Xtensa Processor Interface (PIF). The Xtensa microprocessor may break a single store request into multiple store requests to generate byte enable values that are supported by the PIF. For load transactions on the PIF, byte disables are essentially ignored; all the bytes corresponding to the size of the load request are enabled, except for the special case of a conditional load as described in Section 6.3.5. Refer to the *Xtensa LX Microprocessor Data Book* for specific restrictions on byte enable values on the PIF.

6.3.5 Conditional Load/Store Instructions

Setting LoadByteDisable/StoreByteDisable to all 1’s kills the respective load/store operation. If a load/store operation is killed, no data-address break point or load/store exception is generated, and no memory access is seen on the processor interface or local memory interface. When a load operation is conditionally killed with LoadByteDisable, the load data MemDataIn<*n*> has value <*n*>’b0.

It is important to remember that killing a load/store operation does not kill the instruction. Any operand and state output of the instruction will still be updated. To prevent writing a specific state or operand, the conditional write signal as explained in Section 7.4 “Conditional Writes to Arguments and States (Predication)” on page 43 must be used.

1. This definition of stores is not symmetrical to that of loads; that is, when a virtual address is not aligned to the data width, a load followed immediately by a store of the same data will actually change the memory contents.

The following example illustrates the use of LoadByteDisable to implement a conditional load instruction.

```
state LASTDATA 32

operation LOAD32 {out AR data, in AR *base_addr, in AR offset}
     {inout LASTDATA, out VAddr, out LoadByteDisable, in MemDataIn32}
{
     wire [31:0] address = base_addr + offset;
     wire lskill = (address[1:0] != 2'b0 );
     assign VAddr = address;
     assign LoadByteDisable = {4{lskill}};
     assign data = (lskill) ? LASTDATA : MemDataIn32;
     assign LASTDATA = (~lskill) ? MemDataIn32 : LASTDATA;
}
```

In this example, the load operation is killed if the address is not word aligned. This operation is killed by assigning the signal LoadByteDisable to 4'b1111 because the size of the load operation is 4 bytes. If the load is killed, the output register is updated with the value in state LASTDATA.

The following example illustrates the use of StoreByteDisable to implement a store instruction that writes either all 32-bits or only the least significant 16-bits from an AR register to memory.

```
state WR_DIS 1 add_read_write
operation ST32 {in AR data, in AR *base_addr} {in WR_DIS, out VAddr,
     out StoreByteDisable, out MemDataOut32}
{
     assign VAddr = base_addr;
     assign MemDataOut32 = data;
     assign StoreByteDisable = {{2{WR_DIS}}, 2'b00};
}
```

In this example, bits [1:0] of the StoreByteDisable signal are always 0, while bits [3:2] are determined by the value of the stage register WR_DIS. Thus this store operation will write all 32-bits if the value of WR_DIS is 0, and only the lower 16-bits if the value of WR_DIS is 1.

6.3.6 PIFAttribute

User-defined load and store instructions can drive an optional TIE interface PIFAttribute, which will be directly issued onto the user defined control bits of the PIF. The interface can be driven in the M-stage of a user-defined load or store instruction. PIFAttribute can only be driven from register file, state, or immediate values. To use this

interface, you must select the PIF3.2 configuration option in the Xtensa Processor Generator. Detailed description of PIF3.2 protocol and the usage of PIFAttribute interface can be found in the *Xtensa PIF Protocol Reference Manual*.

The following example considers a scenario in which the user wishes to define a posted and a non-posted write. The defined bits of the PIFAttribute are allocated as follows:

- PIFAttribute[0]: If 1’b1 then indicates that the request has additional "posted" control encoding in bit 1. Requests from core load and store instructions are issued with 1’b0.
- PIFAttribute[1]: If 1’b0 then the request is non-posted, else if 1’b1 then the request is posted. It is only valid if bit 0 is 1’b1.

Please note the system designer needs to decide whether PIF reads and writes from core loads and stores should be treated as posted or non-posted by default, and assign meaning to PIFAttribute bits.

Having assigned meaning to the PIFAttribute bits, the user defines the following posted and non-posted store instructions:

```
operation S32POSTED {in AR data, in AR addr} {out VAddr,
                     out MemDataOut32, out PIFAttribute} {
    assign VAddr = addr;
    assign MemDataOut32 = data;
    assign PIFAttribute = 4'b0011;
}

operation S32NONPOSTED {in AR data, in AR addr} {out VAddr,
                        out MemDataOut32, out PIFAttribute} {
    assign VAddr = addr;
    assign MemDataOut32 = data;
    assign PIFAttribute = 4'b0001;
}
```

7. Instruction Operation (operation) Sections

The operation section describes the name, format, and behavior of a single TIE instruction. It is typically the simplest and most compact way to describe a new instruction, and in some instances is the only section required to completely specify an instruction. This one construct specifies the name or mnemonic of the instruction, its assembly and C/C++ language syntax, and the computation performed by the instruction. It combines the functionality of the opcode, operand, iclass, and reference statements, thus providing an alternative and simpler way to describe a new instruction. While the opcode, operand, iclass, and reference statements are supported for backward compatibility reasons, the operation statement is recommended for new TIE development.

Note that it is illegal to have an iclass or reference description section for an instruction that is described using the operation construct. The *argument-list* and the *state-interface-list* of the operation replaces the iclass description, and the *computation* section of the operation replaces the reference description. It is optional to have an opcode declaration for an instruction described using the operation construct. If no opcode section is present, the TIE compiler automatically assigns the opcode encoding for the instruction.

7.1 Instruction Operation Syntax

```
operation-def ::= operation name argument-list
                        state-interface-list [{computation}]
name ::= a unique identifier
argument-list ::= { [dir [*] argument [, dir [*] argument]*] }
state-interface-list ::= { [dir state-interface-name
                                    [, dir state-interface-name]*] }
computation ::= see Chapter 28
dir ::= (in | out | inout)
argument ::= argument-type argument-name
argument-type ::= (operand-name | regfile-name | table-name |
                        immediate-range-name)
argument-name ::= an identifier unique to other argument-names
                        in this operation
operand-name ::= a previously defined operand name
regfile-name ::= a previously defined regfile name
table-name ::= a previously defined table name
immediate-range-name ::= a previously defined immediate_range name
state-interface-name ::= (state-name | interface-name)
state-name ::= a previously defined state name
interface-name ::= an interface name as described in Chapter 6
```

name is a unique identifier that is used as the name (or mnemonic) of the instruction. *argument-list* is a list of zero or more arguments that represent the register file or immediate operands of the instruction. *state-interface-list* is a list of zero or more arguments that specify the state and interface signal operands of the instruction. The `computation` section specifies the behavior of the instruction, as described in Section 28 “Computation Sections” on page 215. The `computation` section is optional if the instruction defined by this operation has a semantic description as described in Chapter 9, “Instruction Semantics (`semantic`) Sections” on page 61. However, it is recommended that a computation section be provided with the operation description, even if the instruction has a semantic description. TIE compiler generates formal verification scripts for the Encounter Conformal Equivalence checking tool from Cadence Design Systems. Use the scripts to verify that the operation description matches the semantic description.

7.2 Operation Arguments

The arguments of an `operation` represent the operands of the instruction being defined. Each entry of the argument-list specifies the direction and type of the corresponding operand, and a name to refer to that argument in the `computation` section. The direction of an argument specifies whether the corresponding operand is a source (`in`), destination (`out`), or both (`inout`) for the computation performed by the instruction. An argument’s type can be the name of a previously defined `operand`, `regfile`, `table`, or `immediate_range`. If the argument type is an operand name, it represents that operand. If the argument type is a regfile name, it represents an operand that accesses the register file. If the argument type is a table, it represents an immediate operand that can take any value from the table. If the argument type is an `immediate_range`, it represents an immediate operand that can take on any value in the immediate range. For operations that define load/store instructions, a pointer type is used for the argument that specifies the memory address. This is indicated with an asterisk (*) before the argument name. An operand from the predefined Xtensa AR register file is allocated for such an argument.

An operation may contain multiple operation arguments accessing the same register file. There is no restriction on the number of read and write operation arguments in one operation, provided the global register file port restriction is satisfied[1]INOUT.

The state-interface-list specifies the states and interface signals that can be accessed by the instruction. Each entry of state-interface-list specifies the direction and name of a state or interface signal. Note that state registers and interface signals are implicit operands of an instruction in the sense that they are not encoded in the instruction word and are not specified as operands in the assembly language syntax of the instruction.

1. Before release RE-2012.0 of the TIE compiler, one operation can only read and write three arguments of the AR register file.

The order of arguments in the argument-list is significant in that it determines the order of operands in the C/C++ intrinsic and assembly language syntax for the instruction. The order of arguments in a state-interface-list has no significance.

The assembly language syntax of an instruction lists the register and immediate operands of the instruction in the same order as the argument-list. The C intrinsic syntax of an instruction takes one of two forms:

- If the argument-list of an operation has one and only one register operand of type `out` (and not operands of type `inout`), its C intrinsic lists all the input operands from argument-list as arguments to the intrinsic, and the return value is assigned to the output operand. Note that if the output register operand uses the `_kill` feature for predication as explained in Section 7.4 on page 43, it is considered an `inout` from the compiler's point of view. Thus the C intrinsic follows the rule of the subsequent bullet.
- For all other cases, the C intrinsic syntax lists all the operands (of type `in`, `out`, and `inout`) from the argument-list as arguments to the intrinsic, and there is no return value of the intrinsic. This is illustrated in the examples that follow in Section 7.3.

An operation is invalid if the computation accesses any variables not declared in its argument-list or state-interface-list. It is also invalid if it accesses any of them in a manner inconsistent with their declaration as an `in`, `out`, or `inout` operand. An empty argument list indicates that the instruction does not access any register file or immediate operands. An empty state-interface-list indicates that the instruction does not access any states or interfaces.

7.3 Examples with Assembly and C Syntax

The following operation describes an instruction with two arguments in the *argument-list*. The first argument represents an operand that reads from and writes to the predefined Xtensa processor address register file referred to by its name AR. The second argument represents an operand that reads from the AR register file. The instruction does not access any states or interfaces, and hence has an empty state-interface-list.

```
operation ADDACC {inout AR a, in AR b} {} {
     assign a = a + b;
}
```

The assembly and C syntax of the ADDACC instruction is illustrated below. The assembly syntax shows two arbitrarily chosen AR register entries as operands of the instruction. Note that the first operand is of type `inout`. The C syntax assumes that we are operating on variables of type `int`, and is similar to the assembly syntax. Note that both operands are listed as arguments to the intrinsic, and the intrinsic has no return value. This is

because the first argument of the operation is of type inout. Thus the variable acc is passed as an argument to the C intrinsic, even though it is modified by the call. This behavior is similar to references in C++, but is a non-standard behavior for C programs.

```
#Assembly language syntax
ADDACC a4, a5       # a4 = a4 + a5

// C language syntax
#include <xtensa/tie/filename.h>
int acc, a;

ADDACC(acc, a);     // acc = acc + a
acc = ADDACC(a);    // ERROR: acc is of type inout
```

The following operation uses a table name as the type for two different arguments. Thus the instruction takes as input two values from the table. The operation also has one argument representing an operand that writes to the AR register file.

```
table TBL 32 4 {1, 5, 7, 12}
operation SUBT {out AR x, in TBL y, in TBL z} {} {
     assign x = y - z;
}
```

Following is the assembly and C language syntax of the SUBT instruction. Note that the table values (and not the table indices) are used as the immediate operands of the instruction. Because this instruction has only one operand of type out, the C intrinsic has a return value that is assigned to the variable that represents the output.

```
# Assembly language syntax
SUBT a7, 12, 7      # a7 = 12 - 7
SUBT a7, 0, 1       # ERROR: 0 is not a legal operand. Table values
                    # and not table indices are valid operands

// C language syntax
#include <xtensa/tie/filename.h>
int diff;

diff = SUBT(5, 12);      // diff = 5 - 12;
SUBT(diff, 5, 12);       // ERROR: diff should not be listed
                         // as in input argument of the intrinsic
```

The following operation uses an immediate_range name as the type of an argument. The operation also has two arguments representing operands that access the AR register file, and one argument representing a table value.

```
immediate_range I3 0 7 1
table TBL 32 4 {1, 5, 7, 12}
operation ADDTI {out AR a, in AR b, in TBL c, in I3 d} {} {
     assign a = b + c + d;
}
```

The assembly and C language syntax of the ADDTI instruction follows.

```
# Assembly language syntax
ADDTI a15, a8, 12, 2# a15 = a8 + 12 + 2
ADDTI a7, a6, 5, 12 # ERROR: Last operand should be a value from the
                    # immediate range (values between 0 and 7)

// C language syntax
#include <xtensa/tie/filename.h>
int sum, in0;

sum = ADDTI(in0, 7, 2);  // sum = in0 + 7 + 2;
sum = ADDTI(1, in0, 6);  // ERROR: incorrect order of operands
```

The following operation uses a designer-defined register file and a state register. While the register file operand is specified in the *argument-list*, the state operand is specified in the *state-interface-list*.

```
regfile NR 64 16 n
state ACCUM 32
operation ACCNT {in NR reg} {inout ACCUM} {
     assign ACCUM = ACCUM + reg;
}
```

The assembly and C language syntax of the ACCNT instruction follows. The short name n of the register file is used in the assembly syntax. The long name NR is used as the type of the variable in the C language syntax. The state ACCUM is an implicit operand of the instruction in the sense that it is not mentioned in the assembly or C syntax of the instruction.

```
# Assembly language syntax
ACCNT n7            # ACCUM = ACCUM + n7
ACCNT n3, ACCUM     # ERROR: State operand is not used in
                    # the assembly language syntax

// C language syntax
#include <xtensa/tie/filename.h>
NR sum;

ACCNT(sum);         // ACCUM = ACCUM + sum;
ACCUM = ACCNT(sum)  // ERROR: State operand is not used in the
                    // C language syntax
```

The following operation defines an instruction that loads a 16-bit value from memory to a designer-defined register file XR. The example uses the interface signals VAddr, which represents the memory address, and MemDataIn16, which represents the 16-bit read data from memory. These interface signals are listed in the *state-interface-list* of the operation and are described in Chapter 6, “Interface Signals (interface) Sections” on page 29.

```
regfile XR 16 16 x
operation LDXR {out XR x, in XR *addr} {out VAddr, in MemDataIn16} {
     assign VAddr = addr;
     assign x = MemDataIn16;
}
```

Note that the argument addr is used to generate the memory address, and that it is of type “pointer to XR.” When defining a load or store instruction, the operation should use a pointer type to refer to the argument representing the memory address. Using a pointer type causes the corresponding C intrinsic to accept the appropriate ctype. The TIE compiler uses an AR register file operand to represent this argument.

The assembly and C language syntax of the LDXR instruction follows. The assembly syntax shows that the second operand of this instruction (memory pointer) is an AR register file operand. The C language syntax shows that the input operand of the intrinsic is of type “pointer to XR.” The interface signals VAddr and MemDataIn16 are not mentioned in the assembly or the C syntax of the instruction.

```
# Assembly language syntax
LDXR x3, a4    # x3 = MEMORY[a4]

// C language syntax
#include <xtensa/tie/filename.h>
XR value_array = [2, 4, 6, 8, 10, 12];
XR ldreg;

ldreg = LDXR(&value_array[3]);     // lgreg = 8
```

The following operation statement defines an instruction that operates only on designer-defined states.

```
state COUNT    16
state ROUND     8   add_read_write
operation INIT {} {inout COUNT, out ROUND} {
     assign ROUND = 8'hFF;
     assign COUNT = COUNT + 1;
}
```

Because the operation INIT does not have any register or immediate operands, its assembly or C intrinsic does not have any arguments as shown below. This is consistent with it having an empty *argument-list*. Note that an *argument-list* is always required, even if it is an empty list. This operation uses two designer-defined states that are declared in the *state-interface-list*.

```
# Assembly language syntax
INIT           # No operands for this instruction

// C language syntax
#include <xtensa/tie/filename.h>
INIT();        // No operands for this instruction
```

The following operation statement has a *state-interface-list* that specifies the state FOO as an output operand of the instruction. The computation section of the operation also accesses the interface signals VAddr and MemDataIn32, which are not declared in the *state-interface-list*. Thus this is an invalid operation description and results in an error message from the TIE compiler.

```
state FOO 32
operation LDFOO {in AR *addr} {out FOO} {
     assign VAddr = addr;
     assign FOO = MemDataIn32;
}
```

The following operation statement declares the argument sum to be of type out. However, the operation body uses it as an inout operand. Thus the declaration of the argument is inconsistent with its use, and this is signaled as an error by the TIE compiler.

```
operation WRONG {out AR sum, in AR in0} {} {
     assign sum = sum + in0;  // Error: sum cannot be used as input
}
```

7.4 Conditional Writes to Arguments and States (Predication)

If an instruction only produces a new value for an argument or state when certain conditions are met, the argument or state should still be declared as a destination (out or inout) in the *argument-list* or the *state-interface-list* of the operation. The TIE language provides a mechanism to specify the exact condition under which the assignment of the new value may be suppressed, as part of the *computation* section of the operation. When an argument or state xyz is declared as out or inout, it becomes a legal variable to which a value can be assigned in the *computation*. Likewise, a single bit signal xyz_kill is also a legal variable such that any assignment to xyz is suppressed if xyz_kill is true.

In the following example, the instruction OPKILL_A assigns the value of input argument in0 to the output argument out0, when the input argument in1 is non-zero. If in1 is equal to zero, out0 should retain its old value.

```
regfile XR 32 16 x
operation OPKILL_A {out XR out0, in XR in0, in XR in1} {} {
     assign out0 = in0;
     assign out0_kill = (in1 == 0);
}
```

It is possible to write such an operation without using the _kill signal by making a conditional assignment to the output argument. However, this can create false data dependencies in hardware, or create additional read ports on register files as illustrated below:

```
regfile XR 32 16 x
operation OPKILL_B {inout XR out0, in XR in0, in XR in1} {} {
     assign out0 = (in1 == 0) ? out0 : in0;
}
```

The definition of instruction OPKILL_B requires all three of its operands to be read from the register file XR, while the instruction OPKILL_A requires only two of its register operands to be read. Thus, while operations OPKILL_A and OPKILL_B perform the same function, OPKILL_B requires three read ports on the XR register file while OPKILL_A requires only two. Furthermore, OPKILL_B has a data dependency on the argument out0, which can result in extra stall cycles in situations where it is preceded by an instruction that writes to out0.

The preceding example illustrates why it is advantageous to use the _kill signal instead of declaring the corresponding operand to be of type inout. The advantages are hardware implementation specific; avoiding unnecessary hardware interlocks (stalls) and unnecessary read ports on register files. From the C compiler standpoint, OPKILL_A and OPKILL_B are equivalent in the sense that the compiler cannot remove an earlier instruction that writes to the same entry of the XR register file (elimination of dead code). In order to model this, the instruction prototype for both OPKILL_A and OPKILL_B treat out0 as an inout variable. Thus the C intrinsic for both OPKILL_A and OPKILL_B have the same syntax as follows:

```
void OPKILL_A(XR out0 /* inout */, XR in0, XR in1);
void OPKILL_B(XR out0 /* inout */, XR in0, XR in1);
```

7.5 Partial Conditional Writes to Arguments

It is not uncommon to partially update an argument in SIMD instructions. The TIE language provides a mechanism to specify the exact condition under which the assignment of the new value of an output argument is partially suppressed. When an argument xyz

is declared as an output operand of an instruction, the bit kill signal xyz_bitkill of the same width as xyz is implicitly declared. Each bit in xyz_bitkill controls whether the corresponding bit in xyz is updated. If a bit in xyz_bitkill is zero, the corresponding bit in xyz is updated with the new value. If a bit in xyz_bitkill is one, the corresponding bit in xyz preserves its original value. The bit kill signal xyz_bitkill only takes effect when the kill signal xyz_kill is not assigned to value one.

In the following example, the instruction XR_partial_write partially assigns the value of the input argument b to the output argument a. The assignment is suppressed when the highest bit of input argument k (k[31]) is one. In the case that k[31] is zero, the lowest byte of the output argument a always retains its original value; the second lowest byte is always assigned to the value of the second lowest byte of the input argument b; the highest and second highest byte of a are assigned to new values when k[3] and k[2] have value zero, respectively, otherwise they retain the original values.

```
regfile XR 32 16 xr
operation XR_partial_write {out XR a, in XR b, in AR k} {} {
   assign a = b;
   assign a_kill = k[31];
   assign a_bitkill = {{8{k[3]}}, {8{k[2]}}, 8'b0, 8'hff};
}
```

Many SIMD instructions divide a wide operand argument into multiple data elements, with each data element updated independently. The bit kill signal is divided into slices, with each slice corresponding to one data element. All bits in one slice are assigned to the same value, but bits in different slices are assigned to different values. With this characteristics, the TIE compiler imposes the following restrictions to the assignment of the bit kill signals, in order to calculate the data element width and reduce the hardware cost.

- The expressions assigned to the bit kill signals need to be concatenations, or a TIEsel built-in module whose data inputs are concatenations.
- The concatenation needs to be composed of slices of equal size.
- In the case that the bit kill behavior of one slices is determined at runtime, the slice needs to be written as a replication of a one bit wire.
- In the case that the bit kill behavior of one slice is determined at design time, the slice needs to be written as a constant. All bits in the constant need to have the same value.

The following example illustrates the usage of the assignment to the bit kill signals:

```
regfile XR 24 16 xr

operation pw1 {out XR a, in XR b, in AR k} {} {
       assign a = b;
       assign a_bitkill = {{12{k[1]}}, {12{k[0]}}};
```

```
}

operation pw2 {out XR a, in XR b, in AR k} {} {
       assign a = b;
       assign a_bitkill = {{8{k[2]}}, {8{k[1]}}, {8{k[0]}}};
}

operation pw3 {out XR a, in XR b, in AR k} {} {
       assign a = b;
       assign a_bitkill = {6'b111111, 6'b00000, {6{k[1]}}, {6{k[0]}}};
}

semantic pwsem {pw1, pw2} {
      assign a = b;
      assign a_bitkill = TIEsel(pw1, {{12{k[1]}}, {12{k[0]}}},
                                pw2, {{8{k[2]}}, {8{k[1]}}, {8{k[0]}}});
}
```

In the above example, three instructions are defined, `pw1`, `pw2`, and `pw3`. They divide the write to the output argument `a` into 2, 3, and 4 slices, and conditionally writes each slice based on the value assigned to the `a_bitkill` signal. The TIE compiler calculates the data element width of the register file `XR` to be the greatest common divisor of the data element width of all instructions that access `XR`, which is two bits. Register files with small data element widths usually require more hardware logic than register files with large data element widths. The data element width of a user register file that implements the partial conditional writes can be found in the report file generated by the TIE compiler.

All register files except `AR` and `BR` can be augmented to be written conditionally by creating TIE operations that conditionally write to the arguments of the register file. States cannot be written conditionally in an operation by assigning the bit kill signal to the states. Users who desire this behavior can implement the operations that read in the original value of the states and write out the new value of the states.

8. Schedule (schedule) Sections

A schedule section provides a mechanism for the designer to define the pipeline stages in which input operands of a TIE instruction are read and output operands are computed. In general, the TIE language allows instruction descriptions at the ISA (Instruction Set Architecture) level; the schedule section is an exception. It specifies the implementation at the micro-architectural level.

The most common use of a schedule section is to define multi-cycle TIE instructions—those that take more than one clock cycle to complete.

8.1 Schedule Syntax

```
schedule-section ::= schedule name opcode-list { usedef [usedef]* }
name ::= unique name for the schedule
opcode-list ::= { opcode-name [, opcode-name]* }
opcode-name ::= a previously defined opcode or operation name
usedef ::= used | defd
used ::= use use-name stage;
defd ::= def def-name stage;
use-name ::= name of an operand, state or operation argument.
def-name ::= name of an operand, state, operation argument or wire
stage ::= integer | symbolic
integer ::= an integer specifying the pipeline stage
symbolic ::= symbolic-stage-name [+|- integer constant]
symbolic-stage-name ::= Rstage | Estage | Mstage | Wstage
```

name is a unique identifier used to identify a schedule. *opcode-name* is a previously defined opcode name. If an instruction is defined using the operation construct, *opcode_name* is the same as the operation name. One or more opcodes with the same input and output specifications can be grouped into a schedule.

One or more use statements are used to specify when the state, operand, or operation argument can be read. One or more def statements specify when the state, operand, or operation argument can be written to. It is also legal to specify the def stage of wires declared within the instruction description. However, use stages cannot be specified for wires. Note that instructions defined using the operation construct use argument names inside schedule constructs, while instructions defined with the iclass construct use operand names. Beyond this, the terms *operation argument* and *operand* is used interchangeably in this chapter.

stage is an integer specifying the number of the pipeline stage in which each operand of an instruction is read from or written to. A stage value of 0 is defined to represent the decode stage (R stage) of the processor, while a stage value of 1 represents the execution stage (E stage) of the processor. Most Xtensa ISA ALU instructions read and write their operands in stage 1. For a 5-stage pipeline, stage 2 refers to the memory read/write stage (M stage), whereas for a 7-stage pipeline, stage 3 refers to the memory read/write stage. Figure 8–1 shows the integers associated with the different stages of a 5-and 7-stage Xtensa processor.

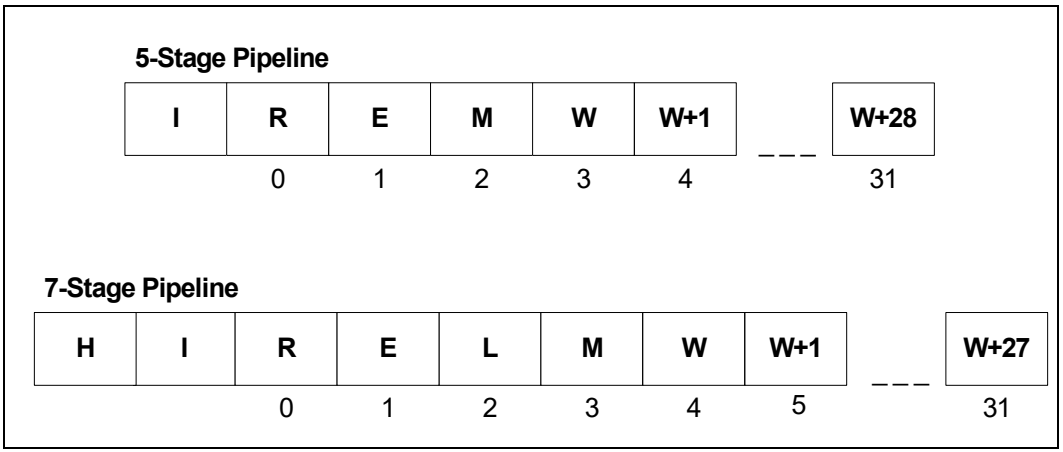

Figure 8–1. Example of 5- and 7-Stage Pipeline Schedules

For LX cores **Note:** The 7-stage pipeline illustrated in Figure 8–1 is not available in Xtensa 8.

stage can also be specified as a symbolic name or an expression. A symbolic name is a keyword for pipeline stages in the Xtensa processor. The valid symbolic stage names are `Rstage`, `Estage`, `Mstage`, and `Wstage`. All other stages can be expressed as an expression with an integer offset from a symbolic stage. For example, in Figure 8–1, the stage `Estage+1` refers to the M stage in a 5-stage pipeline and the L stage in a 7-stage pipeline. The use of symbolic stages improves the portability of the TIE description to Xtensa processors of different pipeline depths. For example, a load instruction should define its results in `Mstage` rather than 2 or 3. On the other hand, a 2-cycle ALU instruction not dependent on memory should have a def stage of 2 or `Estage+1` rather than `Mstage`.

8.2 Examples of Schedules

Following is a simple example of the schedule construct:

```
operation ADD16 {out AR sum, in AR a, in AR b} {} {
     wire [15:0] t0 = a[15: 0] + b[15: 0];
```

```
     wire [15:0] t1 = a[31:16] + b[31:16];
     assign sum = {t1, t0};
}

schedule add16_sched {ADD16} {
     use a Estage; use b Estage; def sum Estage;
}
```

In the above example, the ADD16 instruction has three operands from the AR register file. Operation arguments a and b are inputs while, sum is an output. The schedule construct specifies the pipeline stage in which input arguments are read and output arguments become available. Thus the use Estage schedule for the input arguments means that arguments a and b are read by the instruction semantic early in the E stage of the processor pipeline. The def Estage schedule for the output arguments means that the argument sum is computed late in the E stage of the processor pipeline. The schedule can also be written using numeric stages instead of symbolic stages as follows:

```
schedule add16_sched {ADD16} {
     use a 1; use b 1; def sum 1;
}
```

Consider the following TIE code that defines a 32-bit register file, an updating load instruction for the register file and the instruction schedule:

```
regfile XR 32 16 x
immediate_range IMR 0 124 4

operation LDXR {out XR reg, inout XR *addr, in IMR offset}
              {out VAddr, in MemDataIn32} {
     wire [31:0] tmp = addr + offset;
     assign VAddr = tmp;
     assign addr = tmp;
     assign reg = MemDataIn32;
}

schedule foo {LDXR} {
     use offset Rstage;
     use addr Estage;
     use MemDataIn32 Mstage;
     def VAddr Estage;
     def addr Estage;
     def reg Mstage;
}
```

The address for the load is formed by adding the base address (addr) to an immediate value (offset). Immediate values are encoded in the instruction word, and thus available in the instruction decode stage or the R stage of the pipeline. Thus the argument

offset has been assigned a use Rstage schedule. The base address comes from the AR register file, and the earliest that a register value is available is the E stage of the pipeline. Thus the base address argument has been assigned a use Estage schedule. The other input is MemDataIn32, which is an interface signal used to read load data from memory. This interface signal has a fixed use stage as described in Section 6.3.2, thus it has a use Mstage schedule.

VAddr is an output interface in which the address of the load is presented to memory. Because VAddr is an interface, it also has a fixed use stage as described in Section 6.3.1, and thus has a def Estage schedule. This is an updating load instruction, so the base address is updated with the incremented value, thus the argument addr is an inout operand. An inout argument has both a use and a def stage in the schedule. The use stage for addr was Estage as explained previously. In this example, its def stage is also the Estage. The XR register argument reg is the register to which the load data is written. The load data is available in the Mstage and so the schedule for the reg operand is def Mstage.

While all the operation arguments of the ADD16 instruction were read and computed in the same pipeline stage, this is not the case with the LDXR instruction; different input and output arguments of the LDXR instruction are read and computed in different pipeline stages.

8.3 Default Schedules

Every input operand of a TIE instruction has a use stage, which is the pipeline stage in which that operand value is read. Similarly, every output operand of a TIE instruction has a def stage, which is the pipeline stage in which that operand value is computed. The use and def stages of operands can be explicitly specified in the schedule as illustrated in the examples in Section 8.2. It is however not necessary to specify the use and def stage of every operand of every TIE instruction. This is because every operand has a default use and/or def stage, and if a schedule is not specified, the default use/def stage is assumed. It is thus legal to not have a schedule section for an instruction, or to have a partial schedule—one in which the use/def stages for some (but not all) of the operands of an instruction are specified.

The use stage of all immediate operands is the R stage. Immediate operands are encoded in the instruction word, and are thus available in the instruction decode stage. Thus, Rstage is not only the default use stage for immediates, it is the only use stage that is allowed. As a result, you do not need to specify a schedule for an immediate operand of a TIE instruction.

All TIE interfaces have a fixed use and def stage; the pipeline stage in which they can be accessed is predetermined and cannot be changed. TIE interfaces such as VAddr, MemDataIn, MemDataOut, LoadByteDisable, and StoreByteDisable are used to

define load/store instructions. All of these interfaces have a fixed use/def stage as described in Chapter 6, Interface Signals (interface) Sections. TIE ports and queues also have fixed use and def stages as summarized in Table 8–4 on page 58. The use and def stages for a TIE lookup are as defined in the lookup definition. They can change from one lookup to another, but are fixed for a given lookup.

All register and state operands can be read and written in or after stage 1 or Estage. The default use stage of a register or state operand is 1, unless it is directly assigned to an output interface, in which case the default use stage is the same as the def stage of the interface. Similarly, the default def stage of a register or state operand is 1, unless it is the target of an interface, in which case the default def stage is the same as the use stage of the interface. This is illustrated in the following example:

```
regfile YX 32 8 yx
immediate_range IMR 0 124 4

operation STORE_YX {in YX reg, in YX *addr, in IMR offset}
                {out VAddr, out MemDataOut32} {
     assign VAddr = addr + offset;
     assign MemDataOut32 = reg;
}

operation LOAD_YX {out YX reg, inout YX *addr, in IMR offset}
                {out VAddr, in MemDataIn32} {
     wire [31:0] newaddr = addr + offset;
     assign VAddr = newaddr;
     assign addr = newaddr;
     assign reg = MemDataIn32;
}
```

In the above example, ST_YX is a store instruction, while LD_YX is a load instruction. No schedule is specified for either instruction, so all operands take their default schedules. For the ST_YX instruction, the operation argument reg is an input which is assigned to the MemDataOut32 interface. Thus, the default use stage of reg is the same as the def stage of MemDataOut32, which is Mstage. Note that if the assignment to MemDataOut32 is an expression involving reg (instead of a direct assignment of reg), then the default use stage of reg would continue to be Estage. Only in the case of a direct assignment is the use stage adjusted to match the def stage of the interface. The term *direct* assignment includes assignment of concatenation, repetition of bits or bit slices of the register. Thus the following expressions would also qualify as direct assignment to an interface:

```
assign MemDataOut32 = {16'b0, reg[15:0]}
assign MemDataOut32 = {{16{reg[15]}}, reg[15:0]}
assign MemDataOut32 = {reg[15:0], reg[31:16]}
```

For the LD_YX instruction, the operation argument reg is an output and addr is an in-out. The def stage of addr is Estage. However, the def stage for reg is Mstage instead of Estage, because it is the target of the interface MemDataIn32.

Note again that the rules for default schedules apply only when no schedule is specified in the TIE description. If you provide a schedule for an instruction, it is always followed independent of the default schedule rules.

8.4 Single-Cycle TIE Instructions

Consider the operation and schedule description of the instruction SUB20 in the following description. The instruction operates upon a designer-defined register file named VR.

```
regfile VR 64 8 v
operation SUB20 {out VR diff, in VR in0, in VR in1} {} {
     assign diff = in0 - in1;
}
schedule sub20 {SUB20} {
     use in0 1; use in1 1; def diff 1;
}
```

For this instruction, the use stage of both input operands is 1, and the def stage of the output operand is also 1. The input operands are available at the beginning of the first clock and the output is computed at the end of the first clock. Thus the computation is performed in one clock cycle. Such instructions are referred to as single cycle instructions.

As explained in Section 8.3, the default use/def stage of register operands is 1 (or Estage), and thus it is not necessary to specify a schedule for the instruction SUB20. The schedule statement for the SUB20 instruction as defined in the previous example is optional, and could be left out entirely. If the instruction SUB20 was defined with a schedule such that the input operands in0 and in1 were read in a later cycle (for example, 3) and the output operand diff was defined in the same cycle (cycle 3), it would still be a single cycle instruction. However, in this case the schedule construct would be required because the operands no longer use the default use/def stages.

Note that if the instruction SUB20 was defined using the iclass construct instead of the operation construct, the schedule statement uses the operand names from the iclass section instead of the argument names from the operation section. The following description shows that vr, vs, and vt represent register operands of the designer-defined register file names VR.

```
iclass sub20 {SUB20} {out vr, in vs, in vt}
```

```
schedule sch_sub20 {SUB20} {
     use vs 1; use vt 1; def vr 1;
}
```

8.5 Multi-Cycle TIE Instructions

The computations described in a TIE instruction can span multiple clock cycles, and such instructions are referred to as multi-cycle instructions. Multi-cycle TIE instructions are defined as instructions for which the def stage of an output operand is greater than the earliest use stage of any input operands used to compute its value. Consider the following operation and schedule description of the MULA16 instruction:

```
regfile VR 64 8 v
operation MULA16 {inout VR accum, in VR m0, in VR m1} {} {
     assign accum = accum + m0 * m1;
}
schedule vec_mac {MULA16} {
     use m0 1; use m1 1; use accum 2; def accum 2;
}
```

This schedule shows an instruction that uses m0 and m1 on the first clock and uses accum on the second clock. The result is then written into accum at the end of the second clock, as shown in Figure 8–2. Thus the computation described by the instruction spans over more than one cycle, making MULA16 a multi-cycle instruction. Note that multi-cycle instructions are fully pipelined, so they can be issued every cycle even though each instruction takes more than one cycle to complete.

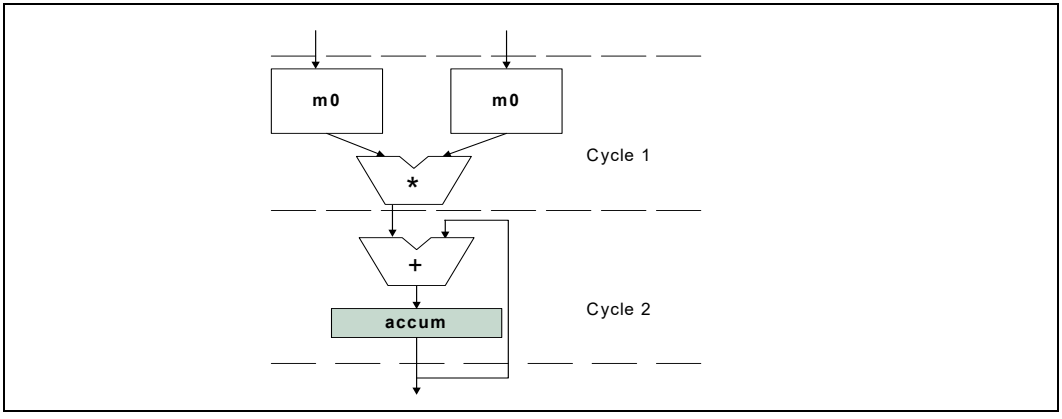

Figure 8–2. 2-Cycle MAC Operation

8.5.1 Pipeline Stalls Due to Multi-Cycle Instructions

Multi-cycle TIE instructions can cause data dependencies in the program flow that result in pipeline stalls. Based on the schedule description of TIE instructions, the TIE compiler generates a hardware implementation that causes stalls in the processor pipeline when required. Consider the following sequence of code using the instructions MULA16 and SUB20 defined earlier:

```
MULA16 v0, v1, v2
SUB20 v5, v3, v0
```

In the preceding case, there is a data dependency between the two instructions. The result v0 of the instruction MULA16 is an input to the next instruction SUB20. Because the result vr of the instruction MULA16 has def 2, and the input vt of instruction SUB20 has use 1, this instruction sequence causes the instruction SUB20 to stall for a cycle. Now consider two back-to-back MULA16 instructions as follows:

```
MULA16 v0, v1, v2
MULA16 v0, v3, v5
```

In this case, the data dependency is due to the “accumulator” output v0 of the first MULA16 instruction being read as an input of the second MULA16 instruction. However, this does not cause any pipeline stall. The input accum of instruction MULA16 has a use 2 and def 2 in the schedule. Therefore, the second MULA16 instruction does not read v0 until the second execution cycle and thus does not need to stall.

Multi-cycle TIE instructions with late def stages may result in several cycles of pipeline stalls, or may result in a pipeline stall several cycles later. In a simplified scenario, if an instruction with a “def n” on an operand is immediately followed by an instruction with a “use m” of the same operand (n > m), the processor will stall for (n - m) cycles. Note that to accurately compute the number of stall cycles for any particular instruction, you need to consider the dependencies of all of its input operands on the output operands of several preceding instructions. Such a computation, especially in the context of a FLIX design, is implementation dependent and can become quite complicated.

8.5.2 Multi-Cycle Load Instructions

The following example illustrates a simple load instruction for a designer-defined register file:

```
immediate_range IMMR 0 60 4
regfile ZR 32 16 z
operation LDZR {out ZR zreg, in ZR *addr, in IMMR offset}
               {out VAddr, in MemDataIn32} {
     assign VAddr = addr + offset;
     assign zreg = MemDataIn32;
}
```

As explained in Section 8.3, the default def stage for zreg is Mstage because it is the target of the interface MemDataIn32, which is available in Mstage. In this example, there is a simple assignment of MemDataIn32 to zreg, for which this default schedule is fine. Load instructions that manipulate the incoming memory data before assigning it to a state or register file may require a def stage later than Mstage. This is especially true if the target frequency of operation for the hardware is aggressive. In the following example, a value from the AR register file is added to MemDataIn32 before being assigned to zreg. Because the interface signal MemDataIn32 becomes available late in the M stage of the processor pipeline, there may not be enough time to perform the addition in the same cycle. Thus, the schedule for this operation allows an extra clock cycle to perform the addition, as follows:

```
regfile ZR 32 16 z
operation LOADZ{out ZR zreg, in ZR *addr, in AR in0}
               {out VAddr, in MemDataIn32} {
     assign VAddr = addr;
     assign zreg = MemDataIn32 + in0;
}

schedule sch_loadz {LOADZ} {
     def zreg Mstage+1;
}
```

This schedule makes the load instruction LOADZ a multi-cycle instruction. This is necessary if there is any logic in the path of the MemDataIn interface before it is assigned to a register and you are targeting an aggressive clock frequency for your design.

8.5.3 Synthesis of Multi-Cycle Instructions

When a TIE description contains multi-cycle instructions, the TIE compiler generates functionally correct RTL implementations of the instructions by inserting appropriate flip-flops. However, it relies on the synthesis tools, such as Design-Compiler from Synopsys, to move the flip-flops around to balance the delays to achieve optimal performance. Specifically, if the Design-Compiler is being used for synthesis, a DC-Ultra license is necessary to enable the behavioral retiming feature. However, retiming the RTL may impose difficulties on formal verifications of the RTL and gate implementations. It is possible to synthesize designs with multi-cycle TIE instructions without using behavioral retiming as explained in Section 8.6.

8.6 Assigning a def Stage to Wires

In general, a schedule section specifies the use and def stages for the input and output operands of an instruction. For multi-cycle TIE instructions, it is not necessary to specify the position of the pipelining flip-flops in the instruction semantics. The RTL implementa-

tion relies on behavioral retiming to appropriately position these flip-flops during the synthesis process. It is also possible to explicitly specify the position of these flip-flops by specifying def stages for intermediate wires of a computation.

Consider the problem of creating an instruction that adds four 32-bit numbers and returns the result in an AR register. The following TIE code shows one implementation for such an instruction.

```
state IN0 32 add_read_write
state IN1 32 add_read_write
state IN2 32 add_read_write
state IN3 32 add_read_write

operation SUM4 {out AR sum} {in IN0, in IN1, in IN2, in IN3} {
     assign sum = TIEaddn(IN0, IN1, IN2, IN3);
}

schedule sum4 {SUM4} {
     use IN0 1; use IN1 1; use IN2 1; use IN3 1;
     def sum 2;
}
```

This implementation of the SUM4 instruction uses the TIE built-in module TIEaddn to sum up the four 32-bit state values. To prevent this computation from becoming the critical timing path in the design, SUM4 is defined to be a two-cycle instruction. The RTL implementation of this instruction will need behavioral retiming to break the combinational logic of the computation into two halves and position the pipeline flops in-between. An alternative implementation of the SUM4 instruction follows:

```
operation SUM4 {out AR sum} {in IN0, in IN1, in IN2, in IN3} {
     wire [31:0] s0, c0, s1, c1;
     assign {c0, s0} = TIEcsa(IN0, IN1, IN2);
     assign {c1, s1} = TIEcsa(IN3, (c0 << 1), s0);
     assign sum = (c1 << 1) + s1;
}

schedule sum4 {SUM4} {
     use IN0 1; use IN1 1; use IN2 1; use IN3 1;
     def c1 1; def s1 1;
     def sum 2;
}
```

In this implementation, the summation is broken up into two carry-save adders followed by a full adder. The carry and sum outputs of the second carry-save adder are assigned to wires c1 and s1. The schedule for this instruction assigns a def stage of 1 to wires c1 and s1. This implies that wires c1 and s1 are flopped (registered) before being used in any subsequent computation. As a result of this schedule, the RTL implementation per-

forms the two carry-save additions in the first cycle of the instruction and the result of this computation is flopped. The full adder used to compute the final result sum operates upon the registered values of `c1` and `s1`, and thus is scheduled for the second cycle of the instruction.

It is not necessary to provide schedules for all the wires of a computation. Thus in the preceding example no schedule is provided for wires `c0` and `s0`; the TIE compiler automatically determines their schedule.

By default, the TIE compiler marks every multi-cycle TIE instruction as a candidate for behavioral retiming. Even if this instruction has a schedule that positions the pipeline flops based on the TIE designer's estimate, retiming may be able to achieve better timing by moving the position of these flip-flops. Thus, it is advisable to run all multi-cycle TIE instructions through the retiming step during synthesis. Some VLSI design flows may not support the behavioral retiming step. This could be because of license issues (a DC-Ultra license is required for retiming), or because of limitations of formal verification tools in comparing RTL designs to gate level netlists in which the flip-flops have moved as a result of retiming. For such flows, the retiming step can be disabled by modifying the synthesis scripts as described in the *Xtensa Hardware User's Guide*. Before disabling retiming, ensure that the instruction operands and wires are appropriately scheduled to meet the timing requirements of the design.

Note that it is not possible to assign use stages to the wires used in a computation. The TIE compiler automatically derives the use stages for wires used in any computation. The earliest use stage of a wire that has a def stage assigned to it is the def stage plus one. This is because assigning a def stage to a wire implies that the wire is flopped (registered) before being used in any subsequent computation.

The earliest pipeline stage in which state and regfile operands are available inside the instruction semantic is the `Estage`. However, immediate operands are available in the R stage of the pipeline. It is thus possible to assign a `def Rstage` (or def 0) schedule to wires that are formed from immediate operands only. This forces the computation in the R stage, and the registered result of the computation becomes available in the E stage.

8.7 Implementation Restrictions

Following are the implementation restrictions for the `schedule` sections:

- Operands of register files and states can have a use and def stage ranging from 1 to a maximum of 31.

- The earliest use stage of a TIE state must be less than or equal to its latest def stage. In other words, it is illegal to have the first use stage of a TIE state be greater than its last def stage. Similarly, the earliest use stage on every read port of a TIE register file must be less than or equal to the latest def stage of all register file write ports.
- It is not possible to assign the use stage for intermediate wires declared in a computation.
- It is not possible to assign the def stage for intermediate wires declared inside a TIE function.

Table 8–4 summarizes the use and def stages for various TIE constructs. Note that this table is implementation specific, and may change in future releases of the Xtensa processor.

Table 8–4Summary of Use/Def Stages for Various TIE Constructs

Name	Use/Def Stage
`immediates`	Do not require a schedule. They are available in the R stage of the pipeline.
`state`	You can define use and def stages in the range 1 to 31. If not explicitly specified, default use/def stage is the E stage of the pipeline. If the state is the target of a load, the default def stage is the M stage of the pipeline. Exported states are limited to a maximum def stage of 30.
`regfile`	You can define use and def stages in the range 1 to 31. If not explicitly specified, default use/def stage is the E stage of the pipeline. If the register is the target of a load, then the default def stage is the M stage of the pipeline.
`queue`	Input and Output queues. use/def stage is implicit. use stage for input queues is the M stage of the pipeline def stage for output queues is the M stage of the pipeline use stage for the _NOTRDY signal, and def stage for the _KILL signal for both input and output queues is the M stage.[1]
`lookup`	def stage for lookup address, or output is restricted to: 5-stage pipeline: E, M or W stage. E stage only if optional rdy interface is used 7-stage pipeline: L, M or W stage. L stage only if optional rdy interface is used. use stage for lookup input is restricted to the range (def stage + 1) - (def stage + 10)
`import_wire`	use stage for import wire is implicit and is the E stage of the pipeline
`VAddr`	Addresses for load/store instructions. def stage is implicit, and is the E stage of the pipeline
`MemDataIn*`	Load data for 8/16/32/64/128/256/512 bit loads. use stage is implicit, and is the M stage of the pipeline

Table 8–4Summary of Use/Def Stages for Various TIE Constructs (continued)

Name	Use/Def Stage
`MemDataOut*`	Store data for 8/16/32/64/128/256/512 bit stores def stage is implicit, and is the M stage of the pipeline
`LoadByteDisable`	Byte disable signals to be used for conditional or partial loads. def stage is implicit, and is the E stage of the pipeline
`StoreByteDisable`	Byte disable signals to be used for conditional or partial stores. def stage is implicit, and is the M stage of the pipeline

1. In prior releases of the TIE compiler, the use stage of the NOTRDY interface for output queues was the E stage.

9. Instruction Semantics (semantic) Sections

The behavior of designer-defined TIE instructions is defined with either semantic, reference, or operation description sections. An operation or reference section describes the behavior of a single designer-defined instruction, with emphasis on readability and simplicity. A semantic section can describe one or more instructions, and is written with an emphasis on efficient hardware implementation. For a description of the operation construct, refer to Chapter 7, “Instruction Operation (operation) Sections” on page 37. For a description of the reference construct, refer to Chapter 40, “Instruction Reference (reference) Sections” on page 281.

A semantic section is typically specified in addition to the reference or operation description of an instruction. An instruction with both an operation and semantic description can be thought of as having two views; the operation view is the “easy to read” description and the semantic view is the one optimized for hardware implementation. Cadence recommends that new instructions be first defined using operation (or reference) descriptions. Consider using semantic descriptions only when the hardware cost and performance need improvement.

A semantic description also allows multiple instructions to share the same piece of hardware, resulting in a smaller area as compared to replicating the hardware for every instruction. Instructions defined with individual reference or operation descriptions (that do not also have a semantic description) never share any hardware[1], no matter how similar the instructions. A semantic description is not necessary for instructions whose reference or operation description is already optimal for hardware implementation. There is no opportunity for sharing hardware with other instructions.

A TIE instruction can either be defined using an operation description section, or a combination of opcode, iclass, and reference description sections. These two methods of describing an instruction use a different syntax for specifying the operands of the instruction. As a result, the instructions grouped into a single semantic must either all be defined using operation sections, or must all be defined using opcode/iclass/reference sections. Instructions from the two groups cannot be mixed into a semantic description.

The TIE compiler automatically generates formal verification scripts to verify the operation or reference description of an instruction with its semantic description. These scripts are for the Incisive Conformal Equivalence checking tool that Cadence Design Systems provides, although they can be easily modified to work with other simi-

1. With the exception of using shared functions

lar tools. Cadence recommends running a formal equivalence check between the semantic and operation/reference descriptions to ensure that the two implementations of the same instruction are equivalent.

If there is a semantic description, it is always the description that is used for the generation of hardware. By default, the Instruction Set Simulator also uses the semantic description for simulation, although it can simulate reference descriptions based on a command line switch. The C/C++ compiler may use the operation or reference description to automatically infer the use of TIE instructions from standard C/C++ programs. It is thus important that the operation/reference description (if provided) be functionally equivalent to the semantic description. If an instruction is not included in any semantic description section, then its operation or reference description is used for the generation of hardware and all the software tools. Instructions defined with opcode and iclass description sections that also have a semantic description need not have a reference description.

9.1 Semantic Syntax

```
semantic-section ::= semantic name opcode-oper-list {computation}
name ::= a unique identifier
opcode-oper-list ::= { opcode-oper-name [, opcode-oper-name]* }
computation ::= see Chapter 28
opcode-oper-name ::= (opcode-name | operation-name)
opcode-name ::= a previously defined opcode name
operation-name ::= a previously defined operation name
```

name is a unique identifier for this semantic section. *opcode-oper-list* is a list of one or more *opcode* or *operation* names. The computation section describes the behavior of the instruction, and is described in Chapter 28, “Computation Sections” on page 215. Valid variables for a computation in a semantic section are listed below.

For instructions defined with an operation description section:

- operation name—Chapter 7. Use any operation-name itemized in the opcode-oper-list in the computation section. An operation-name is a single-bit variable. This single-bit variable inherits the name of the operation and evaluates to 1 when that instruction representing the operation is executed. In the example in Section 9.2.1, ADD32 is such a variable.
- *table* name—Chapter 3. Use any table name in the *computation* section.
- *state* name—Chapter 4. Any state declared in the *state-interface-list* of the operation declaration.
- <*state*>_kill where <*state*> is a legal state name as described above. The state must be either an out or inout operand of the instruction.

- *argument* name—Chapter 7. Use any name of all the arguments in the *argument-list* of the operation description of all the instructions in opcode-oper-list in the *computation* section.
- <*argument*>_kill where <*argument*> is a legal argument name as described above. The argument should be of type out or inout.
- <*argument*>_bitkill where <*argument*> is a legal argument name as described above. The argument should be of type out or inout.
- *interface* name—Chapter 6. Use any interface signal in the computation section if the operation section does not include a *state-interface-list*. If such a list is defined, then only use the interfaces declared in the list.
- *wire* name— Wires are temporary or intermediate signals created inside a semantic description as described in Section 28.2.1 on page 215.

For instructions defined with opcode/iclass description sections:

- opcode name—Chapter 33. Only use an opcode-name itemized in the opcode-oper-list in the computation section. An opcode-name is a single-bit variable. A single-bit variable inherits the name of the opcode and evaluates to 1 when the opcode is detected. This variable indicates the presence of the corresponding instruction in the computation section. In the example in Section 9.2.2, ADSEL is such a variable.
- *table* name—Chapter 3. Use any table name in the *computation* section.
- *state* name—Chapter 4. Only use the states in the *state-list* of the iclasses for the opcodes in the *opcode-oper-list* in the *computation* section.
- <*state*>_kill where <*state*> is declared as either out or inout in the *state-list* of the iclasses for the opcodes in the *opcode-oper-list*.
- *operand* name—Chapter 34. Only use the operands in the *operand-list* of the iclasses for the opcodes in the *opcode-oper-list* in the *computation* section.
- <*operand*>_kill where <*operand*> is declared as either out or inout in the *operand-list* of the iclasses for the opcodes in the *opcode-oper-list*.
- <*operand*>_bitkill where <*operand*> is declared as either out or inout in the *operand-list* of the iclasses for the opcodes in the *opcode-oper-list*.
- *interface* name—Chapter 6. Only use the interface signals in the *interface-list* of the iclasses for the opcodes in *opcode-oper-list* in the *computation* section.
- *wire* name— Wires are temporary or intermediate signals created inside a semantic description as described in Section 28.2.1 “Wires Statement” on page 215.

9.2 Examples

Following are two semantic examples, one for instructions with operation descriptions and the other with reference descriptions.

9.2.1 Semantics for Instructions with operation Descriptions

Consider a 64-bit register file VR as defined in the following example. Assume that this register file holds vector operands, where each vector consists of two 32-bit wide scalars. We define a vector add and a vector subtract instruction that operates on this register file.

```
regfile VR 64 16 v

operation ADD32 {out VR sum32, in VR input0, in VR input1} {} {
     wire [31:0] sum0 = input0[31: 0] + input1[31: 0];
     wire [31:0] sum1 = input0[63:32] + input1[63:32];
     assign sum32 = {sum1, sum0};
}

operation SUB32 {out VR sum32, in VR input0, in VR input1} {} {
     wire [31:0] sum0 = input0[31: 0] - input1[31: 0];
     wire [31:0] sum1 = input0[63:32] - input1[63:32];
     assign sum32 = {sum1, sum0};
}
```

The implementation of this TIE description results in two 32-bit adders for the ADD32 instruction and two more 32-bit adders for the SUB32 instruction. By using the following semantic section, the same two adders can be shared between the two instructions, resulting in a better hardware implementation.

```
semantic add_sub {ADD32, SUB32} {
     wire [63:0] oprnd = ADD32 ? input1 : ~input1;
     wire carry_in = SUB32 ? 1 : 0;
     wire [31:0] sum0 = TIEadd(input0[31: 0], oprnd[31: 0], carry_in);
     wire [31:0] sum1 = TIEadd(input0[63:32], oprnd[63:32], carry_in);
     assign sum32 = {sum1, sum0};
}
```

The semantic description, add_sub, uses the TIE built-in module TIEadd, which implements a two input adder with a carry-in. TIE built-in modules are described in Chapter 29, “TIE Built-in Modules” on page 233. The description exploits the property of two’s complement binary numbers, so that subtracting one number from another is the same as adding the inverted value with a carry in.

The semantic description makes use of wire declarations to create temporary variables such as sum0 and sum1 to hold intermediate results. It also uses the automatically generated, single bit, one-hot encoded signals to make selections based on the instruction being executed. For example, wire carry_in is set to 1 if the SUB32 instruction is being executed, otherwise it is set to 0.

It is important for the operation description of both the ADD32 and SUB32 instructions to use the same names for their arguments (sum32, input0, input1), so that the semantic description can be written as shown in the preceding example. This naming indicates to the TIE compiler that the same register file operand should be used for the input arguments of these two instructions. If different names are used, the TIE compiler may assign different operands to these arguments, which would result in extra hardware that could have been avoided.

Consider the problem of creating another instruction that adds an immediate value in the 0 to 255 range to both of the scalars in the vector register. It also writes the result of the lower portion of the register operand to a state. The operation description of this instruction follows:

```
state st 32 add_read_write
immediate_range IMRNG 0 255 1
operation ADDIMM {out VR sum32, in VR input0, in IMRNG immd} {out st} {
     wire [31:0] sum0 = input0[31: 0] + immd;
     wire [31:0] sum1 = input0[63:32] + immd;
     assign sum32 = {sum1, sum0};
     assign st = sum0;
}
```

Because this instruction also performs an addition, it can be implemented in the semantic add_sub as follows:

```
semantic add_sub {ADD32, SUB32, ADDIMM} {
     wire [63:0] oprnd = ADD32 ? input1 : ~input1;
     wire carry_in = SUB32 ? 1 : 0;
     wire [63:0] addin = ADDIMM ? {immd, immd} : oprnd;
     wire [31:0] sum0 = TIEadd(input0[31: 0], addin[31: 0], carry_in);
     wire [31:0] sum1 = TIEadd(input0[63:32], addin[63:32], carry_in);
     assign sum32 = {sum1, sum0};
     assign st = sum0;
}
```

The first aspect to note about the preceding example is that while the vector input argument of the ADDIMM instruction is named input0, the immediate argument is not named input1. The *type* of the first input argument of all three instructions is the same (all are registers from the register file VR), so their names are the same. The type of the second input argument of ADD32 and SUB32 is also the same, so their names are the same. However, the type of the second input argument of the instruction ADDIMM is dif-

ferent (it represents the immediate_range IMRNG), so it has a different name. It would be illegal to name this argument input1 because inside the semantic block there is not a way to distinguish between the ADD32 argument, which is of type VR, and the ADDIMM argument, which is of type IMRNG. Thus, when two instructions sharing a semantic declare arguments of different types, they are required to have different names. The appropriate argument is selected within the semantic description based on the instruction being executed.

Another aspect the above semantic illustrates is that immediate arguments are always 32 bits wide. Note that immediate_range IMRNG can be represented by an 8-bit field. However, the argument immd of type IMRNG is a zero-extended 32-bit value, so that the expression {immd, immd} represents a 64-bit number.

The restrictions on the naming of operation arguments relaxes if the corresponding instructions do not share a semantic. In the above example, the immediate argument of the ADDIMM instruction could have been called input1 if it was not part of the add_sub semantic. Also, there is no advantage to having the same names for the arguments of the ADD32 and SUB32 operations without an add_sub semantic.

In addition to writing to operation argument sum32, operation ADDIMM also writes to a state st. In the semantic, state st is directly assigned to the value. It is not qualified by the opcode ADDIMM. It is because all other operations ADD32, SUB32 do not write to the state. TIE compiler ensures that the value of state st is only updated when operation ADDIMM is exercised. When operations ADD32 and SUB32 are exercised, the value of state st is unchanged.

9.2.2 Semantics for Instructions with reference Descriptions

The following TIE code shows the iclass and reference description of two instructions, ADSEL and ACS.

```
opcode ADSEL op2=0 CUST0
iclass adsel {ADSEL} {out arr, in ars, in art}
reference ADSEL {
     wire [15:0] add0, add1, sub0, sub1;
     wire [15:0] hi_res, lo_res;

     assign add0 = ars[15: 0] + art[4:0];
     assign add1 = ars[31:16] + art[4:0];
     assign sub0 = ars[15: 0] - art[4:0];
     assign sub1 = ars[31:16] - art[4:0];

     assign hi_res = art[17] ? add0 : sub1;
     assign lo_res = art[16] ? add1 : sub0;
     assign arr = {hi_res, lo_res};
}
```

```
opcode ACS op2=1 CUST0
iclass acs{ACS} {out arr, in ars, in art}
reference ACS {
     wire [15:0] add0, add1, sub0, sub1;
     wire d0, d1;

     assign add0 = ars[15: 0] + art[4:0];
     assign add1 = ars[31:16] + art[4:0];
     assign sub0 = ars[15: 0] - art[4:0];
     assign sub1 = ars[31:16] - art[4:0];

     assign d0 = (add0 > sub1);
     assign d1 = (add1 > sub0);

     assign arr = {art[29:16], d0, d1, art[15:0]};
}
```

In the preceding `reference` description, the computation for the signals add0, add1, sub0, and sub1 is exactly the same for the instructions ADSEL and ACS. Hence, the following semantic description shares this logic.

```
semantic viterbi {ADSEL, ACS} {
     wire [15:0] add0, add1, sub0, sub1;
     wire [15:0] hi_res, lo_res;
     wire [31:0] acsres, adselres;
     wire d0, d1;

     assign add0 = ars[15: 0] + art[4:0];
     assign add1 = ars[31:16] + art[4:0];
     assign sub0 = ars[15: 0] - art[4:0];
     assign sub1 = ars[31:16] - art[4:0];

     assign d0 = (add0 > sub1);
     assign d1 = (add1 > sub0);

     assign hi_res = art[17] ? add0 : sub1;
     assign lo_res = art[16] ? add1 : sub0;
     assign adselres = {hi_res, lo_res};
     assign acsres = {art[29:16], d0, d1, art[15:0]};

     assign arr = ADSEL ? adselres : acsres;
}
```

10. Instruction Format (format) Sections

Instruction formats can be viewed as templates for specifying which operations can be grouped (packed) into a single FLIX instruction. Under each instruction length, there exists one or more instruction formats. An instruction `format` contains one or more instruction slots. Each instruction `slot` contains a collection of operations that can be issued from that slot. Instruction formats are relevant only for FLIX TIE files. Think of non-FLIX TIE files as having a single format with one slot.

10.1 Instruction Format Syntax

```
format-def ::= format name (syntax1 | syntax2)
name ::= a unique identifier
syntax1 ::= length {slot-list}
syntax2 ::= length-name {slot-list} [decoding-expr]
length ::= an integer specifying the length of the format
slot-list ::= slot [, slot]
slot ::= a slot in the format
length-name ::= name of a previously defined length
decoding-expr ::= {constraint-expr [ && constraint-expr] }
constraint-expr ::= InstBuf[slice] == sized-constant
slice ::= n:m | n
```

name is a unique identifier for the instruction format. A format declaration follows one of two available syntax referred to as *syntax1* and *syntax2*. For *syntax1*, *length* is an integer specifying the length of an instruction belonging to this format. Its value can be from 32 to 128 with an increment of 8, and this value must be no greater than the maximum instruction width parameter of the Xtensa configuration with which you intend to compile this TIE description.

For format declarations that use the alternative syntax (*syntax2*), *length-name* is the name of a previously defined `length` as described in Chapter 37, “Length (`length`) Sections” on page 273. *slot-list* is a list of slots that belong to this format. *decoding-expr* specifies the decoding logic for the instruction format. If there is only one format corresponding to the instruction length, *decoding-expr* is not required. However, when there are multiple instruction formats within a length, *decoding-expr* distinguishes between them. The only valid decoding expression is one that compares bits of `InstBuf` to a constant. `InstBuf` is a special predefined variable that refers to the instruction buffer, which is assumed to be as wide as the widest instruction supported by your Xtensa processor configuration. The decoding logic of a `format` must be logically disjointed from the decoding logic of its *length* and that of all other formats in the same length.

When a format description is specified using *syntax1*, the TIE compiler internally generates a *length* description corresponding to the format. If more than one format is assigned to a length, the TIE compiler automatically generates the decoding expression for those formats. Note that it is illegal to specify a decoding expression in a format description that uses *syntax1*.

10.2 Example

The following example defines a single format for a 32-bit wide Xtensa processor configuration. The format consists of two slots, `slot_a` and `slot_b`.

```
format f32 32 {slot_a, slot_b}
```

Following is an alternative syntax for the above format declaration. The *length* declaration associated with the format is shown for completeness.

```
// For a little endian configuration
length l32 32 {InstBuf[3:0] == 15}
format f32 l32 {slot_a, slot_b}

// For big endian configuration
length l32 32 {InstBuf[31:28] == 15}
format f32 l32 {slot_a, slot_b}
```

The following example defines two different formats for a 64-bit wide, little-endian configuration. Format `f64a` and format `f64b` both belong to length `l64`, requiring a decoding expression in the format declaration. Bit [63] of `InstBuf` distinguishes between these two formats. Format `f64a` consists of two slots while format `f64b` consists of three slots.

```
// This definition is for little endian processor configurations
length l64 64 {InstBuf[3:0] == 15}

format f64a l64 {slotx, sloty} {InstBuf[63] == 0}
format f64b l64 {slota, slotx, slotc} {InstBuf[63] == 1}
```

An alternative, and simpler syntax for the above declaration follows:

```
format f64a 64 {slotx, sloty}
format f64b 64 {slota, slotx, slotc}
```

In this example, the TIE compiler internally generates a length description for formats `f64a` and `f64b`. It also automatically assigns the decoding expression to distinguish between these two formats. This format description is endian-independent because the TIE compiler will use the appropriate `InstBuf` bits (for the decoding expression) based on the endianness of your Xtensa processor.

If a format is specified using *syntax1* in a TIE file, it can be re-defined using *syntax2* to specify its encoding, as shown in the following example[1]:

```
format fmt 64 {slot0, slot1}

length l64 64 {InstBuf[3:0] == 15}
format fmt l64 {slot0, slot1} {InstBuf[63] == 1'b1}
```

10.3 Slot Indices of a Format

The order of the slots listed in the *slot-list* of a `format` declaration is very important. It determines the assembly language syntax of instructions belonging to that format, and the amount of hardware that is shared by instructions belonging to different formats.

Each slot in the *slot-list* is assigned a slot index, starting from 0 and increasing by 1 as you go from left to right. This definition is independent of the endianness of the configuration and is illustrated in the example below. All the 24-bit, base Xtensa processor instructions belong to the predefined slot named `Inst`, which has a slot index of 0. Table 10–5 lists the slot indices for the formats.

```
format f64a 64 {slot_a, slot_b}
format f64b 64 {slot_x, slot_y, slot_z}
format f64c 64 {sl_m, sl_n, sl_o, sl_p}
```

Table 10–5. Slot Indices of a Format

	Slot Index			
Format	**0**	**1**	**2**	**3**
Xtensa	`Inst`			
f64a	`slot_a`	`slot_b`		
f64b	`slot_x`	`slot_y`	`slot_z`	
f64c	`sl_m`	`sl_n`	`sl_o`	`sl_p`

10.3.1 Assembly Language Syntax

The assembly language syntax of instructions always follows the slot-list order. Thus, operations from slot index 0 are listed first, followed by operations from slot index 1 and so on. Since the slot index assignment is independent of endianness, the assembly language syntax is also independent of endianness.

1. Before release RE-2013.0 of the TIE compiler, format cannot be redefined.

Consider the following piece of TIE code that defines a format with two slots. The instructions available in each slot are specified using the *slot_opcodes* statement, which is described in Chapter 11, “Slot Opcode (slot_opcodes) Sections” on page 77.

```
format flen 32 {slot_a, slot_b}

// INST_A, INST_AA, INST_B, INST_BB are user defined TIE instructions
slot_opcodes slot_a {INST_A, INST_AA}
slot_opcodes slot_b {INST_B, INST_BB}
```

Following are some examples of legal and illegal assembly language syntax for the above TIE description. Note that the assembler for the Xtensa processor is capable of automatically generating legal FLIX instructions from “scalar” or straight line assembly code. Automatic bundling is enabled with the --schedule flag of the assembler.

```
{INST_A;  INST_B}        # Legal
{INST_A;  INST_BB}       # Legal
{INST_AA; INST_B}        # Legal
{INST_AA; INST_BB}       # Legal

{INST_A;  INST_AA}       # Illegal: INST_AA is not allowed in slot_b
{INST_BB; INST_A}        # Illegal: INST_BB is not allowed in slot_a
                         # and INST_A is not allowed in slot_b
```

10.3.2 Hardware Sharing Between Slots

The hardware implementation of a FLIX design has one copy of an instruction semantic for every slot index in which the instruction appears. Thus, when a particular instruction is available in more than one slot, the resulting hardware may have one or more instances of the instruction semantic. If an instruction is available in two slots that belong to the same slot index (across different formats), there is only one copy of the instruction semantic in the hardware. If the instruction is available in two slots that belong to different slot indices, there will be two instances of the instruction semantic in the hardware implementation.

Consider the following example, which has two formats. Format f32a has two slots and format f32b has three slots as shown in Table 10–6. The slot_opcodes statement lists the operations available in each slot.

```
format f32a 64 {slot_x, slot_y}
format f32b 64 {slot_p, slot_q, slot_r}

// FOO, BAR, and TEE are designer defined TIE instructions
// The remaining are Xtensa ISA instructions available in
// the respective slots
slot_opcodes slot_x {FOO, ADD}
```

```
slot_opcodes slot_y {BAR, TEE}
slot_opcodes slot_p {SUB, FOO, L32I}
slot_opcodes slot_q {BEQ, BAR}
slot_opcodes slot_r {FOO, BAR, TEE}
```

Table 10–6. Example: Slot Indices of a Format

Format	Slot Index 0	1	2
f32a	slot_x	slot_y	
f32b	slot_p	slot_q	slot_r

In this example, slot_x and slot_p both have the designer-defined instruction FOO. Because they have the same slot index (slot index 0) they can share the hardware that implements the instruction FOO. Note that slot_r, whose slot index is 2, also supports the instruction FOO. However, slot_r will need its own copy of the hardware because it has a different slot index. The reason a separate copy is needed is that operation FOO in slot_p and slot_r need to be executed in parallel. Thus, in the above configuration, there are three slots that support the instruction FOO, but only two copies of the semantic hardware. One copy is shared by slot_x and slot_p, and the other copy is for the exclusive use of slot_r. Similarly, instruction BAR in slot_y and slot_q can share one semantic, but BAR in slot_r requires a separate semantic.

All the 24-bit, predefined Xtensa ISA instructions belong to a slot named Inst, whose slot index is 0. Thus the ADD instruction in slot_x shares hardware with the ADD instruction in slot Inst. The same is true for instruction SUB and L32I in slot_p.

Instruction TEE are defined in slot_y and slot_r, whose slot indices are 1 and 2, respectively. Because they are in different slot indices, their hardware is not shared by default. However, slot_y and slot_r belong to different instruction formats, and thus it is not possible to create a FLIX bundle in which two instances of operation TEE are issued. This means that only one instance of the hardware for the operation TEE is sufficient. Starting with version RD2010.0, TIE compiler has the capability to share semantic hardware in different slot indices. This is achieved by using shared_semantic property described in Section 21.6 “Semantic Sharing Property” on page 187. In this example, by adding the following property:

```
property shared_semantic TEE {1, 2}
```

TIE compiler only generates one copy of semantic TEE. If the property were not specified, two instances of the semantic would be generated in hardware.

Table 10–7 summarizes the hardware that is shared between various slots in this example.

Table 10–7. Example: Hardware Sharing Between Slots

Slot Index	Slot Name	Comment
0	slot_x, slot_p, Inst	▪ Hardware for FOO shared between slot_x and slot_p ▪ Hardware for ADD shared between slot_x and Inst ▪ Hardware for L32I shared between slot_p and Inst ▪ Hardware for SUB shared between slot_p and Inst ▪ Hardware for BEQ used only by Inst
1	slot_y, slot_q	▪ Hardware for BAR shared between slot_y and slot_q ▪ Second copy of hardware for BEQ, used only by slot_q ▪ Hardware for TEE is used only by slot_y if shared_semantic property is not specified. It is shared with the hardware for TEE in slot_r if shared_semantic property is specified.
2	slot_r	▪ Second copy of hardware for FOO, not shared ▪ Second copy of hardware for BAR, not shared ▪ Second copy of hardware for TEE is generated if shared_semantic property is not specified. It is shared with the hardware for TEE in slot_r if shared_semantic property is specified.

10.3.3 Load/Store Instructions in FLIX Formats

Load and store instructions need to be associated with a load/store unit in the hardware. Load/store units can only be shared by slots that have the same slot index. All pre-defined Xtensa ISA instructions are associated with load/store unit 0, which is always present in all Xtensa configurations. In the example of Section 10.3.2, `slot_p` supports the predefined Xtensa load instruction `L32I`. Because `slot_p` has a slot index of 0, it can share load/store unit 0 with the Xtensa ISA instructions belonging to the `Inst` slot. A TIE description in which a load/store instruction is implemented in a slot whose slot index is non-zero requires an Xtensa configuration with two load/store units. Furthermore, all such load/store instructions (that do not belong to slots with slot index zero) must be grouped in slots that have the same slot index.

The following TIE code is legal for configurations with two load/store units, but illegal for configurations with only one load/store unit. It has load/store instructions in `slot_z`, whose slot index is non-zero. The Xtensa ISA instructions in the `Inst` slot of this configuration use the load/store unit associated with slot index 0, and the instructions of `slot_z` require the second load/store unit.

```
format f64 64 {slot_y, slot_z}
slot_opcodes slot_y {BLT, BNE, AND}
slot_opcodes slot_z {L16UI, S16I, NEG, ABS}
```

If you swap the order of `slot_y` and `slot_z` in the format declaration, then the instructions of `slot_z` can share a load/store unit with the `Inst` slot and the TIE would work with a single load/store unit configuration.

Load or store operations to Instruction RAM (IRAM) can only be issued from slots with slot index 0. If a load/store to IRAM is issued from a slot with non-zero index, the Xtensa processor will take an exception due to a load/store error.

10.4 Implementation Restrictions

An Xtensa configuration can have a maximum of 24 formats. This number includes the `Inst` format, two formats for density option if selected, and formats used by Xtensa DSP ISA options such as HiFi2, ConnX BBE16, *etc.*

An Xtensa configuration can have a maximum of 64 slots across all FLIX formats, including formats defined as part of core configuration options such as FLIX3 or HiFi3.

Formats with width 64-bits or less can have a maximum of 15 slots each. Formats with width more than 64-bits can have a maximum of 30 slots each.

An Xtensa configuration that does not have any wide instructions (32-bit, 64-bit, or 128-bit) nor the density option is assumed to have all instructions of length 24-bits. It is further assumed that all 24-bits belong to the predefined `Inst` slot. It is thus not possible to create additional formats with multiple slots in such a configuration.

11. Slot Opcode (slot_opcodes) Sections

The slot_opcodes construct provides a way to define the set of operations or instructions that belong to a particular slot. slot_opcodes sections are typically used to specify the instructions available in a designer-defined slot of a FLIX instruction. They are also used to specify designer-defined instructions that need to be encoded as part of the predefined Xtensa processor slot Inst.

Every slot listed in the slot-list of a format declaration must have a matching slot_opcodes section. A slot can have multiple slot_opcodes sections; this is equivalent to having one slot_opcodes section that is a concatenation of all the instructions listed in the individual sections.

11.1 Slot Opcode Syntax

```
slot_opcodes-def ::= slot_opcodes slot-name {inst-list}
slot-name ::= name of a previously defined slot
inst-list ::= inst-name [, inst-name]*
inst-name ::= name of an instruction or group of instructions
```

slot-name is the name of a previously defined slot. The slot should have been previously defined as part of the slot-list of a format declaration. *inst-list* is a list of one or more instructions that belong to this slot. This list may include designer-defined TIE instructions, and a subset of the predefined Xtensa ISA instructions.

inst-name must be either a user TIE instruction defined elsewhere in the TIE description, or a predefined Xtensa ISA instruction. Designer-defined TIE instructions are typically defined using the operation construct which is described in Chapter 7, “Instruction Operation (operation) Sections” on page 37. This definition may occur before or after the slot_opcodes statement; the order is not important. Xtensa ISA instructions that are assigned to a slot using the slot_opcodes statement should not be defined in the TIE description. These instructions are predefined, and a redefinition of these instructions is illegal. Note that all instructions in *inst-list* must be listed using the same case (upper or lower case alphabet) in which they are defined. Xtensa ISA instructions must be listed in all upper case alphabet. *inst-name* can also be the name of a group of instructions as explained in Section 11.3.1.

To use a designer-defined TIE instruction in the 24-bit instruction format (used by most Xtensa ISA instructions), assign it to the predefined slot Inst. Alternately, an instruction that is not assigned to any slot is assumed to belong to the slot Inst. A TIE instruction may be defined in a FLIX slot as well as in the Inst slot. When such an instruction can be paired with another operation, it can be issued as a FLIX instruction to achieve higher

performance. When the instruction cannot be paired with any other operation, the 24-bit version can be issued to achieve better code density. Designer-defined TIE instructions cannot be assigned to the 16-bit instruction format reserved for specific predefined Xtensa processor instructions.

11.2 Encoding of Instructions in a Slot

All the instructions listed in the `slot_opcodes` list of a slot have to be encoded using the bits available in that slot. The designer explicitly defines the number of bits available in a slot (as illustrated in Chapter 38, “Instruction Slot (`slot`) Sections” on page 275) or it is automatically determined by the TIE compiler. When the TIE compiler decides the width of a slot, it allocates enough bits to ensure that all the instructions of that slot can be encoded. However, this may not be possible if the sum total of the bits required for all the slots in a TIE description exceeds the instruction length. It is the designer’s responsibility to ensure that the slot is wide enough to accommodate all its instructions if the designer specifies the slot width. In either case, the TIE compiler will generate an error if a slot is not wide enough to encode all the instructions required in it.

If there is a problem fitting your instructions into the instruction length, there are several ways to solve it, depending upon the requirements of the design. Following are some suggestions:

- Reduce the number of instructions in that slot
- Modify one or more of the instructions so that they take fewer bits to encode. For example, an instruction with an 8-bit immediate operand can be changed to use a 6-bit immediate.
- Modify the instructions of another slot (as above), thereby making more bits available for this slot.

11.3 Xtensa ISA Instructions in FLIX Slots

All Xtensa ISA instructions included in your configuration are available in their native 24-bit or 16-bit formats. A subset of these instructions can be included in designer-defined FLIX slots. To include a particular Xtensa ISA instruction in a FLIX slot, simply include the instruction name in the `slot_opcodes` declaration of that slot. Instructions can be included in multiple FLIX slots, by including them in the `slot_opcodes` declaration of each of the desired slots. Note that Xtensa ISA instructions must be listed in a `slot_opcodes` declaration using all upper case text. Using lower case text will result in an error message from the TIE compiler.

Only a subset of the Xtensa ISA instructions can be assigned to FLIX slots. Table 11–8 lists the instructions that can be assigned to FLIX slots. In addition, the wide branch instructions listed in Table 11–10 are assigned to FLIX slots.

Note: It is illegal to assign any Xtensa ISA instructions that are not in either Table 11–8 or Table 11–10 to a FLIX slot.

Table 11–8. Xtensa ISA Instructions Available in FLIX Slots

Instruction Class	Available Instructions
Load/Store	L16SI, L16UI, L32I, L32I.N, L32R, L8UI, S16I, S32I, S32I.N, S8I
ALU	ABS, ADD, ADD.N, ADDI, ADDI.N, ADDMI, ADDX2, ADDX4, ADDX8, ALL4, ALL8, AND, ANDB, ANDBC, ANY4, ANY8, MAX, MAXU, MIN, MINU, NEG, OR, ORB, ORBC, SUB, SUBX2, SUBX4, SUBX8, XOR, XORB
Shift	SLL, SLLI, SRA, SRAI, SRC, SRL, SRLI, SSA8B, SSA8L, SSAI, SSL, SSR
Branch	BALL, BANY, BBC, BBCI, BBS, BBSI, BEQ, BEQI, BEQZ, BEQZ.N, BF, BGE, BGEI, BGEU, BGEUI, BGEZ, BLT, BLTI, BLTU, BLTUI, BLTZ, BNALL, BNE, BNEI, BNEZ, BNEZ.N, BNONE, BT, CALL0, CALLX0, CALL8, CALLX8, LOOP, LOOPGTZ, LOOPNEZ, J, JX, RET, RETW
Move	MOV.N, MOVEQZ, MOVF, MOVGEZ, MOVI, MOVI.N, MOVLTZ, MOVNEZ, MOVT
Multiply	MUL16S, MUL16U, MULL, MULSH, MULUH
MAC16	LDDEC, LDINC, MUL.AA.*, MUL.AD.*, MUL.DA.*, MUL.DD.*, MULA.AA.*, MULA.AD.*, MULA.DA.*, MULA.DD.*, MULS.AA.*, MULS.AD.*, MULS.DA.*, MULS.DD.*, UMUL.AA*
Prefetch[1]	DPFR, DPFR0, DPFW, DPFW0
Block Prefetch[1]	DPFR.B,DPFW.B,DPFM.B, DPFR.BF, DPFW.BF, DPFM.BF, DHI.B, DHWB.B, DHWBI.B, PFNXT.F, PFWAIT.R, PFWAIT.A, PFEND.O, PFEND.A
Miscellaneous instructions	CLAMPS, EXTUI, NOP, NSA, NSAU, SEXT, DEPBITS
Coprocessor	All coprocessor instructions such as instructions in Floating Point, Vectra LX, HiFi2 Audio Engine, and ConnX D2 etc. coprocessors
Designer-defined instructions	RUR, WUR

1. Instructions can only be put to slots with index zero. The instructions cannot be bundled with a load or store instruction.

While the Xtensa ISA instructions can be assigned to any FLIX slot, it is advantageous to assign them to slots whose slot index is 0. This allows them to share the semantic hardware with the base Xtensa processor. If the instructions cannot be assigned to slots whose slot index is 0. Use `shared_semantic` property described in Section 21.6 to share semantic hardware if possible. In a single load store unit configuration, the pre-defined Xtensa load/store instructions can only be assigned to slots with slot index 0. This is explained in detail in Section 10.3 “Slot Indices of a Format” on page 71.

When an Xtensa ISA instruction is included in a FLIX slot, you may supply your own encoding or let the TIE compiler automatically do the opcode assignment. In either case, it is not necessary for the opcode encoding in the FLIX slot to be the same as the opcode encoding in the `Inst` slot.

When an Xtensa ISA instruction is included in a slot with slot index 0, using the same encoding as the `Inst` slot is likely to result in less decoding logic. Furthermore, if all the fields are assigned to match with the `Inst` slot, the amount of multiplexing necessary to generate the operands for the various instruction semantics is reduced.

In many cases, it may be possible to encode an Xtensa ISA instruction with fewer than 24 bits in a FLIX slot. This is particularly true when only a few of these instructions are included in a particular slot. If the slot in question has a non-zero slot index, it does not share hardware with the base Xtensa processor, so there is no specific advantage in assigning it the same encoding as the `Inst` slot. In this situation it may be advantageous to give it a different, narrower encoding. When the TIE compiler selects the encoding for an Xtensa ISA instruction in a slot with non-zero slot index, it tries to take advantage of this to the fullest extent possible.

Take special care with Xtensa branch instructions assigned to more than one slot of a given format. While it is legal to issue the branch instruction in any of the slots, it is illegal to create a FLIX instruction that has a branch instruction in more than one slot. The assembler will generate an error if this is attempted.

11.3.1 Instruction Groups

To facilitate the inclusion of groups of Xtensa ISA instructions in FLIX slots, there are several predefined groups of instructions as listed in Table 11–9. For user defined instruction groups, refer to Chapter 22, “Instruction Group (instruction_group) Sections” on page 193. When you list the group name in a slot_opcodes statement, the slot gets all the instructions belonging to that group.

Table 11–9. Predefined Groups of Xtensa ISA Instructions

Group Name	List of Instructions
xt_loadstore	`L8UI, L16UI, L16SI, L32I, S32I, S16I, S8I`
xt_alu	`ADD, ADDI, ADDMI, ADDX2, ADDX4, ADDX8, SUB, SUBX2, SUBX4, SUBX8, ABS, NEG, AND, OR, XOR`
xt_shift	`SLLI, SLL, SRA, SRAI, SRC, SRL, SRLI, SSA8B, SSA8L, SSAI, SSL, SSR`

Table 11–9. Predefined Groups of Xtensa ISA Instructions (continued)

Group Name	List of Instructions
xt_branch[1]	BALL, BANY, BBC, BBCI, BBS, BBSI, BEQ, BEQI, BEQZ, BGE, BGEI, BGEU, BGEUI, BGEZ, BLT, BLTI, BLTU, BLTUI, BLTZ, BNALL, BNE, BNEI, BNEZ, BNONE, J, JX
xt_move	MOVEQZ, MOVGEZ, MOVI, MOVLTZ, MOVNEZ
xt_boolean	ALL4, ALL8, ANDB, ANDBC, ANY4, ANY8, BF, BT, MOVF, MOVT, ORB, ORBC

1. Note that the BF and BT branch instructions are included as part of the boolean group and not as part of the branch group.

Thus, as per Table 11–9, the following two statements are equivalent:

```
slot_opcodes slot0 {xt_move}

slot_opcodes slot0 {MOVEQZ, MOVGEZ, MOVI, MOVLTZ, MOVNEZ}
```

11.4 Wide Branch Instructions

The Xtensa ISA supports a rich variety of branch instructions that perform a test and branch operation in one instruction. These instructions specify the branch target address as an 8- or 12-bit immediate operand encoded in the instruction word. The limited branch range limits the ability of the XCC compiler to optimize the layout of code, and in some cases requires the use of a two branch sequence for what would otherwise be a single branch.

To address this problem, a special set of wide branch instructions are available for use in your TIE design. These instructions have the same functionality as the Xtensa ISA branch instructions, except that they have a wider immediate operand to encode the branch target address. Two such sets are available, the first with all branch instructions having a 15-bit immediate offset and another with all branches having an 18-bit immediate offset. The 18-bit immediate version is well suited for inclusion in a two slot format of a 64 bit instruction while the 15-bit version is suited for inclusion in a single slot format of a 32-bit instruction. These branch instructions are summarized in Table 11–10.

Table 11–10. Wide Branch Instructions

Group Name	List of Instructions
xt_widebranch15[1]	BALL.W15, BANY.W15, BBC.W15, BBCI.W15, BBS.W15, BBSI.W15, BEQ.W15, BEQI.W15, BEQZ.W15, BGE.W15, BGEI.W15, BGEU.W15, BGEUI.W15, BGEZ.W15, BLT.W15, BLTI.W15, BLTU.W15, BLTUI.W15, BLTZ.W15, BNALL.W15, BNE.W15, BNEI.W15, BNEZ.W15, BNONE.W15
xt_widebranch18	BALL.W18, BANY.W18, BBC.W18, BBCI.W18, BBS.W18, BBSI.W18, BEQ.W18, BEQI.W18, BEQZ.W18, BGE.W18, BGEI.W18, BGEU.W18, BGEUI.W18, BGEZ.W18, BLT.W18, BLTI.W18, BLTU.W18, BLTUI.W18, BLTZ.W18, BNALL.W18, BNE.W18, BNEI.W18, BNEZ.W18, BNONE.W18

1. Note that the BF and BT branch instructions do not have a wide branch equivalent.

These two groups of wide branch instructions are available to include in your FLIX slots, similar to the Xtensa ISA instructions. You do not have to define the operation of these instructions, just include the group name in the desired FLIX slot(s). Note that the wide branch instructions can only be included as a group and not individually. Thus you either get all the instructions in the group or none of them. In contrast, the Xtensa ISA instructions can be included individually, or as groups defined in Table 11–9. Another difference is that the Xtensa ISA instructions exist both in the FLIX slot(s) and in the 24-bit or 16-bit native form. However, the wide branch instructions only exist in the FLIX slot.

The wide branch instructions take a lot of bits to encode, so you should be careful when including them in FLIX slots. In particular, the `xt_widebranch18` group requires a 64-bit instruction width — they do not fit in a 32-bit instruction word. The wide instruction groups `xt_widebranch15` and `xt_widebranch18` cannot co-exist in one configuration. Only use the group that most suits your application.

11.5 NOP Instructions in FLIX Slots

Every FLIX slot is required to have a NOP instruction. A NOP instruction performs no operation. This is necessary so that the compiler and assembler can bundle operations in situations where not all slots can have a useful operation. The TIE compiler automatically includes a NOP instruction in every slot, even if it is not specified as part of the `slot_opcodes` declaration.

11.6 Examples

The following example defines two slots in a format, and the instructions that belong to each slot:

```
format fmt 64 {slot_x, slot_y}

slot_opcodes slot_x {INST_A, INST_B, ADD}
slot_opcodes slot_y {INST_B, INST_C, BEQ}
```

slot_x includes designer-defined TIE instructions INST_A and INST_B, and the Xtensa ISA instruction ADD. slot_y includes the designer-defined instructions INST_B and INST_C, along with the Xtensa ISA instruction BEQ. The preceding example assumes that the instructions INST_A, INST_B, and INST_C are defined elsewhere in the TIE file. Note that because they are predefined Xtensa ISA instructions, it is illegal to define the ADD and BEQ instructions. The TIE compiler automatically includes a NOP instruction in both slots.

The following example has multiple slot_opcodes statements for the same slot. This is legal and has the same effect as having one slot_opcodes statement that assigns both FOO and BAR to slot myslot.

```
operation FOO { .. } {} {
     ...
}

slot_opcodes myslot {FOO}

operation BAR { .. } {} {
     ...
}

slot_opcodes myslot {BAR}
```

The following example has a format with two slots, both of which have branch instructions in them. Such a definition is legal, but there are some restrictions with its usage.

```
format wfmt 64 {slot_p, slot_q}

slot_opcodes slot_p {L32I, ADD, BGE}
slot_opcodes slot_q {BLTZ, SLL, XOR}
```

For the above TIE description, it is legal to issue a branch operation from either slot of an instruction, provided the other slot of the instruction does not have a branch. This is illustrated in the following assembly code examples:

```
{L32I, BLTZ}   // OK - branch in slot_q

{BGE, XOR}     // OK - branch in slot_p

{BGE, BLTZ}    // Illegal - cannot have branch in both slots
```

The following example has a designer-defined instruction, RADD, that is available as part of a FLIX slot (sl0) as well as a 24-bit, non-FLIX instruction:

```
format infmt 64 {sl0, sl1}

slot_opcodes sl0 {ABS, RADD}
slot_opcodes sl1 {MYOP, YOP}

// Inst is the predefined 24-bit instruction slot
slot_opcodes Inst {RADD}
```

Thus, for the preceding TIE description, all the following instructions are legal:

```
{RADD, MYOP}   // RADD paired with MYOP as a 64-bit FLIX instruction
{RADD, YOP}    // RADD paired with YOP as a 64-bit FLIX instruction
{RADD, NOP}    // RADD paired with NOP as a 64-bit FLIX instruction
{RADD}         // RADD issued as a 24-bit instruction
```

In this example, RADD can be paired with MYOP or YOP when possible to execute two operations with one instruction. If such a pairing opportunity does not exist, RADD can be paired with NOP, which is the “no operation” instruction automatically available in all slots. However, in this situation it is advantageous to issue RADD in its 24-bit form because this occupies less code space and provides the same functionality.

In the following example, the designer’s intent is to use the Xtensa ISA instructions ADD and SUB in slot slota, and the designer-defined instruction add64 in slot slotb.

```
format f64 64 {slota, slotb}

regfile XR 64 16 x
operation add64 {out XR sum, in XR in0, in XR in1} {} {
     assign sum = in0 + in1;
}

// Both declarations are incorrect because of case mismatch
slot_opcodes slota {add, sub}
slot_opcodes slotb {ADD64}
```

The slot_opcodes statement for slota is incorrect because it lists the Xtensa ISA instructions in lower case text. All Xtensa ISA instructions must be used with upper case text. The slot_opcodes statement for slotb is incorrect because it lists the instruc-

tion add64 in upper case text, whereas the instruction is defined using lower case text. All designer-defined instructions must be listed using the same case as that used in their definition.

The next example illustrates the use of instruction groups, including the wide branch instructions. slot0 includes all the instructions in the group xt_alu as defined in Table 11–9 on page 80, and the L8UI instruction. slot1 includes all the 18-bit wide branch instructions as defined in Table 11–10 on page 82.

```
format f64 64 {slot0, slot1}

slot_opcodes slot0 {xt_alu, L8UI}
slot_opcodes slot1 {xt_widebranch18}
```

Note that it would be illegal to define a subset of the wide branch instructions in any slot — these instructions can only be included as a group. Thus the following statement is illegal and will generate an error from the TIE compiler:

```
slot_opcodes slot1 {BEQ.W18, BALL.W18} // This is illegal
```

12. Import Wire (import_wire) Sections

The import_wire construct defines an input to the Xtensa processor that can be read by designer-defined instructions. Reading an import_wire does not cause any change in the system; that is, the read does not have any side effect. Also, there is no output of the Xtensa processor that indicates when the input is being read. The typical usage of import_wire is to read the status of some external logic, device, or another processor in a system.

12.1 Import Wire Syntax

```
import_wire-def ::= import_wire name width
name ::= a unique identifier
width ::= integer constant specifying the width of the tie port
```

name is the name of the port. *width* specifies the bit-width of the port. An import_wire defined as above can be regarded as an input interface in the TIE description. See Chapter 6, “Interface Signals (interface) Sections” on page 29 for a description of interfaces. The name of the import_wire can be included in the state-interface-list of an operation or in the interface list of an iclass. The name of the import_wire then becomes a valid variable name inside the operation or semantic body that can appear on the right side of any assignment, similar to input interfaces.

12.2 Reading an Import Wire

The read of an import_wire does not have any other side effect; that is, there is no indication of the signal being read, and no external state changes when the signal is read. Also, reads from an import_wire can never stall the processor because the data is not available.

The instruction reading the import_wire can use the data in the E stage. In other words, an import_wire has an implicit use stage of 1. The data is registered before use in any instruction, so that the instruction semantic and the external logic that drives this port have no timing contention in a cycle. The instruction reads the import_wire speculatively; that is, before the instruction commits in its W stage. However, since the read does not modify any external state, it can be read multiple times. If the instruction reading an import_wire gets killed, the data that was read is discarded. If the instruction gets replayed, the import_wire is read again.

An import_wire can be assigned to any output of the instruction directly, or used in computation like any other input operand of the instruction.

12.3 Input Port for import_wire

Declaring an import_wire adds a new input port to the Xtensa processor. The port is a wire or a bus, named TIE_<*name*>. The signal description and timing constraints for these ports can be found in the *Xtensa LX Microprocessor Data Book*.

12.4 Synchronization and Ordering

It is important to understand that reads from import_wire happen earlier in the pipeline than writes to external queues and state exports. The read may happen before a previous write takes effect in the external logic. Therefore, use the synchronization instruction EXTW if a read from an import_wire has a dependency on a previous write to a queue, state export, or memory reference. This subject is also covered in Section 4.3 “Exporting a State” on page 16 and Section 13.5 “Synchronization and Ordering” on page 97.

The Xtensa C compiler ensures that reads from the same import_wire happen in the order as they appear in the program. However, reads from import_wires can be reordered with respect to reads from other import_wires, reads and writes to queues, and memory references. If this is not desirable, then the user program must provide pragmas for the compiler to maintain strict program order for all memory and external I/O accesses. For details, refer to the *Xtensa C and C++ Compiler User’s Guide*.

12.5 Example

The following example defines an import_wire and uses it in an operation. The import_wire reads the status of a device and the operation sets a Boolean register to indicate that the device is ready:

```
import_wire status 2

operation device_read_ok { out BR ready } { in status }
{
assign ready = ( status[0] == 1'b1) ;
}
```

The import_wire declared is two bits wide. The instruction device_read_ok checks the bit 0, and if it is 1, then it sets the Boolean register output ready to 1. The program executing on the Xtensa processor can poll the status of the device as shown in the following example.

```
while( device_read_ok() ) {

/* execute code */

}
```

12.6 Implementation Restrictions

The maximum width of an `import_wire` can be 1024 bits. Also, the total number of TIE ports generated by designer-defined instructions is limited to 1024. This includes ports generated from `import_wire`, `queue`, `state export`, and `lookup`.

The read stage of `import_wire` can be E stage only.

The data read from an `import_wire` can only be assigned to an internal processor state. It cannot be assigned to any interface.

13. Queue (queue) Sections For LX cores

The queue construct defines an interface for designer-defined instructions to read or write data from an external queue. The queue construct does not instantiate a queue inside the Xtensa processor. Instead, it provides an interface to a queue that is external to the processor. The interface assumes that the external queue is a synchronous First In, First Out (FIFO) type device. The queue feature provides a mechanism to perform data transfer to and from an Xtensa processor in a single or multiple processor system with a customized I/O.

13.1 Queue Syntax

```
queue-def ::= queue name width direction
name ::= a unique identifier
width ::= integer constant specifying the width of the queue data
direction := direction of the queue, [in|out]
```

name is the name of the queue interface. *width* specifies the bit-width of the data of the queue interface. *direction* specifies the direction of the queue and can be either `in` or `out`.

A `queue` may be regarded as an `interface` in the TIE description. Refer to Chapter 6 for details on the interface construct. An instruction that accesses a queue must include the name of the queue in the state-interface-list of its `operation` declaration. The name of the queue then becomes a valid variable name inside the operation or semantic body.

13.2 Description of Input Queue

An input queue in TIE defines an interface to read data from an external queue.

13.2.1 Input Queue Ports

For an input queue, the following ports are generated in the Xtensa processor:

```
input [width-1:0] TIE_<name>
input TIE_<name>_Empty
output TIE_<name>_PopReq
```

These ports are automatically derived based on the queue description, and are illustrated in Figure 13–3. The port `TIE_<name>` is the input port to the Xtensa processor that is connected to the data output of the external queue and has the same name and width as specified in the queue declaration. The port `TIE_<name>_Empty` is a 1-bit input port to the Xtensa processor that is connected to the Empty output of the queue. The port `TIE_<name>_PopReq` is a 1-bit output of the Xtensa processor that is connected to the Pop (or read request) input of the queue.

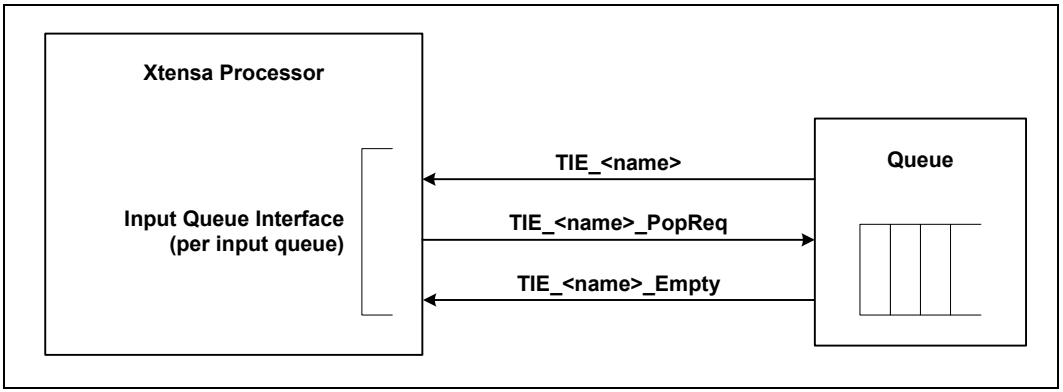

Figure 13–3. Input Queue

13.2.2 Input Queue Interface Protocol

The input queue interface should be connected to a synchronous FIFO type device. A high value on the Empty output of this queue indicates that there is no data in the queue. When Empty is low (that is, the queue is not empty), the queue should present valid data on its data port. A high value on the PopReq signal indicates to the queue that the Xtensa processor wishes to read data from the queue. If PopReq is high and Empty is low, the Xtensa processor latches the data on the data port of the queue, the read operation is completed, and the queue state should change in the next clock cycle. Conversely, if Empty is high when PopReq is high, no data transfer takes place. If the queue becomes empty as a result of a pop, the queue should change Empty in the following clock cycle. There should be no direct combinational path from PopReq to Empty in the external queue design.

The queue interface is synchronous in the sense that the data transfer takes place at the rising edge of the clock. Refer to the *Xtensa LX Microprocessor Data Book* for timing diagrams and additional information on connecting devices to this interface.

13.2.3 Reading an Input Queue

An instruction reads a queue if the queue name appears in the interface list of its `operation` or `iclass` declaration. The data read from an input queue is available to the instruction semantic in the M stage. Thus the implicit use stage of an input queue is two for a 5-stage pipeline and three for a 7-stage pipeline. The Xtensa processor automatically does the actual read or pop of the external queue prior to this cycle, and the data read from the queue is registered before use in the semantic, thus allowing the semantic to use the full cycle for computation. It is important to note that the PopReq signal is automatically generated and not written by the instruction.

If the Empty signal is asserted, indicating that the queue is empty, the instruction reading the queue will stall until Empty is low. It is possible to write instructions that determine whether the next read from an input queue will stall, and execute the read only if there is no stall. This is explained in more detail in Section 13.6.2 on page 101.

The instruction reading an input queue causes the external queue to be popped before the instruction commits; that is, the queue read is speculative. The instruction can get killed in the pipeline after having popped the data from the queue without using the data. If this happens, the data read from the queue is held in a buffer inside the Xtensa processor. The next instruction reading from the queue reads this buffer and no data is popped from the external queue. The speculative nature of the read is handled by providing this buffer inside the Xtensa processor, and the external queue does not need to have a speculative design. The queue interface is read early to allow for more cycles to perform computation on the data read from the queue without affecting the latency of the operation. The system designer should be aware of the fact that data popped from a queue may be used by an instruction after an arbitrary number of cycles.

13.2.4 Example: Reading One Queue

In the following example, an instruction reads from a queue and performs some computation on the data before writing to a user-defined register file. Note that the queue data width is independent of the width of any processor memory or peripheral interface, and can be of any arbitrary width subject to the limitations of Section 13.8.

```
queue INPQ 80 in

regfile DATA 64 16 d
operation STREAMIN { out DATA X } { in INPQ }
{
//Read INPQ once and assign value to different wires
wire [15:0] header = INPQ[79:64];
wire [31:0] data0 = INPQ[63:32];
wire [31:0] data1 = INPQ[31:0];
//computation
wire [31:0] result = TIEadd(data0, data1, 1'b0);
```

```
assign X = {16{header[0]}, result, header};
}

schedule streamin { STREAMIN } { def X 3; }
```

Note that for a 5-stage pipeline, the earliest `def` stage for `X` can be 2 because that is when the queue data `INPQ` is available in the semantic. If a schedule section is not specified in the TIE source, the TIE compiler automatically generates a schedule with a def of 2 for the output `X` of the instruction. If the schedule section defines the def stage of `X` to be less than 2, the TIE compiler issues an error. Since the data read from the queue is registered before it is used in the semantic, it is possible to use the full cycle for computation without causing timing problems.

The following instruction sequence illustrates the speculative read from an input queue. The instruction `STREAMIN` is preceded by an instruction that causes a replay of the pipeline, thus killing the `STREAMIN` instruction before it commits, but after it has popped the queue `INPQ`. In this case, when the `STREAMIN` instruction is replayed, no data is popped from the external queue. Also, consider the scenario where the queue becomes empty after the data was popped. This will not cause a stall when the instruction replays because it reads the data in the internal buffer.

```
isync          // Xtensa ISA instruction that causes a replay
STREAMIN d3
```

13.2.5 Example: Reading Multiple Queues

In the following example, an instruction reads from two input queues, concatenates their data values, and writes the result to an output register.

```
regfile DATA 40 16 dat
queue DATA1 20 in
queue DATA2 20 in

operation MDATA {out DATA X} {in DATA1, in DATA2}
{assign X = {DATA1, DATA2};}
```

In this example, both queues `DATA1` and `DATA2` will be popped when the instruction `MDATA` is executed. Thus the instruction will stall if either queue is empty.

When one or more input queue name(s) are listed in the *state-interface-list* of an operation statement as in the preceding example, all the queues will be popped and the data from all the queues is assumed to be consumed by the instruction. This is independent of whether the queue data is actually used in the computation of any output value. Refer to Section 13.7.1 on page 103 for an illustration of how input queue data can be selectively consumed based on a run-time condition.

13.3 Description of Output Queue

An output queue in TIE defines an interface to write data to an external queue.

13.3.1 Output Queue Ports

For an output queue, the following ports are generated:

```
input TIE_<name>_Full
output [width-1:0] TIE_<name>
output TIE_<name>_PushReq
```

These ports are automatically derived based on the queue description and are illustrated in Figure 13–4. The data port `TIE_<name>` is the output of the Xtensa processor that is connected to the data input of the queue and has the same name and width as specified in the queue declaration. The port `TIE_<name>_Full` is a 1-bit input to the Xtensa processor that should be connected to the Full output of the queue. The port `TIE_<name>_PushReq` is a 1-bit output of the Xtensa processor that should be connected to the Push (or write enable) input of the queue.

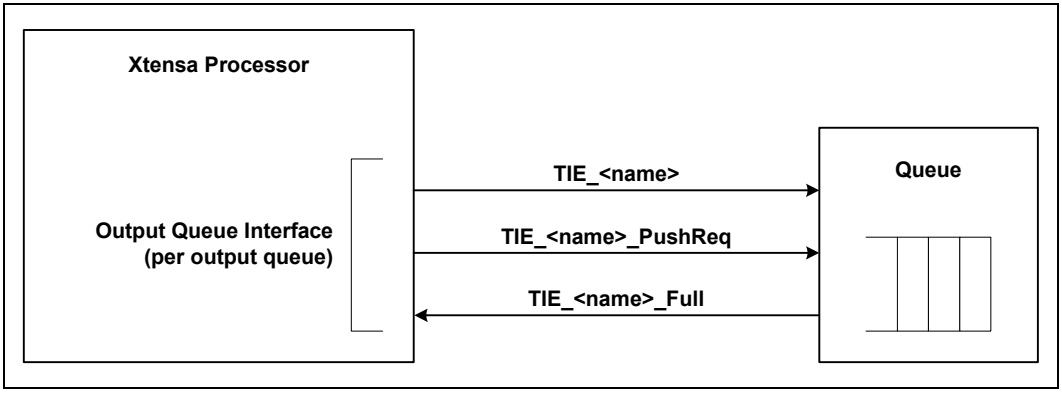

Figure 13–4. Output Queue

13.3.2 Output Queue Interface Protocol

The output queue interface should be connected to a synchronous FIFO type device. A high value on the Full output of this queue indicates that the queue cannot receive any more data. A high value on the PushReq signal indicates to the queue that the Xtensa processor wishes to write data to the queue. When PushReq is high, the Xtensa processor will present valid data on the data port of the queue interface. If PushReq is high and Full is low, the Xtensa processor assumes that the queue has latched the data and the write operation is completed. Conversely, if Full is high when PushReq is high, no data

transfer takes place. If the queue becomes full as a result of a push, the queue should change to Full in the next clock cycle. There should be no direct combinational path from PushReq to Full in the external queue design.

The queue interface is synchronous in the sense that the data transfer takes place at the rising edge of the clock. Refer to the *Xtensa LX Microprocessor Data Book* for timing diagrams and additional information on connecting devices to this interface.

13.3.3 Writing an Output Queue

An instruction writes a queue if the queue name appears in the interface list of its `operation` or `iclass` declaration. The output of the instruction is written into a register at the end of the M stage, and is written to the external queue in the next cycle if Full is low. This means the implicit def stage of an output queue in an instruction is the M stage, and the earliest cycle in which the data is written to the external queue is the W stage. It is important to note that the PushReq signal is automatically generated and not written by the instruction.

If the Full signal is asserted, indicating that the queue is full, and the internal buffer or register already has data waiting to be written to the queue, then the instruction writing the queue will stall. The instruction will proceed when Full becomes low and the data in the internal register has been written to the external queue. It is possible to write instructions to check whether the next write to an output queue will stall, and execute the write only when there is no stall. This is explained in more detail in Section 13.6.1 on page 100.

13.3.4 Example

In the following example, an instruction reads from two input queues and writes the output of the computation to an output queue.

```
queue INDATA1 40 in
queue INDATA2 40 in
queue OUTDATA 40 out

operation QADD{} { in INDATA1, in INDATA2, out OUTDATA}
{
assign OUTDATA = TIEadd(INDATA1, INDATA2, 1'b0);
}
```

Note that in the preceding example, no schedule section is specified in the TIE source. The implicit schedule based on the read and write stages of queues will be as follows:

```
use INDATA1 2;      // M stage for a 5-stage pipeline
use INDATA2 2;
```

```
def OUTDATA 2;
```

13.4 Timing of Queue Signals

The data input from the external queue is registered before being used in an instruction, and the output of an instruction is registered before writing it to the external queue. This ensures that there will be no timing contention between the Xtensa processor and the external queue logic within a cycle. The control inputs Full and Empty, and control outputs PushReq and PopReq are not registered. The timing constraints on these signals are set accordingly. Refer to the *Xtensa LX Microprocessor Data Book* for details, including timing diagrams of data transfer over these interface signals.

13.5 Synchronization and Ordering

Instructions that read or write TIE queues, access a TIE lookup, read from an `import_wire`, or write to an exported state all have effects that are visible external to the Xtensa LX processor. It is possible that the order of execution of these instructions in the processor is different from the order of events visible outside the processor. In many situations, this ordering behavior is not a problem. For example, the system may not care about the relative order of instructions that access two unrelated queues. Conversely, you may have a system in which this order is important, as illustrated in the example in Figure 13–5. Consider the following instruction sequence. There is a series of writes to a queue `OUTQ` followed by a write to an exported state FINISH to signal to the external logic that the processor has completed its writes to `OUTQ`.

```
WOUTQ(0x1);
WOUTQ(0x2);
WFINISH(0x1);
```

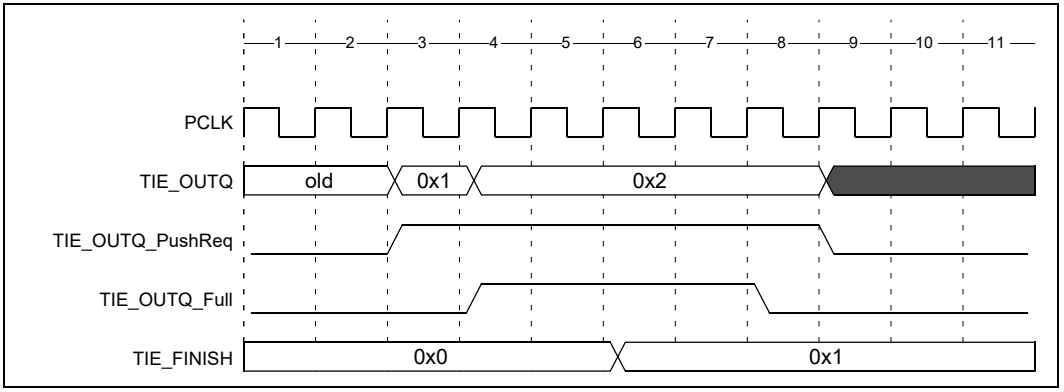

Figure 13–5. Queue Write Followed by State Export without Synchronization

As shown in Figure 13–5, the write from the first instruction WOUTQ succeeds and as a result of this the queue OUTQ becomes full. The write from the second instruction WOUTQ cannot be completed until OUTQ_Full becomes low. The output of the second WOUTQ instruction remains in the internal buffer. This is followed by the instruction WFINISH, which writes the state FINISH. As soon as WFINISH is past its commit stage, the change in the state value is reflected on the Xtensa port. Thus the external logic is signaled that the writes from the Xtensa processor to the queue OUTQ are completed, although the last write is not yet completed.

To ensure that the system works correctly, the write to the state FINISH must be executed after all writes to OUTQ are completed; that is, they have left the Xtensa processor. In other words, your system requires that the externally visible effect of the earlier instruction be seen before the effect of the subsequent instruction is seen. For these situations, the Xtensa ISA defines a special synchronization instruction called External Wait or EXTW. This instruction ensures that all instructions ahead of it have gone past the commit stage (the W stage), and that all externally visible effects of those instructions have been presented to the external logic before any subsequent instruction executes.

In the preceding example, if EXTW is inserted in the program between the last write to OUTQ and the write to the state FINISH, all the writes to OUTQ will complete before the value of FINISH changes. This is shown in Figure 13–6.

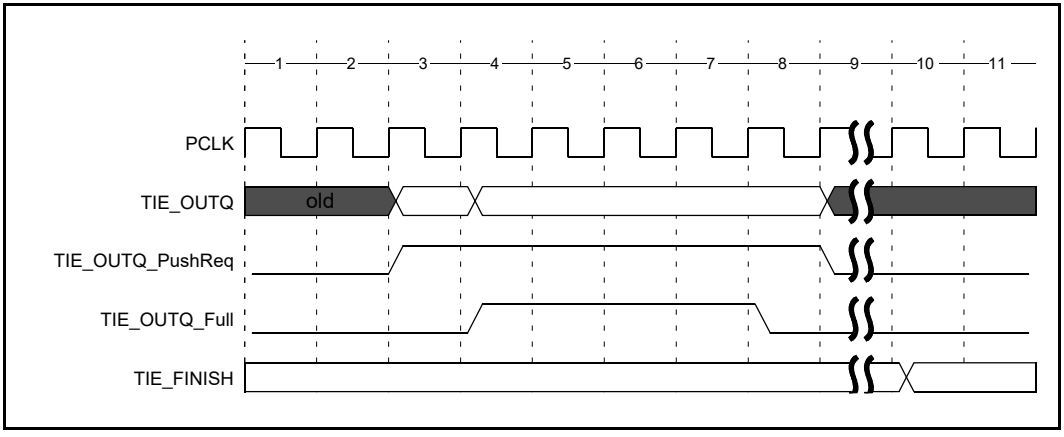

Figure 13–6. Queue Write Followed by State Export with Synchronization

You must use the EXTW instruction at synchronization points in your assembly program where the ordering of external events is important. Note that the EXTW instruction ensures that the data from previous instructions leaves the Xtensa LX processor before subsequent instructions execute. However, there may be additional delays in your system before this data is processed by the consumer of the data; this delay is not accounted for by EXTW. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details on the EXTW instruction.

Note that the ordering behavior described above also holds true between instructions that access TIE ports/queues/lookups and load/store instructions. The externally visible effects of TIE ports/queues/lookups and any access on XLMI or PIF are also not guaranteed to be seen in order.

You must also pay special attention to ordering and synchronization issues when writing and compiling C code for the Xtensa processor. The Xtensa C Compiler (XCC) ensures that instructions that read and write from the same queue execute in the order that they appear in the program. However, reads or writes to a queue can be reordered with respect to reads or writes to other queues, accesses to TIE lookups, reads from `import_wires`, state exports, and memory references. If this is not desirable, then the user program must provide pragmas for the compiler to maintain strict program order for all memory and external I/O accesses across the pragma points. Alternately, a command line switch may be used to enforce this ordering for the whole program. The XCC compiler offers multiple options for controlling the ordering of instructions and their externally visible effects; refer to the *Xtensa C and C++ Compiler User's Guide* for details.

Reads from input queues deserve special mention in this discussion on ordering and synchronization. An input queue is popped speculatively, before the instruction reading the data has reached the commit stage. The instruction may be killed for a variety of reasons, and hence the data popped from the queue may not be consumed at that time. This data is buffered in an internal buffer, to be used by the next instruction that attempts to read from the same queue. This may happen after an arbitrarily large amount of time from when the queue was popped. As a result of this behavior, it is not possible to maintain strict order of externally visible events with respect to input queues, even when synchronization primitives are used.

13.6 Testing Queue Status

When a system is running at full bandwidth, stalls due to empty or full queues should be unlikely and it is not efficient to check the status of a queue before every read or write. However, it may be necessary to do so when switching a task or in other scenarios where it is important to determine that the next read or write will not stall. It may seem that an instruction could read the Empty or Full signal and determine the queue status, but this is not the case. For example, if the internal buffer for an input queue has data that was popped, but not consumed by an instruction, the next read from the queue will not stall regardless of the external queue being empty.

To solve this problem, an interface called <*name*>`NOTRDY` is automatically generated for each queue in the TIE description. This interface can be read by an instruction to determine if the next read or write will stall. The behavior of this interface is strictly defined as the following: If the value of <*name*>`_NOTRDY` is low, then the next read or write to the

queue <*name*> is guaranteed not to stall. Note that this applies to the next read or write only, and not subsequent reads or writes. Conversely, a high value of the NOTRDY interface does not guarantee that the next read or write will stall due to the queue.

The use stage of the NOTRDY interface is the same as the use or def stages of the input or output queue respectively (that is, the M stage[1]).

The correctness of this interface relies on the fact that an input queue can go from being not empty to empty only as a result of a PopReq from the Xtensa processor, and an output queue can go from being not full to full only as a result of a PushReq from the Xtensa processor. If that is not the case, then the value of the NOTRDY interface does not correctly reflect the status of the queue.

It is important to understand that the interface <*name*>_NOTRDY cannot be read in the same instruction that reads or writes a queue and selectively assert PopReq or PushReq depending on its value. As mentioned earlier, the PushReq and PopReq signals are not asserted in the instruction. Correct usage of the NOTRDY interface is illustrated in the examples that follow.

13.6.1 Output Queue NOTRDY

The following example shows how to use the NOTRDY interface signal for an output queue to ensure that writes to the queue do not stall. The operation WRITEQ writes the output queue OUTQ. The operation OUTQFULL reads the interface OUTQ_NOTRDY and sets a Boolean register.

```
queue OUTQ 32 out
operation WRITEQ {in AR dat} { out OUTQ }
{
     assign OUTQ = dat;
}
operation OUTQFULL { out BR b } { in OUTQ_NOTRDY }
{
     assign b = OUTQ_NOTRDY;
}
```

The following sample C code running on the Xtensa processor checks the value of OUTQ_NOTRDY and executes the WRITEQ instruction only when the value is low to ensure that the write does not cause the processor to stall if the queue is not ready to receive data.

```
while ( ! OUTQFULL() ) {
     WRITEQ(val);
}
```

1. In prior releases of the TIE compiler, the use stage of the NOTRDY interface for output queues was the E stage.

Note that the compiler understands the association of the NOTRDY interface with the queue. Therefore, reads from the interface <*name*>_NOTRDY are not reordered with respect to reads and writes to the queue <*name*>.

13.6.2 Input Queue NOTRDY

The following example shows how to use the NOTRDY interface signal for an input queue to flush all remaining entries from a queue. Consider the following example of an Xtensa processor reading data from a queue DATA:

```
queue DATA 32 in
operation GET_DATA { out AR a } { in DATA }
{
     assign a = DATA;
}
```

Suppose the Xtensa processor wants to stop the current stream of data from the queue DATA, flush the external queue and internal buffer and start over. This can be achieved using the following scheme. In the example, we define an exported state, STOP_DATA, which signals to the external logic to stop writing data to the queue. The example also has an instruction DATA_EMPTY, which checks the value of the interface DATA_NOTRDY and sets a Boolean register.

```
queue DATA 32 in
state STOP_DATA 1 1'b0 add_read_write export
operation DATA_EMPTY { out BR b } { in DATA_NOTRDY }
{
     assign b = DATA_NOTRDY;
}
```

The following sequence of instructions running on the Xtensa processor will stop the current stream of data and flush the queue. First, the state STOP_DATA is written with a value of 1. When this write is detected by the external logic, it should stop writing new values to the queue DATA. The queue and internal buffers can be flushed by checking the value of DATA_NOTRDY and executing an instruction to read the queue until there is no more data to read. The exact C code sequence is as shown in the example. Notice that the write to the state STOP_DATA is followed by a synchronization pragma flush. This ensures that the Xtensa C Compiler (XCC) will not reorder instructions relating to TIE ports and queues across this barrier and will also insert the synchronization instruction EXTW. EXTW will ensure that the write completes and is visible on the external port before the subsequent instructions execute. Then the code checks the value of DATA_NOTRDY and executes a read from the queue until the value is 1.

```
/* Tell the external logic to stop writing data to the queue */
WUR.STOP_DATA(1);
/* Synchronization to make sure the new value of STOP_DATA
```

```
is seen by the external logic
*/
#pragma flush

/* Read (and discard) data from queue until there is no data*/
{
     unsigned tmp;
     while ( ! DATA_EMPTY() ) {
          tmp = GET_DATA();
     }
}
```

Note that in this example, we are relying on the high value of the NOTRDY signal for an input queue to indicate that there is no more data to read from the queue. This is possible because the external logic is not writing more data into the queue. Refer to Section 13.7.5 for an alternative implementation of input queue flush.

13.7 Conditional Queue Access

In the discussion of queues so far, it is assumed that an instruction, once issued, always reads or writes data from or to a queue. Section 13.6 shows how to test the status of the queue before issuing an instruction that accesses the queue. However, it may be useful to conditionally access a queue, based on the external queue's status or any other run-time condition. This may result in better code performance in some applications, as compared to the *test and branch* approach.

A 1-bit interface called <*name*>_KILL is automatically generated for each queue in the TIE description. This interface can be written by an instruction to specify if the queue read or write is being killed or not. If this interface of an output queue is assigned a logical 1 value, then the write of the queue is cancelled. This is similar to the kill signal associated with output operands of an instruction, which allows for predication based on a run-time condition. For an input queue, assigning a logical 1 to the KILL interface means that the queue data will not be consumed. Either the external queue will not be popped, or the popped data will be buffered inside the processor for use by a subsequent instruction. If a queue access is killed, the instruction accessing the queue will not stall even if the input queue is empty or the output queue is full.

To use the KILL interface, it must be declared as an output operand in the state-interface list of an operation that accesses the queue. It is illegal to use the KILL interface of a queue in an operation that does not access the queue. The def stage of the KILL interface is the same as the use or def stage of an input or output queue respectively (that is, the M stage). Assignment to the KILL interface can be made from any legal expression of the input operands of the instruction, with one restriction related to the use of the NOTRDY interface as described in Section 13.7.3 on page 104.

13.7.1 Input Queue KILL

The following example shows how to use the KILL interface for an input queue to conditionally read the queue data.

```
queue DATA0 32 in
queue DATA1 32 in

operation GET_DATA { out AR reg, in BR2 select }
     { in DATA0, out DATA0_KILL, in DATA1, out DATA1_KILL }
{
     wire select0 = (select == 2'b01);
     wire select1 = (select == 2'b10);
     wire kill0 = ~select0;
     wire kill1 = ~select1;
     assign reg = (select0) ? DATA0 : DATA1;
     assign DATA0_KILL = kill0;
     assign DATA1_KILL = kill1;
     assign reg_kill = kill0 && kill1;
}
```

In the example, the instruction GET_DATA either reads a data value from one of two input queues and assigns this value to its output operand reg, or leaves the value of reg unchanged (effectively becoming a NOP in this case). The operation is controlled by the 2 bit select input from the Boolean register file. When the select signal has the value 2'b01, data from queue DATA0 is assigned to reg, and access to the queue DATA1 is killed by assigning a logical 1 value to DATA1_KILL. Similarly, when the select signal has the value 2'b10, data from queue DATA1 is assigned to reg, and access to the queue DATA0 is killed. For other values of the select signal, accesses to both the input queues are killed, and the value of reg is unchanged.

As described in previous sections, the external queue is popped before the instruction semantic reads the data or assigns the kill interface. This means that if the external queue has data (and the internal buffer is empty), it will be popped even if the queue access is later killed in the instruction. In this case, the data stays in the internal buffer and is used by the next access. If the queue is empty and the queue access is killed, the PopReq signal is de-asserted and the instruction no longer blocks or stalls for the data. In this case, the queue data input to the semantic is undefined.

Note that when a queue access is killed, this does not automatically kill the updates of the output operands of the instruction. In this example, the update of the output reg is killed by assigning the signal reg_kill to a logical 1 when none of the queues are read.

13.7.2 Output Queue KILL

The following example shows how to use the KILL interface for an output queue to conditionally write the queue data.

```
queue STREAM 128 out
state DATA 128
state COUNT 2 2'b00
operation VMUL_WRITE { in AR a, in AR b }
     { inout COUNT, inout DATA, out STREAM, out STREAM_KILL }
{
     wire [31:0] result = TIEmul(a[15:0], b[15:0], 1'b0);
     wire [7:0] samt = COUNT << 3'd5;
     wire done = (COUNT == 2'd3);
     assign DATA = ( DATA << samt ) | result;
     assign STREAM = DATA;
     assign STREAM_KILL = ~done;
     assign COUNT = COUNT + 1'b1;
}
```

In this example, the state DATA holds a vector of four 32-bit results of multiplication from the instruction VMUL_WRITE. This instruction also writes the vector output to the output queue STREAM after a full vector is computed. Therefore, the output queue is written every fourth time the instruction VMUL_WRITE is executed. The state COUNT keeps the iteration counter, and has a value of zero upon reset. It determines if the vector is computed and ready to be written out to the queue. The interface STREAM_KILL is assigned to 1 unless the counter COUNT is 3. Without conditional queue access, the program would execute four instructions to create the 128-bit DATA value, and then execute a separate queue push instruction. With conditional queue access, the program becomes more efficient as a result of combining the computation and the push operation.

Note that if the output queue is full when the queue write is killed, the instruction will not stall. The VMUL_WRITE instruction will only stall when the queue is full and STREAM_KILL is false.

13.7.3 Non-blocking Queue Access

The read and write of a queue interface is blocking in nature; that is, once issued, the instruction reading or writing the queue stalls until data can be read or written.
Section 13.6 described how to test the queue status before issuing a queue read or write instruction. This is useful if you want to issue a queue instruction only if it is guaranteed to not stall. Another option would be to perform a non-blocking queue access by using the KILL interface of a queue. This is achieved by assigning the NOTRDY interface to the KILL interface. By doing so, the queue access is killed when the queue is empty and there is no data to be read. The designer is responsible for setting a status in the processor indicating the success or failure of the queue access and testing that status

as needed. In the case of a non-blocking read of an input queue, the updates of the outputs should also be cancelled appropriately. This is because the data returned from an input queue access that is killed is undefined.

The following example shows a non-blocking read of an input queue.

```
queue DATAIN 32 in
operation READ_NOBLOCK { out AR a, out BR status }
     { in DATAIN, in DATAIN_NOTRDY, out DATAIN_KILL }
{
     assign a = DATAIN;
     assign DATAIN_KILL = DATAIN_NOTRDY;
     assign a_kill = DATAIN_NOTRDY;
     assign status = ~DATAIN_NOTRDY;
}
```

In this example, the queue access to `DATAIN` is killed when there is no data to be read. The update of the output register, a, is also killed under the same condition, because the read data is not defined when the queue access is killed. The Boolean output `status` indicates if the queue access is successful. The instruction `READ_NOBLOCK` reads the input queue when it has data, and does nothing except setting the status bit when the queue is empty. The instruction never stalls on an empty queue.

13.7.4 Peek of Input Queue

In some applications, it may be useful to look at the queue data without popping it. This is also known as peek. The `KILL` interface provides a way to do a peek of an input queue, as illustrated in the following example.

```
queue DATAIN 32 in
operation CHECK_HDR { in AR check, out BR match }
     { in DATAIN, in DATAIN_NOTRDY, out DATAIN_KILL }
{
     wire [7:0] hdr = DATAIN[31:24];
     assign match = ( hdr == check[7:0] );
     assign DATAIN_KILL = ~DATAIN_NOTRDY;
}
```

The instruction `CHECK_HDR` checks the upper bits of data from input queue `DATAIN`, and sets the Boolean output `match` to true if these bits match the bits of input `check`. The instruction looks at the queue data but does not consume it, hence the term *peek*. This is done by the last assignment in the operation body. If the queue has data, then the interface `DATAIN_NOTRDY` will be false, which means the output `DATAIN_KILL` will be true. Thus the `CHECK_HDR` instruction looks at the queue data, but does not consume it because it also kills the queue access. Note that the `CHECK_HDR` instruction is blocking, such that if the queue `DATAIN` is empty, the instruction will stall until there is data. This

is done to prevent peeking at invalid data. The instruction will stall until there is valid data, but once it peeks at the valid data, the access is killed and hence the data is not consumed. Note also that the data will be popped from the external queue and held in the internal buffer to perform the peek.

A non-blocking version of the peek is shown below.

```
queue DATAIN 32 in
operation CHECK_HDR_NOBLOCK { in AR check, out BR match }
     { in DATAIN, in DATAIN_NOTRDY, out DATAIN_KILL }
{
     wire [7:0] hdr = DATAIN[31:24];
     assign match = ( hdr == check[7:0] ) && ~DATAIN_NOTRDY;
     assign DATAIN_KILL = 1'b1;
}
```

The instruction CHECK_HDR_NOBLOCK always assigns the interface DATAIN_KILL to 1. If the queue DATAIN had data, that is, the interface DATAIN_NOTRDY is 0, the upper bits are compared with the input check to set the output match. If there is no data to read, then the peek data is undefined and in this case match should be 0. Note that because the read of the queue is always killed, the instruction CHECK_HDR_NOBLOCK does not stall when the queue is empty.

13.7.5 Flushing Input Queue

It may be necessary to empty the internal buffer of an input queue when the processor switches to a different task. As shown in Section 13.6.2, the NOTRDY interface can be used to check if there is data in an input queue and flush the contents of it by reading the queue. A variation of the previous example is shown here using the KILL interface.

```
queue DATA 32 in
operation DISCARD_DATA { out AR d, out BR done }
     { in DATA, in DATA_NOTRDY, out DATA_KILL }
{
     assign d = DATA;
     assign DATA_KILL = DATA_NOTRDY;
     assign done = DATA_NOTRDY;
}
```

The instruction DISCARD_DATA reads the input queue DATA until there is no data to read, as indicated by DATA_NOTRDY. When there is no more data, the read is killed and the status bit done is set to 1. Because the instruction DISCARD_DATA does not block when there is no data in the queue, the checking of NOTRDY and the read can be combined. Thus the C code of Section 13.6.2 can be rewritten using the DISCARD_DATA instruction as follows:

```
/* Tell the external logic to stop writing data to the queue */
WUR.STOP_DATA(1);
/* Synchronization to make sure the new value of STOP_DATA
is seen by the external logic
*/
#pragma flush

/* Read (and discard) data from queue until there is no data*/
{
     unsigned tmp;
     xtbool done = false;
     while ( !done ) {
          DISCARD_DATA(tmp, done);
     }
}
```

13.8 Implementation Restrictions

- The maximum data width of a `queue` is 1024 bits. The total number of TIE ports generated by designer-defined instructions is also limited to 1024. This includes ports generated from `import_wire`, `queue`, `lookup`, and `state` export.
- Input queue read data is available in the instruction semantic in the M stage of the processor pipeline. Similarly, data to be written to an output queue must be defined in the instruction semantic in the M stage.
- If the read access to an input queue is killed using the KILL interface, the read data is undefined.
- When using the `NOTRDY` interface to kill a queue access, only a simple assignment of the `NOTRDY` interface to the `KILL` interface is allowed. No other expression using the `NOTRDY` interface is allowed in the assignment to the `KILL` interface.
- When running a program on the Xtensa processor, if a write to an exported state is followed by an access to a TIE queue, the hardware ensures that the new state value is visible outside the processor before the queue access is performed. This is done by inserting an appropriate number of bubbles between the two instructions.
- In a FLIX design, you cannot create a bundle with two or more operations that read the same input queue or write to the same output queue.
- The input and output queue interface has been designed such that only a synchronous FIFO type device can be connected to this interface. Furthermore, no other device should be reading from an input queue or writing to an output queue connected to this interface. Using this interface in a manner inconsistent with these restrictions may result in unexpected or incorrect behavior.

Restrictions are imposed on how TIE queues can interact with other TIE ports and memory interfaces. These restrictions are necessary to ensure the correct functionality of TIE queues in a pipelined processor implementation, and to match the use/def schedules of different interfaces. The following restrictions and guidelines apply to an instruction that reads an input queue and writes the data to an output interface:

- Data read from an input queue can be used to drive a TIE lookup address output only if the lookup def stage is equal to the W stage.
- Data read from an input queue can be used to drive the MemDataOut interface for a store operation.
- Data read from an input queue can be written to an output queue.

The following restrictions and guidelines apply to an instruction that accesses an input data interface and writes the data to an output queue:

- Data read from an import_wire interface cannot be written to an output queue.
- Data read from a TIE lookup input can be written to an output queue only when the lookup use stage is equal to the M stage.
- Data read from the MemDataIn interface of a load operation can be written to an output queue.
- Data read from an input queue can be written to an output queue.

All register files and TIE state (including exported state) are always a valid source and destination for use with queues.

14. Lookup (lookup) Sections For LX cores

The lookup construct defines a set of interfaces for designer-defined instructions to send a request or output to an external device and read the response from the device a fixed number of cycles later. The request and response are treated as an atomic transaction within the same TIE instruction. This feature provides a mechanism to perform a *lookup* operation to an external ROM or other memory device, hence the name. However, its use is not restricted to interfacing with memory devices, and it does not use the memory interface or the load-store data path of the processor. This construct creates a new set of primary interfaces between the Xtensa processor and the external logic to which it interfaces.

14.1 Lookup Syntax

```
lookup-def ::= lookup name {output-width, def-stage}
                                    {input-width, use-stage} [rdy]
name ::= a unique identifier
output-width ::= integer constant specifying bit-width of output
def-stage ::= stage_expr specifying the pipeline stage in which the
output is written
input-width ::= integer constant specifying bit-width of input
use-stage ::= stage_expr specifying the pipeline stage in which the
input is read
stage_expr ::= integer constant or
               symbolic-stage [+/- integer constant]
symbolic-stage ::= keyword specifying pipeline stage
rdy ::= an optional argument to specify that there is a control
interface to indicate the readiness of the external device
```

name is the name of the lookup interface. The *output-width* is the bit-width of the address or the output interface, and *def-stage* is the pipeline stage in which the instruction sends the output or request to the external device. The *input-width* is the bit-width of the response or input data from the external device, and *use-stage* is the pipeline stage in which the input data is read by the instruction. If the optional argument *rdy* is present, the lookup interface will have another control input that indicates to the Xtensa processor when the external device is ready to accept a request. The pipeline stages can be written as numeric values specific to the processor configuration, or as symbolic stages (or expressions thereof). Symbolic stages allowed are Estage, Mstage, and Wstage.

14.2 Description of Lookup

This section describes the new interfaces that are added to the Xtensa processor when a lookup is instantiated in a TIE description. It also provides a description of how to use the lookup construct, along with some examples.

14.2.1 Lookup Ports

For every lookup construct in your TIE design, the following ports are generated in the Xtensa processor:

```
output [output-width-1:0] TIE_<name>_Out
output TIE_<name>_Out_Req
input TIE_<name>_Rdy (optional)
input [input-width-1:0] TIE_<name>_In
```

These ports are automatically derived based on the lookup description, and are illustrated in Figure 14–7. `TIE_<name>_Out` is the output port of the Xtensa processor that is connected to the address input of the external device. `TIE_<name>_In` is the input port to the Xtensa processor that is connected to the data output of the external device. `TIE_<name>_Out_Req` is a 1-bit output of the Xtensa processor that is asserted when a lookup request is being made by the Xtensa processor. It can be used to enable the external device when asserted, and to disable the device (for power savings or other reasons) when de-asserted. `TIE_<name>_Rdy` is a 1-bit input to the Xtensa processor that is present only when the lookup definition has the *rdy* argument. It can be connected to an arbitration device or any other logic that can de-assert this signal when the external device is not ready to accept any requests from the Xtensa processor.

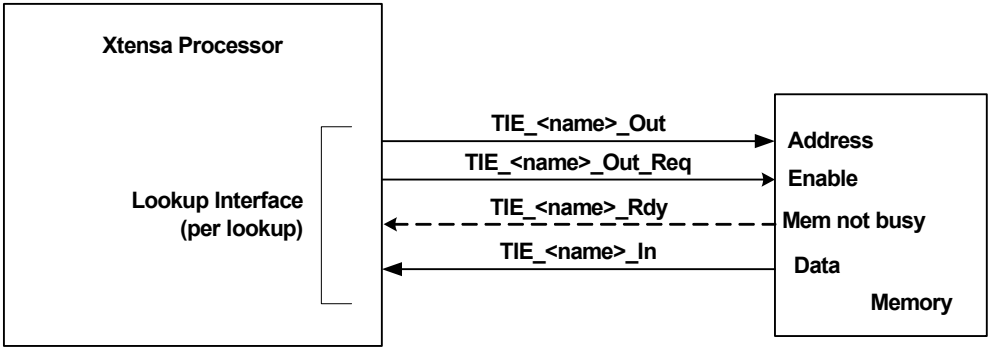

Figure 14–7. Lookup Ports

14.2.2 Lookup Interface Protocol

The lookup interface performs table lookups from an external memory. It can also be used to interface to other external devices that can accept an address and provide data in response to this request, as long as the device can handle speculative reads as explained in Section 14.2.3.

A high value on the TIE_<*name*>_Out_Req port indicates that the Xtensa processor is sending a valid address or request on the TIE_<*name*>_Out port to the external device. The TIE_<*name*>_In input port will then be read after N cycles, where N is the latency of the lookup, and is defined as (*use-stage - def-stage).* Thus the latency of the lookup is specified in its declaration, and is fixed. The Xtensa processor can send a lookup request every cycle. The external device should be able to pipeline requests when the latency is higher than one cycle.

The optional parameter *rdy* adds the TIE_<*name*>_Rdy input port to the processor. This port is used by the external device, to indicate to the Xtensa processor whether it is ready to accept the request or not. The Xtensa processor will assert the output port Req and read the input port Rdy in the same cycle. When both the request and the ready signal are high in the same cycle, the request is considered to have been accepted by the external device. The input data port will be sampled N cycles later. Thus the Rdy signal is associated only with the request and not with the response. After the external device has accepted the request, it must provide a response after N cycles, where N is the latency of the lookup. If Req is high and Rdy is low, the external device is not accepting the request. In this case, the instruction using the lookup will stall until Rdy is high. During the stall, the Req remains asserted and the address or lookup output value is maintained on the TIE_<*name*>_Out port. The request can be de-asserted if there is a processor exception, including an interrupt.

The optional Rdy input allows the external device to be shared between several Xtensa processors or other blocks by having some arbitration. Cadence recommends that the status of the Rdy signal be pre-determined by the arbitration logic, before the Xtensa processor asserts the Req signal. This will prevent a direct combinational path from Req to Rdy, which is undesirable for timing and physical design considerations. Rdy can also be used when the external device is not pipelined, (that is, it cannot accept a request every cycle). In this case, the device can de-assert Rdy after every request until it is ready to accept the next one.

The lookup interface is synchronous in the sense that the data transfer takes place at the rising edge of the clock. Refer to the *Xtensa LX Microprocessor Data Book* for timing diagrams and additional information on connecting devices to this interface.

14.2.3 Using a lookup

A `lookup` can be regarded as a pair of `interfaces` in the TIE description. Refer to Chapter 6 for details on the interface construct. The output and input interface names (to be used in the TIE instructions that access a lookup) are predetermined by the name of the lookup. The output interface is named <*name*>`_Out` and the input interface is named <*name*>`_In`. An instruction that accesses a lookup must include both of these interfaces in the state-interface-list of its `operation` declaration or in the interface-list of its `iclass` declaration. The names of these interfaces then become valid variable names inside the operation, reference, or semantic body. The output interface can be assigned to and the input interface can appear in the right-hand side of an assignment. Both the output and input interface must be declared and used by the instruction — it is illegal for an instruction to use only one of the two interfaces[1].

The address or output data port is written when the instruction is in the *def-stage* specified in the description of the lookup. The Xtensa processor automatically asserts the Req output in the same stage. It is important to note that the Req signal is automatically generated and not written (assigned to) in the instruction description. The input data is sampled and latched by the Xtensa processor when the lookup instruction is in the *use-stage* specified in the description of the lookup.

The pipeline stages in which the output data is written and the input data is read are both configurable and specified in the lookup definition. Typically, the *def-stage* in which the address or request is written is before the instruction commits, making the request speculative. If the instruction is killed after the request is sent to the external device, the data returned is not consumed (or saved) by the processor. Therefore, the output or request should not have any side-effects on the external device. When ready, the external device should read the output of Xtensa when the Req signal is asserted, and return the data N cycles later. The external device can be disabled when the Req signal is deasserted, thus saving power where possible. However, there should be no side-effect that alters the state of the device based on the request. For example, it would be incorrect to connect any device that counts the number of requests made on this interface, or a device such as a FIFO that is popped as a result of the request.

The behavior of the external device and the lookup interfaces can be simulated using the XTSC or XTMP simulation environment. If the external device is a memory, and the lookup interfaces satisfy certain requirements, the behavior of the memory and the lookup interfaces can be simulated using stand-alone ISS. See Section 21.4 “Lookup Memory Property” on page 183 for a detailed description.

1. If the lookup device is a memory and `lookup_memory` property is specified in the TIE file, the <*name*>_In interface may not be used in the computation portion of the semantic, reference, or operation. However, the interface is still required to be declared in as an input interface to the operation. Details see Section 21.4 on page 183.

14.2.4 Example: Using a lookup

In the following example, a lookup MEM1 with 2-cycle latency is defined such that the request is sent in the E stage and the response is received two cycles later. The address is 16 bits, and the data returned is 140 bits. The instruction MEM1_GET uses the lookup. The address is generated from an AR register and the data returned is written to the state DATA.

```
lookup MEM1 {16, Estage+1} {140, Estage+2} rdy

state DATA 140 add_read_write
operation MEM1_GET{ in AR addr } { out MEM1_Out, in MEM1_In, out DATA }
{
     assign MEM1_Out = addr[15:0];
     assign DATA = MEM1_In;
}
```

In this example, the implicit def stage for the state DATA is 3, because it is the target of a lookup whose *use-stage* is defined as Estage+2. It is possible to assign a def stage later than 3 to DATA, which may be necessary if a computation is performed on MEM1_In before being assigned to DATA. However, assigning a def stage earlier than 3 to DATA will result in the TIE compiler generating an error.

This lookup uses the optional *rdy* interface. Thus if the external device is not ready when the MEM1_GET instruction is executed (as indicated by the Rdy signal being deasserted), the instruction will stall until the external device is ready and the Rdy signal is asserted.

14.2.5 Example: Using Chained lookups

In the following example, two lookups are defined. The lookup MEM1 is a lookup whose address is written in the E stage and the data received in the M stage. Note that this lookup specification implies a 1-cycle latency when used in a 5-stage pipeline Xtensa processor, but a 2-cycle latency when used in a 7-stage pipeline Xtensa processor. CAM2 is a lookup whose address is written in the W stage and the data received three cycles later in stage (Wstage+3). The MYOP instruction uses both lookups, and the result of the first lookup MEM2 provides the output for the second lookup CAM2, thus performing a chained memory lookup. In this example, the second lookup could be a Content Addressable Memory (CAM) access that sends a 1 bit result indicating if the data was found.

```
regfile DATA 40 16 dat
lookup MEM2 {8, Estage+1} {32, Mstage} rdy //8 bit address, 32 bit data
lookup CAM2 {40, Wstage} {1, Wstage+3} //40 bit address, 1 bit data

operation MYOP {out DATA d, in AR addr}
          { out MEM2_Out, in MEM2_In, out CAM2_Out, in CAM2_In}
```

```
{
     assign MEM2_Out = addr[7:0];
     wire [39:0] mem2_data = { {8{MEM2_In[31]}}, MEM2_In };
     assign CAM2_Out = mem2_data;
     wire found = CAM2_In;
     assign d = (found) ? mem2_data : 40'h0;
}
schedule myop_sched { MYOP } { def d Wstage+3; }
```

In this example, lookup MEM2 has a Rdy input while lookup CAM2 does not. The second lookup CAM2 must always accept a request and return the result three cycles later. Also, note that there is a cycle between the stage that the instruction MYOP reads the data from MEM2 and the stage that it writes the address of the second lookup CAM2. This is to avoid a combinational path between a primary input and output of the Xtensa processor. The TIE compiler will flag an error if it detects such a combinational path. In this case, defining the lookup CAM2 with an output of M stage would be illegal because of the combinational path from MEM2_In to CAM2_Out in the instruction MYOP. For details of the restrictions on usage of lookups, see Section 14.6.

The output d of the instruction MYOP depends on the interface signal CAM2_In. CAM2_In is available in stage (Wstage+3) as defined in the definition of lookup CAM2. Thus the implicit def stage of d is also the same, and the schedule shown in the above example is redundant. A schedule construct would be necessary if you wanted to assign a def stage later than (Wstage+3) to the output operand d. Conversely, an earlier def stage (earlier than Wstage+3) would result in a TIE compiler error.

14.3 Timing of lookup Signals

The output of a lookup is generated based on the expression that drives the <*name*>_Out interface in the instruction semantic. This value is driven on the TIE_<*name*>_Out port without being registered inside the Xtensa processor. Similarly, the TIE_<*name*>_In input port is not registered before being available in the instruction semantic as the <*name*>_In interface. This is done to avoid adding two extra cycles to the latency of the lookup. You must ensure that the total amount of logic on the lookup output and input paths is such that your timing requirements can be met. On the output path, the delay of the expression generating the address, the routing delay to the external device, and the delay inside the external device before the address is registered must all add up to less than one clock cycle. On the input path, the delay of the external device to generate its data, the routing delay to the Xtensa processor, and the delay due to any further logic in the instruction semantic must all add up to less than one clock cycle.

With reference to the example in Section 14.2.4, the MEM1_Out lookup output is not registered in the processor after the semantic writes it. Similarly, the MEM1_In lookup input is not registered in the processor before it is used in the semantic. This design is

unlikely to have timing problems because there is no computation involved in generating the address, or after reading the data. Cadence recommends that you minimize the amount of logic on these interfaces, and pay careful attention to timing early in the design cycle.

The default timing constraints on lookup ports are set conservatively in the synthesis scripts provided by Cadence. They require the lookup output to be available early in the clock cycle, and assume that the lookup input is available late in the clock cycle. This is to ensure that timing violations in the system are easily detected. Refer to the *Xtensa LX Microprocessor Data Book* for details, including AC timings and timing diagrams of data transfer over these interface signals.

14.4 Stall Behavior of lookup

The latency of the external device that is connected to a lookup is expressed in the definition of the lookup as the number of cycles between the def and the use stage; that is, latency (N) = (*use-stage* - *def-stage*). When the instruction using a lookup is in the *def-stage*, the address or request is sent out, and the external device will generate a response after N cycles. However, the instruction using the lookup may not be in the *use-stage* at that time, due to other stalls in the processor. If that is the case, then the data needs to be held in a buffer until the instruction is ready to use it. This buffer is automatically generated by the TIE compiler. Because the processor can issue a request every cycle, the buffer is deep enough to hold N+1 data values, where N is the latency of the lookup.

14.4.1 Stall Behavior without using Rdy

The role of the buffer is illustrated by the timing diagram of Figure 14–8, using the example from Section 14.2.4. The lookup address is written by instruction `MEM1_GET` in the E stage and the lookup data is read in the W stage.

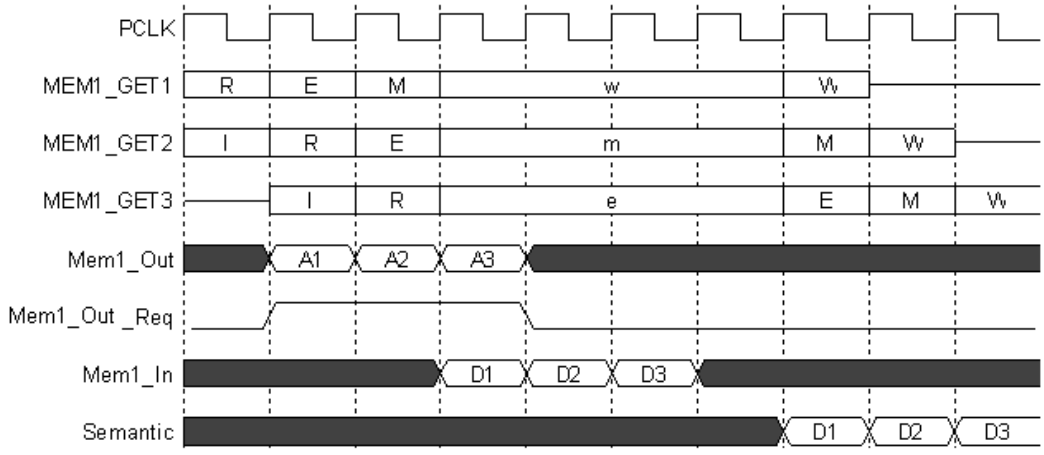

Figure 14–8. Lookup Buffer

As shown in Figure 14–8, there are three consecutive `MEM1_GET` instructions being executed. `MEM1_GET1` writes address `A1`, `MEM1_GET2` writes address `A2`, and `MEM1_GET3` writes the address `A3`. The `TIE_MEM1_Out_Req` signal is high for three cycles, indicating three lookup requests being initiated. Because the lookup is specified to have a latency of 2, the data input `D1` corresponding to `A1` is available two cycles after `A1` is sent out. The same is true for the data inputs `D2` and `D3` corresponding to `A2` and `A3`. However, due to the pipeline stall, none of the instructions are in the W stage when their respective data arrives from the external device. Therefore, a buffer must hold the values `D1`, `D2`, and `D3` until the instructions `MEM1_GET1`, `MEM2_GET2`, and `MEM3_GET3` respectively are in the W stage. Note that the buffer needs to hold three values because a lookup instruction, when stalled in its *def-stage*, will still write its output address in that cycle, as shown by the third instruction `MEM1_GET3` in this example. This is designed to prevent the external device from being accessed unnecessarily during a processor stall unrelated to the lookup operation.

The buffer is implemented as a speculative first-in-first-out buffer. Data from the lookup input is pushed into it every N (latency) cycles after a valid request goes out on the lookup output. Data is popped from it when a previously stalled instruction uses it. The push can be speculative if the instruction has not committed and therefore is canceled if the instruction is killed before the data is consumed. Note that this buffer is bypassed when the instruction has not stalled and is ready to consume the data as soon as it arrives from the external device.

14.4.2 Stall Behavior using Rdy

This example illustrates the behavior of the lookup interface when the external device is not ready. Suppose that in the example used in Section 14.2.4, the external memory is busy because another processor is doing DMA to fill a table. In this case, the Rdy input to the Xtensa processor is deasserted until the DMA is complete.

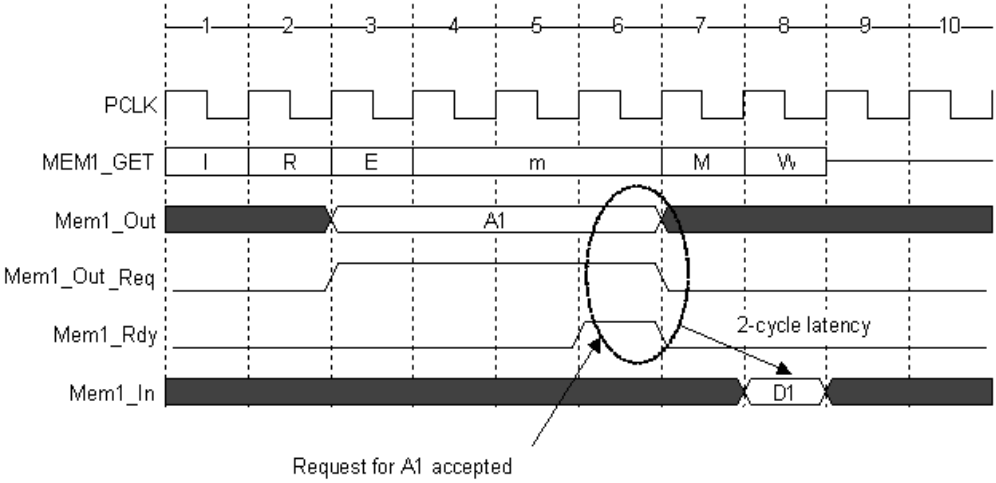

Figure 14–9. Lookup Ready

As shown in Figure 14–9, the MEM1_GET instruction writes the address A1 in its E stage and the Req output is asserted. However, Rdy is low in this cycle, indicating that the request is not accepted. This causes the instruction to stall in the next cycle. The Req output is kept asserted and the address A1 is repeated until Rdy becomes high. The Req output is de-asserted in the following cycle and the input data is sampled two cycles after the request is accepted. Note that a stalled request can be deasserted before the Rdy becomes high if the processor takes an exception, thus killing the lookup instruction.

14.5 Synchronization and Ordering

Instructions that read or write TIE lookups or queues, read from an import_wire, or write to an exported state all have effects that are visible external to the Xtensa LX processor. It is possible that the order in which these instructions are executed in the processor is different from the order of events visible outside the processor. In many situations, this ordering behavior is not a problem. For example, the system may not care about the relative order of instructions that access a lookup and an unrelated queue. Conversely, you may have a system in which this order is important.

Consider the following instruction sequence using the example in Section 14.2.5. There is a series of memory lookups using the MYOP instruction followed by a store to a memory mapped device, wherein the store signals to the external logic that all the previous lookup operations are complete.

```
MYOP
MYOP
MYOP
S32I to <mem mapped device>
```

In the preceding instruction sequence, the store instruction will write to the external memory mapped device when it reaches the commit stage W. At this time, the previous MYOP instruction would not have completed the read from CAM2 because of its latency. A lookup transaction is considered complete after the request is sent and the data is received. Thus in this code sequence, the external logic is signaled that the lookup transactions from the Xtensa processor are completed, although the last lookup from CAM2 is not yet completed.

To ensure that the system works correctly, the store must be executed after all lookup transactions are complete; that is, all requests accepted and responses received. For these situations, the Xtensa ISA defines a special synchronization instruction called External Wait or EXTW. The EXTW instruction ensures that all instructions ahead of it have gone past the commit stage (the W stage), and that all externally visible transactions of these instructions are complete, before any subsequent instruction executes.

In the example above, if EXTW is inserted in the program between the last MYOP instruction and the store, all the lookup transactions will complete before the store happens.

You must use the EXTW instruction at synchronization points in your assembly program where the ordering of external events is important. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for further details on the EXTW instruction.

You must also pay special attention to ordering and synchronization issues when writing and compiling C/C++ code for the Xtensa processor. The Xtensa C Compiler (XCC) ensures that instructions that use the same lookup execute in the order that they appear in the program. However, a lookup can be reordered with respect to other lookups, reads or writes to queues, reads from import_wires, state exports, and memory references. If this is not desirable, then the user program must provide pragmas for the compiler to maintain strict program order for all memory and external I/O accesses across the pragma points. Alternately, a command line switch may be used to enforce this ordering for the whole program. The XCC compiler offers multiple options for controlling the ordering of instructions and their externally visible effects; refer to the *Xtensa C and C++ Compiler User's Guide* for details.

14.6 Implementation Restrictions

- The maximum width of a `lookup` output or input interface is 1024 bits. The total number of TIE ports generated by designer-defined instructions is also limited to 1024. This includes ports generated from `import_wire`, `queue`, `lookup`, and `state` export.
- Lookup *def-stage* is configurable but restricted to E, M, and W stages for a 5-stage pipeline and L, M, and W stages for a 7-stage pipeline.
- For a 5-stage pipeline, the Rdy interface can only be used with lookups when their *def-stage* is the E stage. For a 7-stage pipeline, the Rdy interface can only be used with lookups when their *def-stage* is the L stage.
- Lookup latency, derived from (*use-stage* - *def-stage*) must be less than or equal to ten cycles.
- In a FLIX design, you cannot create a bundle with two or more operations that access the same lookup.

Restrictions are imposed on how lookups can interact with other TIE ports and memory interfaces. These restrictions are necessary to ensure the correct functionality of lookups in a pipelined processor implementation, and to match the use/def schedules of different interfaces. The following restrictions and guidelines apply to the lookup address output interface:

- The MemDataIn interface of a load instruction can be assigned to the lookup address output only when the def stage of the lookup is the W stage.
- The data input interface of one lookup can be assigned to the lookup address output of another lookup only when the second lookup def stage is W and the first lookup use stage is L or M stage.
- The data read from an input queue can be assigned to the lookup address output only when the def stage of the lookup is the W stage.
- The data read from an import_wire cannot be assigned to the lookup address output.

The following restrictions and guidelines apply to the lookup data input interface:

- The input data from a lookup can be assigned to an output queue only when the use stage of the lookup is the M stage.
- The input data from a lookup cannot be assigned to the MemDataOut interface of a store instruction.

All register files and TIE state (including exported state) are always a valid source and destination for use with lookups.

15. C Datatype (ctype) Sections

Programming for designer-defined register files can be very tedious without compiler support for register allocation. TIE provides a way to define one or more new C datatypes associated with a register file so that the Xtensa C/C++ compiler automatically does the register allocation and loads and stores data to and from the register file. A ctype can be a datatype that resides in a register file, or a composite or struct ctype that resides in multiple entries of a register file.

15.1 C Datatype Syntax

```
ctype-def ::= ctype name size alignment regfile-name [default | ctype-
list]
name ::= a unique C datatype name
size ::= size of memory needed to hold the type
alignment ::= alignment requirement for the type
regfile-name ::= name of the associated register file
ctype-list ::= { ctype argument-name [, ctype argument-name]+ }
ctype ::= a previously defined ctype
argument-name ::= an identifier unique to other argument names
                  in this ctype
```

name is a unique identifier that can be used in the C program as a type name. *size* describes two things. First, it is the number of useful bits contained in the data of this type. Second, in combination with alignment, it describes the number of bits used by this ctype when stored in memory: the size in memory is the ctype size rounded up to its alignment. The alignment must be a power-of-two number of bits from 8 up to a maximum of 512 bits. *default* is an optional attribute associated with a ctype. *ctype-list* is an optional list of component ctypes used when this ctype is a structure ctype.

Every register file must have at least one ctype. If a ctype is not defined for a register file, the TIE compiler automatically generates a ctype with the same name and width as the register file and alignment of a power-of-two bits greater than or equal to the width of the register file (or 512 bits if the register file is wider than that). If the C datatype does not contain the optional *ctype-list*, it must reside within one entry of the corresponding register file.

A datatype containing the optional `ctype-list` is called a struct C datatype. A struct C datatype is a software construct used to describe variables that span more than one register file entry. You can view a struct C datatype similarly to the C struct construct, where each constituent ctype in the list corresponds to a field of the struct. Each constituent ctype must be a base ctype. All multiple register file entries in the ctype-list must be from

the same register file. The C datatype for a register group must be a struct C datatype. The use of struct ctype with prototypes is described in Chapter 16, “Prototype (proto) Sections” on page 127.

15.2 Predefined Ctypes

Several ctypes are predefined for the AR register file. Unsigned values are uint8, uint16, and uint32. Signed values are int8, int16, and int32. The ctypes int64 and uint64 are also predefined as struct ctypes[1]:

```
ctype int64 64 64 AR { int32 hi, uint32 lo }
ctype uint64 64 64 AR { uint32 hi, uint32 lo }
```

These predefined ctypes are only used within the TIE language; use the corresponding standard C datatypes when accessing these types from your C programs.

Ctype xtbool is predefined for the core Boolean register file BR. Several struct ctypes are predefined for the register groups of the Boolean register file:

```
ctype xtbool2 2 8 BR2 {xtbool b1, xtbool b0} default
ctype xtbool4 4 8 BR4 {xtbool b3, xtbool b2, xtbool b1, xtbool b0}
default
ctype xtbool8 8 8 BR8 {xtbool b7, xtbool b6, xtbool b5, xtbool b4,
xtbool b3, xtbool b2, xtbool b1, xtbool b0} default
ctype xtbool16 16 16 BR16 {xtbool b15, xtbool b14, xtbool b13, xtbool
b12, xtbool b11, xtbool b10, xtbool b9, xtbool b8, b7, xtbool b6,
xtbool b5, xtbool b4, xtbool b3, xtbool b2, xtbool b1, xtbool b0}
default
```

The AR and BR register file ctypes and the corresponding C datatypes are listed in Table 15–11.

Table 15–11. Predefined ctypes for Xtensa Core Register Files

Ctype Name	Register File	Corresponding C datatype
int8	AR	signed char
int16	AR	short
int32	AR	int
int64	AR	long long
uint8	AR	char
uint16	AR	unsigned short

1. Before release RD2010.0, only pointers to int64 and uint64 data types can be defined. The usage of struct ctypes is described in Section 16.8 on page 143.

Table 15–11. Predefined ctypes for Xtensa Core Register Files

Ctype Name	Register File	Corresponding C datatype
unit32	AR	unsigned int
uint64	AR	unsigned long long
xtbool	BR	xtbool
xtbool2	BR2	xtbool2
xtbool4	BR4	xtbool4
xtbool8	BR8	xtbool8
xtbool16	BR16	xtbool16

15.3 Example 1

```
regfile VR 128 16 v
ctype vector16 128 128 VR
```

This example defines a 128-bit register file and a new `vector16` C datatype that can be used in your C program. With this declaration, it is possible to write C program statements such as:

```
vector16 a, b, c;        // declaring three variables of type vector16
...
c = vector16_add(a, b);  // perform vector16 add
```

Chapter 7, “Instruction Operation (`operation`) Sections” on page 37 describes how to define the semantics of the new instruction `vector16_add`. In this example it illustrates the use of the C datatype. Without the C datatype, the equivalent operation would have to be described in assembly. Using assembly necessitates loading the a, b, and c addresses into base registers, loading the input values into fixed registers in the register file VR, performing the computation, and storing the data from a fixed register location into memory.

For the compiler to automatically load and store the `vector16` data to and from the VR register file, it needs to know the instruction sequences for loading and storing the data. This information is in Chapter 16, “Prototype (`proto`) Sections” on page 127.

15.4 Example 2

```
regfile XR 40 16 x
ctype XRtype 40 32 XR
```

In this example, while the ctype is 40 bits wide, the Xtensa C/C++ compiler allocates 64 bits for a XRtype variable in memory. This happens because the memory size is rounded to the smallest multiple of the alignment that is greater than or equal to the size of the ctype. Thus 64-bits in memory will be consumed by a variable of type XRtype, although only 40 bits of it will contain useful information.

The alignment of XRtype is declared to be 32. This means that variables of type XRtype, when stored in memory, will be stored at memory addresses that are a multiple of 32 bits or 4 bytes. While this is legal, it may be advantageous in this case to specify an alignment of 64, especially if your Xtensa processor supports a memory access width of 64 bits or higher. A 40-bit variable stored with 32-bit alignment in memory will require two load instructions to read the data from memory.

15.5 Example 3

```
regfile MYREG 32 8 my
ctype myreg32 32 32 MYREG default
ctype myreg16 16 32 MYREG
```

In this example, there is a 32-bit register file with two ctypes. This is useful when you want two different types of variables to reside in the MYREG register file. Variables of type myreg32 will be 32-bits wide while variables of type myreg16 will be 16-bits wide. The ctype myreg32 is annotated to be the default ctype corresponding to this register file. The default ctype must span the entire width of the register file.

The default ctype attribute is useful in automatically generating instruction prototypes as explained in Section 16.5 “Giving an Instruction a Unique Prototype” on page 140. When you have a register file with two or more ctypes, you should annotate one of them as the default. If you do not do so, the TIE compiler will select a ctype to be the default ctype for that register file. The debugger will use the loadi and storei prototypes associated with the default ctype to display the value of a register from the register file. Similarly, the operating system running on the Xtensa processor (if any) will use the loadi and storei prototypes associated with the default ctype to save and restore the entries of a register file. These prototypes are described in Section 16.2 “Specifying Instructions for Register Allocation, Context Switching, and Debugging” on page 128.

15.6 Example 4

```
regfile XR 32 8 xr
ctype reg32 32 32 XR default
ctype reg64 64 64 XR { reg32 a, reg32 b}
```

In this example, there is a 32-bit register file with two ctypes. ctype reg32 is the default ctype. ctype reg64 is a struct ctype. It is composed of two reg32 ctypes. ctype reg64 is 64 bits wide and is aligned to 64-bit boundaries. By defining a struct ctype, it is possible to access a data type that is wider than the register file width. You need to manually write the prototypes of struct ctypes. For details, see Chapter 16, “Prototype (proto) Sections” on page 127.

15.7 Defining ctypes on Coprocessor Register Files

If you include the Floating Point, ConnX Vectra LX, HiFi2, ConnX D2, or BBE16 coprocessor(s) in your Xtensa configuration, you can use the register files defined by these coprocessors in your TIE description. In particular, you can define additional operations, protos, or ctypes on these register files. This is illustrated in the example below:

```
ctype myfp 16 32 FR

operation FOO {out FR a, in FR b, in FR c} {} {
     ... // operation description
}
proto FOO {out xtfloat a, in myfp b, in myfp c} {} {
   FOO a, b, c;
}

proto myfp_loadi {out myfp t, in myfp *p, in immediate offset} {} {
     ... // instruction sequence for loadi proto
}
proto myfp_storei {in myfp t, in myfp *p, in immediate offset} {} {
     ... // instruction sequence for storei proto
}
proto myfp_move {out myfp t, in myfp r} {} {
     ... // instruction sequence for move proto
}
```

In this example, we create a new ctype called myfp which represents 16-bit wide variables that reside in the FR register file of the Floating Point coprocessor. The instruction FOO has two input operands from the FR register file which are of type myfp, the newly created ctype. The output of the instruction FOO is of type xtfloat, which is the predefined ctype for the FR register file. xtfloat is also the default ctype for the FR register file, which is why an instruction prototype is necessary for the instruction FOO, to indicate that operands b and c are of type myfp. In the absence of this proto, all operands of the instruction FOO will be assumed to be of type xtfloat. Thus any instruction intended to operate on variables of type myfp will need an instruction prototype. Instruction prototypes are explained in detail in Section 16.5 “Giving an Instruction a Unique Prototype” on page 140.

Every ctype of a register file requires the _loadi, _storei and _move protos as explained in Chapter 16. Thus you need to define myfp_loadi, myfp_storei and myfp_move protos for the newly created ctype. This may in turn require you to create new instructions to be used with these protos. If you do not create these protos, the TIE compiler will automatically create them for you, along with instructions that may be necessary for these protos.

15.8 Implementation Restrictions

Following are the implementation restrictions for the ctype sections:

- The *alignment* parameter must be a power of 2, be less than or equal to 512, and be greater than or equal to 8.
- It is not possible to create new ctypes on the BR register file or the MR register file of the Xtensa core.

16. Prototype (proto) Sections

Prototype sections show the C/C++ compiler, debugger, and the RTOS how to use designer-defined register files, or more specifically, the ctypes associated with designer-defined register files. They are also used for describing instruction aliases, idioms, and type conversions. There are several uses of prototypes, including:

- Specifying the load instruction sequence to load data of a designer-defined ctype from memory into a register file
- Specifying the store instruction sequence to store data of a designer-defined ctype from a register file to memory
- Specifying the move instruction sequence to copy data of a designer-defined ctype from one register to another register
- Describing instruction aliases
- Describing instruction idioms
- Specifying the instruction sequence to convert from one datatype to another

The TIE compiler is capable of automatically generating the load, store, and move prototypes for most designer-defined register files. There are situations (as described in Section 16.2.2 “Example 2” on page 130) where this is not possible. In these cases, provide this information through `proto` description sections.

16.1 Prototype Syntax

```
proto-def ::= proto name {proto-arg-list}{[temp-arg-list]}{code-list}
name ::= a unique proto name
proto-arg-list ::= proto-arg [, proto-arg]*
proto-arg ::= dir ctype [*] arg-name
temp-arg-list ::= temp-arg [, temp-arg]*
temp-arg ::= ctype arg-name
dir ::= in | out | inout
ctype ::= immediate | a previously defined ctype
code-list ::= code; [code;]*
code ::= instruction arg-expr [, arg-expr]*
instruction ::= a previously defined instruction name
arg-expr ::= arg-name | arg-name + constant | arg-name - constant |
             constant | struct-ctype-arg-name
arg-name ::= a unique identifier
struct-ctype-arg-name ::= arg-name->field-name
field-name ::= an identifier specified in struct ctype
```

name is a unique identifier for this prototype. *proto-arg-list* specifies the list of arguments taken by the prototype. Each argument declaration consists of the following:

- A direction, *dir*, which indicates whether the argument is an input, an output, or both
- *ctype*, which indicates the type of the argument (use the keyword `immediate` for inputs to indicate that the value is encoded in an immediate field in the instruction and must be provided as a literal constant)
- An optional *, which indicates that the argument represents a pointer to the specified ctype
- An identifier for the argument

temp-arg-list is an optional list of temporary variables in which the prototype can store intermediate results. *code-list* specifies a list of instructions used to implement the prototype. The TIE compiler checks the *code-list* with the prototype arguments and detects errors, such as a mismatch in the direction of the arguments or missing arguments. Instructions used in the *code-list* of a proto declaration must use the same case (upper case or lower case) as used in the instruction definition. In particular, use upper case for all Xtensa ISA instructions used in proto declarations.

When a *proto-arg* specifies a * before the *arg-name*, the argument is interpreted as a pointer to the specified *ctype*. A pointer argument is typically used to indicate that the argument is being used as the address of a *ctype* value in memory for a load or store operation (as in Section 16.2.1 “Example 1” on page 129). A pointer argument must be an input argument, and the corresponding `iclass` operand or `operation` argument must represent a value from the AR register file.

16.2 Specifying Instructions for Register Allocation, Context Switching, and Debugging

For the compiler to automatically load and store a datatype and perform register allocation, it needs to know how to load data from memory to a register file, store data from a register file to memory, and move data from one register to another. This information is specified using prototypes with certain stylized names. Specifically, for each declared <*ctype*>, the compiler needs to know the load, store, and move instructions which are described by prototypes named <*ctype*>`_loadi`, <*ctype*>`_storei`, and <*ctype*>`_move`. The `loadi` and `storei` prototypes also make designer-defined register files accessible via a debugger when debugging software on an actual hardware platform.

In most situations, the TIE compiler can automatically generate the load, store and move instructions for a designer-defined register file, and then generate the stylized prototypes for the associated ctypes. This automatic generation is not possible under the following conditions:

- If the width of the register file is greater than the data memory access width of the Xtensa processor
- If you define a ctype whose width is less than the width of the associated register file
- If you define a ctype whose alignment is less than the width of the associated register file
- If the ctype is on a register group.

For any of the above situations, you must provide your own `loadi`, `storei`, and `move` prototypes. The compiler requires the instructions implementing the stylized prototypes to meet some requirements that are illustrated in the following example.

16.2.1 Example 1

Consider a designer-defined register file named VR, which is 64 bits wide. A `vec64` ctype is also defined for this register file. Instruction `L64I` loads 64 bits from memory into this register file, while instruction `S64I` stores the contents of this register file to memory. Both of these instructions generate the memory address by adding an immediate value (`imm`) to a base address pointer (`addr`). Instruction `M64` moves the contents of one VR register into another.

```
regfile VR 64 16 v
ctype vec64 64 64 VR

immediate_range IMOFSET 0 248 8
immediate_range SEL 0 1 1

operation L64I {out VR vreg, in VR *addr, in IMOFSET imm}
               {out VAddr, in MemDataIn64} {
     assign VAddr = addr + imm;
     assign vreg = MemDataIn64;
}
operation S64I {in VR vreg, in VR *addr, in IMOFSET imm}
               {out VAddr, out MemDataOut64} {
     assign VAddr = addr + imm;
     assign MemDataOut64 = vreg;
}
operation M64 {out VR vout, in VR vin} {} {
     assign vout = vin;
}

operation VR_AR {out AR aout, in VR vin, in SEL s} {} {
     assign aout = s ? vin[63:32] : vin[31:0];
}

operation AR_VR {inout VR vout, in AR ain, in SEL s} {} {
     assign vout = s ? {ain, vout[31:0]} : {vout[63:32], ain};
```

```
}

proto vec64_loadi {out vec64 v, in vec64 *p, in immediate o} {} {
     L64I v, p, o;
}
proto vec64_storei {in vec64 v, in vec64 *p, in immediate o} {} {
     S64I v, p, o;
}
proto vec64_move {out vec64 v, in vec64 vs} {} {
     M64  v, vs;
}
```

The compiler requires the instructions implementing the stylized prototypes to preserve all the bits of the corresponding ctype. For the case of the register file VR, assume register v1 contains a vec64 ctype value. The compiler stores register v1 using the instructions implementing the vec64_storei prototype, and then loads register v2 from the same memory address using the instructions implementing the vec64_loadi prototype. By using this instruction sequence, the compiler expects that all 64 bits of the vec64 ctype in v1 and v2 are equal.

Because the compiler can use the <*ctype*>_loadi, <*ctype*>_storei, and <*ctype*>_move prototypes at arbitrary points in the program, the instructions implementing these prototypes cannot have any side-effects other than those expected by the compiler. That is, the <*ctype*>_loadi prototype must load a value from the address formed by adding pointer <p> and immediate <o>, and store that value into <v>. The instructions implementing the <*ctype*>_loadi prototype must not alter any other processor state (except for the temporary registers specified in *temp-arg*). Similarly, the <*ctype*>_storei prototype must store <v> to the address formed by adding pointer <p> and immediate <o> without altering any other processor state, and the <*ctype*>_move prototype must move <vs> to <v> without altering any other processor state.

The above example represents a case where the TIE compiler could automatically generate the required prototypes based on the register file declaration alone, assuming that the processor is configured with a data memory access width of 64 or 128 bits.

16.2.2 Example 2

This example illustrates a situation where the TIE compiler cannot automatically generate the required prototypes. Consider a VR register file that contains sixteen 96-bit registers configured with a data memory access width of 64 bits. Two ctypes, vec64 and vec96, are associated with the VR register file. vec64 has size 64 bits and occupies the low-order 64 bits of a VR register. vec96 has size 96 bits and occupies all 96 bits of a VR register.

```
regfile VR 96 16 v
ctype vec64 64 64 VR
ctype vec96 96 64 VR
```

In this situation, specify the prototypes for the vec64 ctype because it is narrower than the width of the register file. Also specify the prototypes for the vec96 ctype, as there is no obvious way to load a 96-bit value on a 64-bit memory interface. Defining the following instructions and prototypes may be one way to meet this requirement:

```
immediate_range IM64 0 120 8
immediate_range IM32 0 60 4

operation L64  {out VR vreg, in VR *addr, in IM64 im8}
               {out VAddr, in MemDataIn64} {
     assign VAddr = addr + im8;
     assign vreg = {32'b0, MemDataIn64};
}
operation S64  {in VR vreg, in VR *addr, in IM64 im8}
               {out VAddr, out MemDataOut64} {
     assign VAddr = addr + im8;
     assign MemDataOut64 = vreg[63:0];
}
operation L32H {inout VR vreg, in VR *addr, in IM32 im4}
               {out VAddr, in MemDataIn32} {
     assign VAddr = addr + im4;
     assign vreg = {MemDataIn32, vreg[63:0]};
}
operation S32H {in VR vreg, in VR *addr, in IM32 im4}
               {out VAddr, out MemDataOut32} {
     assign VAddr = addr + im4;
     assign MemDataOut32 = vreg[95:64];
}
operation MVR {out VR vout, in VR vin} {} {
     assign vout = vin;
}

proto vec64_loadi {out vec64 v, in vec64 *p, in immediate o} {} {
     L64 v, p, o;
}
proto vec64_storei {in vec64 v, in vec64 *p, in immediate o} {} {
     S64 v, p, o;
}
proto vec64_move {out vec64 vout, in vec64 vin} {} {
     MVR vout, vin;
}

proto vec96_loadi {out vec96 v, in vec96 *p, in immediate o} {} {
     L64 v, p, o;
     L32H v, p, o + 8;
```

```
}
proto vec96_storei {in vec96 v, in vec96 *p, in immediate o} {} {
     S64 v, p, o;
     S32H v, p, o + 8;
}
proto vec96_move {out vec96 vout, in vec96 vin} {} {
     MVR vout, vin;
}
```

The L64 (S64) instruction loads (stores) the 64 low-order bits of a VR register. The L32H (S32H) instruction loads (stores) the 32 high-order bits of a VR register. The prototypes for the vec64 ctype are implemented with the L64 and S64 instructions. For this ctype, the upper 32 bits of the register file do not matter and are not used. The vec96_loadi prototype is implemented with two load instructions: L64 to load the low-order 64 bits and L32H to load the high-order 32 bits. Notice that the immediate offset for the L32H increments by eight, so that the L32H loads from the memory address containing the high-order 32 bits of the 96-bit value. Similarly, the vec96_storei prototype is implemented with the S64 and S32H instructions.

16.2.3 Example 3

This example illustrates a case where the TIE compiler is capable of automatically generating the prototypes, but you want to provide a different way to implement the prototype. The register file XR is 32 bits wide, and is associated with the ctype x32.

```
regfile XR 32 16 x
ctype x32 32 32 XR
```

If the above declarations are all that is specified in the TIE file, the TIE compiler creates a load, stores and moves instructions for the XR register file, and uses them to create the appropriate prototypes. However, assume that you do not want to create new load/store instructions in the design, as there already exist 32-bit load/store instructions for the predefined Xtensa processor address register file (AR regfile). Instead, choose to define instructions to move data between the AR regfile and the XR regfile, and use these instructions for the prototypes.

```
operation WX32 {out XR x, in AR a} {} {
     assign x = a;
}
operation RX32 {out AR a, in XR x} {} {
     assign a = x;
}

proto x32_loadi {out x32 x, in x32 *p, in immediate o} {int32 t} {
     L32I t, p, o;
```

```
     WX32 x, t;
}
proto x32_storei {in x32 x, in x32 *p, in immediate o} {int32 t} {
     RX32 t, x;
     S32I t, p, o;
}

proto x32_move {out x32 xout, in x32 xin} {int32 t} {
     RX32 t, xin;
     WX32 xout, t;
}
```

Note the use of the temporary register `t` in the above prototype declarations. The load prototype uses the Xtensa ISA instruction `L32I` to load the memory data into a temporary AR register referred to by the argument `t`. This value is then moved to the XR register using the `WX32` instruction. Similarly, the store prototype first moves the data from the XR register to a temporary AR register and then stores it to memory using the Xtensa ISA store instruction `S32I`. The move prototype moves data from one XR register to another via an intermediate AR register.

This implementation takes two instructions to load/store data to and from the XR register file. This is less efficient than creating a new set of load/store instructions for the XR register file, in which case either operation can be done by one instruction. However, note that the move instruction `RX32` and `WX32` are likely to take less opcode space than the load/store instructions, and are virtually free in terms of hardware cost.

16.3 Additional Load/Store Prototypes

The previous section discussed the <*ctype*>`_loadi`, <*ctype*>`_storei`, and <*ctype*>`_move` prototypes that are necessary for the compiler to automatically load and store datatypes, and perform register allocation. In addition to these, it is possible to optionally provide prototypes that help the compiler generate more compact and efficient code. These prototypes are not generated automatically by the TIE compiler.

Consider the TIE description in Section 16.2.1 “Example 1” on page 129. This example illustrates how to use the instructions `L64I` and `S64I` to define the basic <*ctype*>`_loadi` and <*ctype*>`_storei` prototypes. Consider two new instructions:

```
operation L64IU {out VR vreg, inout VR *addr, in IMOFSET imm}
               {out VAddr, in MemDataIn64} {
     wire [31:0] address = addr + imm;
     assign VAddr = address;
     assign vreg = MemDataIn64;
     assign addr = address;
}
operation S64IU {in VR vreg, inout VR *addr, in IMOFSET imm}
               {out VAddr, out MemDataOut64} {
```

```
        wire [31:0] address = addr + imm;
        assign VAddr = address;
        assign MemDataOut64 = vreg;
        assign addr = address;
    }
```

The instructions L64IU and S64IU update the address pointer (addr) with the incremented address. They are similar to the L64I and S64I instructions in all other respects. A <*ctype*>_loadiu and <*ctype*>_storeiu prototype may be defined with these instructions:

```
proto vec64_loadiu {out vec64 v, inout vec64 *p, in immediate o} {} {
     L64IU     v, p, o;
}
proto vec64_storeiu {in vec64 v, inout vec64 *p, in immediate o} {} {
     S64IU     v, p, o;
}
```

A <*ctype*>_loadiu prototype must load a value from the address formed by adding the pointer <*p*> and immediate <*o*>, store that value into <*v*>, and increment the pointer <*p*> by the immediate <*o*>. Similarly, the <*ctype*>_storeiu prototype must store <*v*> to the address formed by adding the pointer <*p*> and immediate <*o*>, and must increment the pointer <*p*> by the immediate <*o*>. No other processor state (except for the temporary registers specified in temp-arg) can be modified by the <*ctype*>_loadiu or <*ctype*>_storeiu prototypes.

The compiler uses the <*ctype*>_loadiu prototypes to generate more compact code when a <*ctype*>_loadi prototype follows an increment of address pointer <*p*> by an amount equal to <*o*>. For example, consider compiling the following C subroutine:

```
#include <xtensa/tie/v.h>

void vcopy(vec64 a[], vec64 b[], int n) {

   int i;

   for (i=0; i<n; i++) {
     a[i] = b[i];
   }

}
```

Following is the code generated by the compiler without vec64_loadiu and vec64_storeiu prototypes:

```
            loopnez a4, .L2
.L6:
            l64i    v0, a3, 0
```

```
        addi.n  a3, a3, 8
        s64i    v0, a2, 0
        addi.n  a2, a2, 8
        # loop end for .L6
```

The code generated by the compiler with `vec64_loadiu` and `vec64_storeiu` prototypes follows:

```
        loopnez a4, .L2
.L6:
        l64iu   v0, a3, 8
        s64iu   v0, a2, 8
        # loop end for .L6
```

Two other sets of optional prototypes enable the compiler to generate more efficient code. The first one is for indexed loads and stores: <*ctype*>`_loadx`, and <*ctype*>`_storex`. These prototypes are used for indexed load/store instructions that compute the memory address by summing two register values. The specifications for these two prototypes are very similar to their immediate version counterpart, except that the immediate argument is replaced with an unsigned integer variable. For example,

```
proto vec64_loadx { out vec64 v, in vec64 *p, in uint32 x} {} {
   L64X v,p,x;
}

proto vec64_storex { in vec64 v, in vec64 *p, in uint32 x} {} {
   S64X v,p,x;
}
```

They are used when the address computation for the load or store involves a base and a varying offset, such as `a[k]` where `k` is an input argument.

The second set of optional prototypes is for indexed updating loads and stores: <*ctype*>`_loadxu`, and <*ctype*>`_storexu`. The specifications for these two prototypes are very similar to their immediate version counterpart, except that the immediate argument is replaced with an unsigned integer variable. For example,

```
proto vec64_loadxu { out vec64 v, inout vec64 *p, in uint32 x} {} {
   L64XU v,p,x;
}

proto vec64_storexu { in vec64 v, inout vec64 *p, in uint32 x} {} {
   S64XU v,p,x;
}
```

These prototypes are used when the address computation for the load or store involves a base and a varying offset, and the base is incremented (for example, in a loop).

The updating prototypes (<*ctype*>_loadiu, and <*ctype*>_storeiu) have their post-updating counterparts: <*ctype*>_loadip, and <*ctype*>_storeip. Similarly, the indexed updating prototypes (<*ctype*>_loadxu, and <*ctype*>_storexu), have post-updating versions as well: <*ctype*>_loadxp, and <*ctype*>_storexp. The post-updating instructions obtain the address to load or store the data from the address pointer directly, then the address pointer is updated. The XCC compiler can utilize the post-updating prototypes similar to the pre-updating prototypes. In a processor configuration, it is not needed to specify both pre-updating and post-updating instructions and prototypes. The following is an example of the post-updating instructions and prototypes:

```
operation L64IP {out VR vreg, inout VR *addr, in IMOFSET imm}
               {out VAddr, in MemDataIn64} {
     wire [31:0] address = addr + imm;
     assign VAddr = addr;
     assign vreg = MemDataIn64;
     assign addr = address;
}
operation S64IP {in VR vreg, inout VR *addr, in IMOFSET imm}
               {out VAddr, out MemDataOut64} {
     wire [31:0] address = addr + imm;
     assign VAddr = addr;
     assign MemDataOut64 = vreg;
     assign addr = address;
}
proto vec64_loadip {out vec64 v, inout vec64 *p, in immediate o} {} {
     L64IP     v, p, o;
}
proto vec64_storeip {in vec64 v, inout vec64 *p, in immediate o} {} {
     S64IP     v, p, o;
}

operation L64XP {out VR vreg, inout VR *addr, in AR x}
               {out VAddr, in MemDataIn64} {
     wire [31:0] address = addr + x;
     assign VAddr = addr;
     assign vreg = MemDataIn64;
     assign addr = address;
}
operation S64XP {in VR vreg, inout VR *addr, in AR x}
               {out VAddr, out MemDataOut64} {
     wire [31:0] address = addr + x;
     assign VAddr = addr;
     assign MemDataOut64 = vreg;
     assign addr = address;
}
proto vec64_loadxp {out vec64 v, inout vec64 *p, in int32 x} {} {
     L64XP     v, p, x;
```

```
}
proto vec64_storexp {in vec64 v, inout vec64 *p, in int32 x} {} {
     S64XP     v, p, x;
}
```

16.4 Specifying Instructions for Automatic Datatype Conversion

When working with custom register files, you may want the C/C++ compiler to perform automatic type conversions for a variety of situations. For example, it is often useful to initialize variables of custom datatypes with integer values. This information is again specified using prototypes with certain stylized names. It is necessary for a conversion sequence to specify the location of the source and target data, as the data can reside in either memory or registers. Thus the prototype <stype>_<sloc>to<tloc>_<ttype> specifies the conversion sequence for converting data of <stype> to that of <ttype>. <sloc> and <tloc> specify the source and target locations and can be either "r" for register or "m" for memory. These protos are used automatically by the compiler whenever a variable of <stype> is assigned to a variable of <ttype> or whenever a variable of <stype> is passed to a TIE intrinsic that expects a variable of <ttype>. Transitivity is not supported. If there is no proto that directly converts from an <stype> to a <ttype>, it is not possible to assign a variable of <stype> to a variable of <ttype> even if there is a conversion proto from <stype> to <other_type> and from <other_type> to <ttype>.

Because literals are always of a standard C/C++ type, the Xtensa compiler must use a conversion prototype to initialize a ctype to a literal. It is necessary to declare the appropriate conversion prototype to convert the initial value to the declared ctype. One exception to this is static initialization in C, where the bit patterns of the initial values are stored and a warning is generated to alert the designer that no conversion will be performed by the compiler. The size of the value stored in this case cannot exceed the size of the declared ctype and cannot be more than 64 bits.

16.4.1 Example 1

```
regfile VR 24 16 v
ctype rgb 24 32 VR /* pixel in reg-blue-green representation */
ctype yuv 24 32 VR /* pixel in luminance-chrominance representation */
proto rgb_mtor_yuv {out yuv v, in rgb *p, in immediate o} {} {
     ...instructions for memory rgb data to register yuv data...
}
proto yuv_rtom_rgb {in yuv v, in rgb *p, in immediate o} {} {
     ...instructions for register yuv data to memory rgb data...
}
proto rgb_rtor_yuv {out yuv v, in rbg s} {} {
     ...instructions for register rgb data to register yuv data...
}
```

This partial description defines some rules for converting between two possible representations of pixel data. With these rules, the compiler is able to automatically convert between rgb and yuv data when necessary. For example, let pixel_average be an instruction that takes two yuv pixel values and produces a yuv pixel value. A piece of C code without compiler-assisted type conversion might look like:

```
rgb a, b, c;
yuv x, y, z;
x = rgb2yuv(a);     /* explicit conversion */
y = rgb2yuv(b);     /* explicit conversion */
z = pixel_average(x, y);
c = yuv2rgb(z);     /* explicit conversion */
```

With compiler-assisted type conversion, it would look like:

```
rgb a, b, c;
c = pixel_average(a, b);
```

16.4.2 Example 2

```
proto int64_mtor_vec64 { out vec64 v, in int64 *p, in immediate o } {}
{
     L64I v, p, o;
}

proto vec64_rtom_int64 { in vec64 v, in int64 *p, in immediate o } {}
{
     S64I v, p, o;
}
```

The protos in this example do the conversion between 64-bit data type int64 and the TIE data type vec64 by performing a load or store from the TIE register file VR (see Section 16.2.1 “Example 1” on page 129) to memory. Note that the TIE-predefined type int64 can be treated as the type long long.

Beginning with the RD2010.0 release, int64 datatype is treated as a struct ctype composing two AR registers.

```
ctype int64 64 64 AR { int32 hi, uint32 lo }
```

Thus, by implementing the conversion protos below, data can be moved between AR and VR registers in automatic type conversions without going through memory.

```
proto int64_rtor_vec64 {inout vec64 v, in int64 a} {} {
     AR_VR v, a->lo, 0;
     AR_VR v, a->hi, 1;
}
proto vec64_rtor_int64 {out int64 a, in vec64 v} {} {
```

```
     VR_AR a->lo, v, 0;
     VR_AR a->hi, v, 1;
}
```

Following is a sample program that tests the conversion protos, with the definition of the instruction v64AND used in this example.

```
operation v64AND {out VR vout, in VR vin1, in VR vin2} {} {
    assign vout = vin1 & vin2;
}

int main(int argc, char **argv) {
        int err = 0;
        long long x, y;
        vec64 z;

        x = 0xffff << 20;
        y = 0xf << 20;

        z = v64AND(x, y);

        err = ( ((long long)z) != ( x&y) ) ;

        if (  err )
           printf("Test failed\n");
     return(err);
}
```

The compiler uses the conversion proto int64_mtor_vec64 or int64_rtor_vec64 for converting x and y as arguments to the TIE instruction v64AND. The resulting z is converted to type int64 using the conversion proto vec64_rtom_int64 or vec64_rtor_int64 for doing the comparison in (z != (x&y)). The choice of the conversion proto to use depends on the performance analysis of the compiler. Without the right protos, there might be compile errors such as:

```
casts from type "_TIE_test_vec64" to type "signed long long" not allowed
```

Table 16–12 shows the naming convention of the load, store, move, and conversion prototypes associated with a ctype. For a list of all the predefined ctypes corresponding to the predefined Xtensa processor register files, refer to Table 15–11 on page 122.

Table 16–12. Prototype Suffixes and Stylized Names for a Ctype

Name	Description
loadi	Loads data of <ctype> from memory location (address pointer + immediate offset)
loadiu	Loads data of <ctype> from memory location (address pointer + immediate offset) and updates the address pointer
loadx	Loads data of <ctype> from memory location (address pointer + index register)
loadxu	Loads data of <ctype> from memory location (address pointer + index register) and updates the address pointer
storei	Stores data of <ctype> to memory location (address pointer + immediate offset)
storeiu	Stores data of <ctype> to memory location (address pointer + immediate offset) and updates the address pointer
storex	Stores data of <ctype> to memory location (address pointer + index register)
storexu	Stores data of <ctype> to memory location (address pointer + index register) and updates the address pointer
move	Moves data between two variables of the <ctype>
<from>_rtom_<to>	Convert sdata of ctype <from> in register to ctype <to> in memory
<from>_mtor_<to>	Convert sdata of ctype <from> in memory to ctype <to> in register
<from>_rtor_<to>	Converts data of ctype <from> in register to ctype <to> in register

16.5 Giving an Instruction a Unique Prototype

It is possible for several ctypes to be associated with a single register file. For example, the AR register file can contain int's, short's, and char's. Similarly, designer-defined register files can be declared to have multiple ctypes associated with them. When multiple ctypes share a register file, it is likely that some instructions that operate on the register file are only meaningful for one of the ctypes. For example, it may not be meaningful to do a signed multiplication on an unsigned value, even though the hardware does not care if the value in a register is signed or unsigned. To prevent software developers from using instructions on the wrong types, instruction prototypes are used to tell the compiler what ctype is expected for each register file operand of every instruction.

Instruction prototypes are required for every instruction that operates upon a register file with more than one ctype. This prototype will specify the ctype of the variable that is expected to be used with that instruction. The following example illustrates an instruction prototype.

```
regfile SR 16 8 s        // a 16-bit register file
ctype SS 16 16 SR        // one ctype (signed short)
ctype US 16 16 SR        // another ctype (unsigned short)

operation SS_GT {out SR sout, in SR sin1, in SR sin2} {} {
```

```
     ... implementation of SS_GT instruction ...
}

proto SS_GT {out SS r, in SS s, in SS t} {} {
     SS_GT r, s, t;
}
```

In this example, there are two ctypes associated with the register file SR. The SS_GT instruction returns the greater of the two input operands as the result, where the operands are required to be of ctype SS. A prototype specifies that the SS_GT C/C++ intrinsic expects to operate on variables of ctype SS. It is a compile time error to use a variable of ctype US with the SS_GT intrinsic in your C/C++ code.

If there is only one ctype for a certain register file, then an instruction prototype is not required for instructions that operate upon that register file. It is obvious that the one and only ctype associated with the register file is the type of the variable used with the instructions.

16.5.1 Default ctypes and Automatically Generated Prototypes

The need for instruction prototypes in a design that has a register file with more than one ctype has been previously discussed. In designs with a large number of instructions, this requires a lot of prototype definitions. To help mitigate this requirement, the TIE language requires a *default* ctype to be associated with each register file. Every register file with multiple ctypes has one ctype designated as the default ctype for that register file. If an instruction prototype is not specified, the default ctype is assumed to be the type associated with the operands of that instruction. As a result of this behavior, you only need to write instruction prototypes for those instructions that do not use the default ctype.

The default ctype is used by the OS and debugger in context switch and debug. The load, store, and move protos of the default ctype must be able to save and restore the entire register.

The predefined `uint32` ctype is assumed as the default for the AR register file. If an instruction uses something other than `uint32` for an AR register operand, an instruction prototype must be explicitly defined.

16.6 Specifying Instruction Aliases

When one hardware instruction has meaning for multiple types, use prototypes to create aliased C/C++ intrinsics for the instruction.

16.6.1 Example

Consider again the SR register file in Section 16.5. This register file holds a signed datatype and an unsigned datatype. Consider the task of implementing a bitwise AND operation for these datatypes. Because the physical implementation of bitwise AND for both signed and unsigned datatypes is the same, it can be implemented using the SR_AND instruction as follows:

```
regfile SR 16 8 s        // a 16-bit register file
ctype SS 16 16 SR default// one ctype (signed short)
ctype US 16 16 SR        // another ctype (unsigned short)

operation SR_AND {out SR srout, in SR sin1, in SR sin2} {} {
     assign srout = sin1 & sin2;
}

proto SR_AND {out SS r, in SS s, in SS t} {} {
     SR_AND r, s, t;
}
```

An instruction prototype is specified for the SR_AND instruction, and is defined using the type SS. Note that this prototype is optional, because the ctype SS is declared to be the default ctype for the SR register file. Thus the SS ctype would automatically be assumed to be the ctype for any instruction that operates on the SR register file. With this prototype declaration, in a C/C++ program, the intrinsic SR_AND can only be used with variables of type SS. However, you want to perform the AND operation on variables of both types, and because the physical implementation is the same for both types, you do not want to create two separate instructions. This problem can be solved by using prototype declarations as follows:

```
proto SS_AND {out SS r, in SS s, in SS t} {} {
     SR_AND r, s, t;
}
proto US_AND {out US r, in US s, in US t} {} {
     SR_AND r, s, t;
}
```

The prototypes declared above create two new intrinsics available to the C/C++ programmer. The SS_AND intrinsic performs the AND operation on variables of type SS, while the US_AND intrinsic performs the AND operation on variables of type US. Both these intrinsics translate to the SR_AND instruction in assembly code. Thus, these prototypes do not change the assembly language syntax. Assembly language programmers must still use the SR_AND instruction in their code.

Note that it is possible to use the SR_AND instruction to create the US_AND prototype for variables of type US, even though the instruction prototype for SR_AND uses the SS ctype.

16.7 Specifying Instruction Idioms

Instruction idioms can be specified quite easily using proto statements.

16.7.1 Instruction Idiom Example

```
proto SS_MOV {out SS r, in SS s} {} {
     SR_AND r, s, s;
}
```

Using the example in Section 16.6.1 “Example” on page 142, this proto defines an SS_MOV C/C++ intrinsic that is implemented by the compiler as an SR_AND of a register with itself.

16.8 Writing Prototypes With Struct C Datatypes

Struct C datatypes allow you to define a ctype that spans multiple entries of a register file. Each entry in the struct C datatype is identified by an *argument-name* in the optional *ctype-list*. Software programmers may view a struct C datatype as a single datatype, even though it is implemented in multiple register file entries. TIE developers may implement a custom operation to access the entire struct C datatype, or implement operations to access each field of the struct C datatype and use protos to package them together. Details of struct C datatype are described in “C Datatype (ctype) Sections” on page 121.

Struct C datatypes are utilized in prototypes the same way as regular C datatypes. The only difference is that the instructions in the prototypes access the register file entry of the individual field of the struct C datatype. Note that struct C datatypes cannot span across different register files. Use field access operator "->" to access the individual register file entry of the fields. All prototypes utilizing struct C datatypes need to be manually written, including the prototypes to load, store, and move the struct C datatype.

16.8.1 Struct ctype: Example 1

In this example, a struct ctype of 48 bits (int48) is defined on an AR register file. The int48_loadi, int48_storei, and int48_move protos are defined using a sequence of Xtensa core instructions. The conversion protos between int48 and other types can also be implemented. Arithmetic computations on int48 (addition is given as an example) can be implemented using TIE instructions.

```
ctype int48 48 64 AR {int16 hi, uint32 lo}
proto int48_loadi {out int48 res, in int48 * addr, in immediate imm} {}
{
     L32I res->lo, addr, imm;
     L16SI res->hi, addr, imm+ 4;
```

```
}
proto int48_storei {in int48 data, in int48 * addr, in immediate imm}
{} {
     S32I data->lo, addr, imm;
     S16I data->hi, addr, imm + 4;
}
proto int48_move {out int48 res, in int48 data} {} {
     MOV.N res->lo, data->lo;
     MOV.N res->hi, data->hi;
}

operation add.cout {out AR a, in AR b, in AR c, out BR d} {} {
     assign {d, a} = TIEadd(b, c, 1'b0);
}
operation add.c {out AR a, in AR b, in AR c, in BR d} {} {
     assign a = TIEadd(b, c, d);
}

proto add.int48 {out int48 a, in int48 b, in int48 c} { xtbool tmp} {
     add.cout a->lo, b->lo, c->lo, tmp;
     add.c a->hi, b->hi, c->hi, tmp;
}
```

16.8.2 Struct ctype: Example 2

In this example, a 40-bit register file XR is defined. Three ctypes are also defined: default ctype XR, 32-bit ctype xr32, and a 64-bit struct ctype xr32x2. Ctype xr32 sign extends the upper eight bits. Ctype xr32x2 is a vector version of ctype xr32. Designers can define SIMD instructions to explore parallelism and also define SIMD protos using instructions on xr32 ctype to improve program readability.

```
regfile XR 40 16 xr
ctype XR 40 64 XR default
ctype xr32 32 32 XR
ctype xr32x2 64 64 XR {xr32 hi, xr32 lo}

immediate_range immedx4 0 124 4
immediate_range immedx8 0 248 8

operation ld.xr32 {out XR a, in XR *addr, in immedx4 immed}
                    {out VAddr, in MemDataIn32} {
     assign VAddr = addr + immed;
     assign a = {{8{MemDataIn32[31]}}, MemDataIn32};
}
operation st.xr32 {in XR a, in XR *addr, in immedx4 immed}
                    {out VAddr, out MemDataOut32} {
```

```
     assign VAddr = addr + immed;
     assign MemDataOut32 = a[31:0];
}

proto xr32_loadi {out xr32 a, in xr32 *addr, in immediate immed} {} {
     ld.xr32 a, addr, immed;
}
proto xr32_storei {in xr32 a, in xr32 *addr, in immediate immed} {} {
     st.xr32 a, addr, immed;
}
proto xr32_move {out xr32 a, in xr32 b} {} {
     mv.XR a, b;
}

proto xr32x2_loadi {out xr32x2 a, in xr32x2 *addr, in immediate immed}
{} {
     ld.xr32 a->lo, addr, immed;
     ld.xr32 a->hi, addr, immed + 4;
}
proto xr32x2_storei {in xr32x2 a, in xr32x2 *addr, in immediate immed}
{} {
     st.xr32 a->lo, addr, immed;
     st.xr32 a->hi, addr, immed + 4;
}
proto xr32x2_move {out xr32x2 a, in xr32x2 b} {} {
     mv.XR a->hi, b->hi;
     mv.XR a->lo, b->lo;
}
```

16.9 Proto Design Guidelines

Proto constructs connect the C datatypes with the instructions performing computation. Correctly written prototypes are required in order to enable the XCC compiler to generate correct and fast application code. It is the user's responsibility to verify proto behavior as intended. Following are a few guidelines on protos to assist in generating correct and fast code.

XCC assumes that executing a store proto of type `t` immediately followed by a load proto for a valid variable of type `t` will preserve the original variable. In particular, at arbitrary points in the code, if registers are scarce, XCC might save a variable to memory and later restore it using these protos. If `t` is the default ctype, the load and store protos must preserve the entire register because the operating system and debugger will use these protos without knowing the type of the variables being held in the register.

If a user chooses to implement the load, store, and move protos of a ctype, it is recommended that the move proto contains a single instruction. XCC can optimize the application code more efficiently when the move proto is one instruction.

If a proto returns one of the types int8, int16, uint8, uint16, then the proto must ensure that the upper 24/16 bits are appropriately sign or zero extended. Consider the following example:

```
regfile XR 8 16 xr
operation XR_to_int8 {out AR a, in XR b} {} {
     assign a = {{24{b[7]}}, b};
}
proto XR_rtor_int8 {out int8 a, in XR b} {uint32 tmp} {
     XR_to_int8 tmp, b;
}
```

For protos that operate on the AR register file, XCC may better optimize the code if the following guidelines are satisfied.

- For the ctype of the output argument of a proto, always choose the smallest possible ctype. For example, if a proto outputs an 8-bit unsigned value to an AR register file, select uint8 as the output ctype instead of uint16 or uint32.
- For the ctype of the input argument of a proto, if the proto is going to be vectorized, select the smallest possible ctype to facilitate vectorization. See the SIMD Vectorization section in the *Tensilica C Application Programmer's Guide* for vectorizing details.
- For the ctype of the input argument of a proto, if the proto is not going to be vectorized, choose ctype int32 or uint32.

16.10 Writing Correct Prototypes

The TIE compiler checks prototype definitions for certain common errors. This includes the type and direction of the prototype arguments, missing arguments in the declaration, and the special requirements of load and store prototypes as mentioned in Section 16.2 on page 128. Following are some examples of common errors in prototype definitions.

16.10.1 Common Error Example 1

The following is an example of a missing argument declaration in a `loadi` prototype for the VR register file of Section 16.2.1 on page 129:TIE compiler generates an error because argument o is not declared in the argument list.

```
proto vec64_loadi { out vec64 v, in vec64 *p } {} {
     L64I v, p, o;
```

```
}
```

The correct prototype should declare the argument o to be of type *immediate* as follows:

```
proto vec64_loadi { out vec64 v, in vec64 *p, in immediate o} {} {
     L64I v, p, o;
}
```

16.10.2 Common Error Example 2

This is an example of incorrect type declaration in a *storei* prototype for the VR register file of Section 16.2.1 on page 129. The TIE compiler detects that the first argument to the store prototype must be an input and not output, and the second argument has to be a pointer. It also warns that the address pointer is being updated for a non-updating prototype name. This happens if the S64IU instruction implementation is as described in Section 16.3 on page 133.

```
proto vec64_storei { out vec64 v, in vec64 p, in immediate o } {} {
     S64IU v, p, o;
}
```

The correct store prototype should be written as follows, where the S64I instruction is a non-updating version of the S64IU instruction:

```
proto vec64_storei { in vec64 v, in vec64 *p, in immediate o } {} {
     S64I v, p, o;
}
```

16.10.3 Common Error Example 3

This is an example of a load/store prototype that does not load or store all the bits in the ctype. Consider the XR register file of Section 16.2.3 on page 132, which is a 32-bit register file. The instruction RX32 moves a 32-bit value from the XR register file to the AR register file. However, the S16I instruction in the prototype stores only 16 bits to memory. TIE compiler generates an error in this case.

```
regfile XR 32 16 x
ctype x32 32 32 XR

operation RX32 {out AR a, in XR x} {} {
     assign a = x;
}

proto x32_storei { in x32 v, in x32 *p, in immediate o} {int32 t} {
     RX32 t,v;
     S16I t, p, o;
}
```

The correct prototype should store all 32 bits to memory, and is shown using the S32I instruction.

```
proto x32_storei {in x32 v, in x32 *p, in immediate o} {int32 t} {
     RX32 t, v;
     S32I t, p, o;
}
```

17. User Register (user_register) Sections

User registers are virtual registers that are used to move data to/from designer-defined states. In prior architectures[1], data movement is done via the Xtensa core AR register file. Designers can now bypass the AR register file and move data directly between memory and designer-defined states. User registers are not physical registers in the hardware; they are an abstraction that provides a general purpose but well-defined mechanism for software access to designer-defined states. Designer-defined states are described in Chapter 4, “Processor State (state) Sections” on page 15. The mapping of these states to user registers is specified with user_register description sections. Multiple states can be mapped to a user register[2].

Once a user register is defined for a state, designers need to also define how software can access the designer-defined state. In prior versions[3] of the architecture, the TIE compiler automatically creates two new instructions for every user register defined in the configuration; an RUR.<name> instruction to read contents of the user register into an AR register, and a WUR.<name> instruction to write the contents of an AR register to the user register. For details refer to Chapter 17, “Accessing User Registers Using RUR/WUR Instructions” on page 150.

The xfer_proto mechanism can be used to read and write user registers without any side effects. Refer to Chapter 18, “Transfer Prototype (xfer_proto) Sections” on page 155 for details. The benefit of using xfer_proto is that loading and storing does not need to go through AR registers as with WUR/RUR instructions that are automatically created by the TIE compiler. In addition, there is no limitation on the size and number of user registers.

The TIE language does not require every state to be mapped to a user register. However, states that are not mapped to user registers are not accessible in a hardware debug environment, and saving and restoring these states in interrupt or exception handlers and context switching tasks may not be possible.

The TIE language allows designer-defined states and register files to be grouped into coprocessors. Refer to Chapter 19, “Coprocessor (coprocessor) Sections” on page 159 for details. For a state to be grouped into a coprocessor, it must be mapped to a user_register. Furthermore, all processor states mapped to a particular user_register must belong to the same coprocessor.

1. Prior to the RG-2015.0 release.

2. Note that prior to the RG-2015.0 release, the user register width is constrained to 32 bits.

3. Prior to the RG-2015.0 release.

17.1 User Register Syntax

```
user-register-spec ::= user_register name [number] state-expr
name ::= a unique identifier
number ::= an integer representing the user register number
state-expr ::= (state_bits | {state_bits [, state_bits]*})
state-bits ::= (state_name | state_name[from:to] | constant)
state-name ::= name of a previously defined state
```

name is a unique identifier that can be used later to reference the user register. `number` is a unique integer between 0 and 223, which indicates the user register number. This is an optional argument that is only meaningful when no `xfer_proto` is specified on the user register. If it is not specified, the TIE compiler automatically assigns an available number to the user register. A user register cannot have a number if it is larger than 32 bits and `xfer_proto` is specified on the user register. `state-expr` is the mapping expression that bit-by-bit maps one or more states to the user register. Different bits of a state may be mapped to different user registers, subject to the constraint that all bits are mapped. Use this feature to map a state to multiple user registers. Multiple states can be mapped to a single user register by providing a comma separated list of state names enclosed within curly braces. `state-expr` maps the user register and states bit-by-bit. The debugger uses this map to reconstruct the value of the states from the memory locations in which the user register resides. When specifying `xfer_proto` on the user register, it is important to keep the mapping the same when the value is written to memory and when the value is loaded from memory.

17.2 Accessing User Registers Using RUR/WUR Instructions

The feature described in this section is for backward compatibility only. `xfer_proto` provides a better method of accessing user registers. Refer to Chapter 18, “Transfer Prototype (`xfer_proto`) Sections” on page 155 for details.

For every `user_register` defined in a configuration, the TIE compiler automatically creates two new instructions when both of these conditions are satisfied:

- The width of the user register is at most 32 bits
- `xfer_proto` is not defined for the user register

A *read user register* instruction reads the contents of the user register into an AR register, and a *write user register* instruction writes the contents of an AR register to the user register. The mnemonic for the read instruction is `RUR.`<*name*>, and the mnemonic for the write instruction is `WUR.`<*name*>, where *name* is the name of the user register. By accessing a user register through these instructions, you are actually accessing the underlying designer-defined states that are mapped to the user register. These instructions

are similar to the Xtensa ISA's RSR and WSR instructions that are used to access the Xtensa processor special registers. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for definitions of the RSR and WSR instructions.

The RUR.<*name*> and WUR.<*name*> instructions provide an alternative mechanism to read and write designer-defined state registers when xfer_proto is absent. When the primary purpose of mapping states to user registers is to save and restore the states with a context switch, you should map as many states as possible to a single user register. This minimizes the number of instructions required in the save/restore sequence because multiple states can be accessed using a single RUR/WUR instruction.

States can be automatically mapped to user registers by defining them with the optional argument add_read_write The TIE compiler automatically creates the instructions when a state is declared with the optional argument *add_read_write*. A read instruction with the mnemonic RUR.<*name*> (where <*name*> is the name of the state) is created, which reads the contents of the state into an Xtensa core AR register file entry. Similarly, a write instruction with the mnemonic WUR.<*name*> is created, which writes the contents of an AR register to the state.

If the width of the state is less than 32 bits, RUR.<*name*> reads a zero extended value into the AR register, and WUR.<*name*> ignores the upper bits of the AR register when writing to the state. For states that are wider than 32 bits, multiple Read/Write instructions are created by the TIE compiler. Bits [31:0] are accessed using the instructions RUR.<*name*>_0, WUR.<*name*>_0, bits [63:32] are accessed using the instructions RUR.<*name*>_1, WUR.<*name*>_1 and so on.

The assembly language names of the automatically generated instructions are WUR.<*name*> and RUR.<*name*>, and the names of the corresponding C intrinsics are WUR_<*name*> and RUR_<*name*> respectively. The C intrinsic name replaces the period with an underscore character, as is done for any TIE instruction with a period in its name. This is a shorthand notation to map a state to a user register of the same name. The user_register construct provides a more flexible mechanism to perform this same mapping. Note that states defined with the add_read_write flag count towards the limit of 256 user registers in a TIE description.

Using xfer_proto for a user register is a faster and more flexible method to defining load and store behaviors; therefore, it is the preferred method instead of realizing it with WUR/RUR instructions. With xfer_proto, loading and storing need not go through AR registers as with WUR/RUR instructions, and there is no limitation on the size and number of user registers.

17.3 Examples

In the following example, the state KEYC is mapped to a user register of the same name, and the user register number is not specified. If there is no xfer_proto specified for the user register, this definition is equivalent to the one obtained by specifying the state with the optional add_read_write argument.

```
state KEYC 32
user_register KEYC KEYC
```

In the following example, the 64-bit state DATA is mapped to two user registers, DATA_HI and DATA_LO. Use the instructions WUR.DATA_HI/RUR.DATA_HI to write and read the upper 32 bits of the state, and the instructions WUR.DATA_LO/RUR.DATA_LO to read and write the lower 32 bits of the state. It is also possible to use the instructions WUR0/RUR0 and WUR1/RUR1 to achieve the same effect, as the user register numbers are explicitly specified in the declaration. Note that using the add_read_write argument in the state declaration creates the user registers DATA_0 and DATA_1, which maps to bits [31:0] and [63:32] respectively.

```
state DATA 64
user_register DATA_HI 0 DATA[63:32]  /* first of two user registers */
user_register DATA_LO 1 DATA[31:0]   /* second of two user registers */
```

States larger than 32 bits can be mapped to a single user register as in the following example. Since the user register is larger than 32 bits, users need to define a xfer_proto in order to support loading and storing, as shown in the state DATA:

```
state DATA 64
user_register DATA DATA
```

In the following example, two states, TIME and DATE, are mapped to a single user register. The state DATE is mapped to bits [14:0] and the state TIME is mapped to bits [24:15] of the user register ID. Both states are written when a WUR.ID instruction is executed and both states are read when a RUR.ID instruction is executed.

```
state TIME 10
state DATE 15
user_register ID {TIME, DATE} /* two states mapped to a single entry */
```

17.4 Implementation Restrictions

Following are the implementation restrictions for the user_register sections:

The user registers associated with a xfer_proto do not have user register numbers assigned, and there are no limitations on the number of user registers that can be defined. However, the user registers without an associated xfer_proto are numbered from 0 to 223, even though the Xtensa Instruction Set Architecture (ISA) allows up to 256 user registers. The TIE compiler generates a warning for any user register number in the range 224 to 255 because this space is reserved for Xtensa use with configuration options such as the Vectra LX DSP Engine and Floating Point Unit. Use of the numbers in this range may result in compatibility problems with certain Xtensa configuration options.

Mapping of the same bit(s) of a state to multiple user registers is not allowed. In the following example, bits [2:0] of state COUNTER are mapped to two different user registers, which is not allowed.

```
state COUNTER 10
user_register C1 COUNTER
user_register ID COUNTER[2:0]      // bits already mapped to C1
```

Note that this rule does not preclude a state from being mapped to multiple user registers as long as each bit is mapped to a unique user register. Thus, the following example is legal.

```
state MYSTATE 20
user_register MY1 MYSTATE[ 9: 0]
user_register MY2 MYSTATE[19:10]
```

A user register can have the same name as a state, if these conditions are satisfied:

- All the bits of the state must be mapped to the user register.
- No other state bits can be mapped to the same user register.

18. Transfer Prototype (xfer_proto) Sections

The xfer_proto construct provides a faster way to load and store user registers than using WUR/RUR instructions. User registers can be stored to or loaded from memory directly with the proto constructs specified in a xfer_proto construct without going through AR registers as in cases where RUR/WUR instructions are used. If a xfer_proto is defined on a user register, the TIE compiler no longer generates the RUR/WUR instructions for the user register. Instead, interrupt and exception handlers, and context switching tasks related to the user register are carried out with the proto constructs defined in the xfer_proto. For user registers that are larger than 32 bits, the use of xfer_proto is mandatory and serves as the only way to load and store a user register.

18.1 Transfer Prototype Syntax

```
xfer_proto-def ::= xfer_proto type ep1 ep2{proto-name, proto-name}
ep1 ::= endpoint
ep2 ::= endpoint
type ::= [imm]
endpoint ::= ureg {ureg-name} || sysmem {void}
proto-name ::= a previously defined proto name, or NONE if no transfer
is specified in the direction.
ureg-name ::= a previously defined user register name
```

type indicates the type of the transfer prototype. Currently, the only supported type is *imm*, which requires that the two protos specified at the end of the definition have one memory base address pointer argument, and one immediate offset argument. *ep1* and *ep2* are the two endpoints for data transfer. They can be a user register (*ureg*), or memory (*sysmem*). The name of the user register should be provided for the user register (*ureg*). The first and second *proto-name* specifies the protos used to transfer data from *ep1* to *ep2* and *ep2* to *ep1*, respectively. If there is no data transfer in one direction (for example, writing to ROM is not allowed), use NONE as the corresponding proto name.

18.2 Example

The following example illustrates how a xfer_proto is used to save and restore a 64-bit state without a RUR or WUR instruction.

Suppose we have a 64-bit user register defined on two states:

```
state state_a 32
state state_b 32
user_register my_ureg {state_a, state_b}
```

The operations and protos for loading/storing the user register from/to memory are as follows:

```
regfile vec64 64 16 v
immediate_range imm 0 248 8
operation load_my_ureg {in vec64 *p, in imm i} {out VAddr, in
MemDataIn64, out state_a, out state_b} {
    assign VAddr = p + i;
    wire [63:0]tmp = MemDataIn64;
    assign state_a = tmp[31:0];
    assign state_b = tmp[63:32];
}
proto load_my_ureg {in vec64 *p, in immediate i} {} {
    load_my_ureg p, i;
}
operation store_my_ureg {in vec64 *p, in imm i} {out VAddr, out
MemDataOut64, in state_a, in state_b} {
    assign VAddr = p + i;
    assign MemDataOut64 = {state_b, state_a};
}
proto store_my_ureg {in vec64 *p, in immediate i} {} {
    store_my_ureg p, i;
}
```

Since my_ureg is a user register larger than 32 bits, a xfer_proto has to be defined:

```
xfer_proto imm ureg{my_ureg} sysmem{void} {store_my_ureg,
load_my_ureg}
```

In the example above, the proto store_my_ureg is used for transferring data from the user register my_ureg to memory. The memory address is given by the two arguments of the proto: memory address base p and an offset i. The arguments of the proto load_my_ureg are identical to those of store_my_ureg. The proto load_my_ureg is used for transferring data from memory to the user register.

When transferring data from a user register to memory is not needed, as in the case where the memory is read only, we can ignore the proto store_my_ureg and replace it with NONE in the xfer_proto definition above:

```
xfer_proto imm ureg{my_ureg} sysmem{void} {NONE, load_my_ureg}
```

18.3 Implementation Restrictions

If a user does not write xfer_proto constructs, the TIE compiler does not generate them automatically. The user register uses WUR/RUR instructions to load and save, as described in Chapter 17.

Any user register that is more than 32 bits wide must have an associated xfer_proto.

The proto constructs used in xfer_proto should meet the following requirements:

- The protos should access all parts of the user register defined in the xfer_proto, i.e., all the states associated with the user register should be accessed by the instructions included by the proto.
- The instructions included in the proto that store the user register value to memory must store the value in the same bit ordering as specified in the user register.
- For the imm type of xfer_proto, which is the only type of xfer_proto supported for now, the protos should have exactly two arguments. The first argument is used as a base pointer, which should be a pointer type. The second argument is an offset, which should be of immediate type.
- The direction of the proto's first argument can only be input, which means that the base pointer cannot be modified by the proto.

19. Coprocessor (coprocessor) Sections

The TIE language `coprocessor` construct groups together a set of processor states (TIE state and register files) or interfaces (TIE ports, queues, and lookups). It serves to manage them as a group, such as for controlling their access, for conditional or lazy context-switch, or for display in debugger and related tools.

The Xtensa ISA provides support for up to sixteen general purpose coprocessors. Choosing the coprocessor option in the Xtensa Processor Generator adds a special register called `CPENABLE` to the Xtensa processor. This register has a bit associated with each coprocessor, which is used to enable or disable the coprocessor.

19.1 Coprocessor Syntax

```
coprocessor-def ::= coprocessor name [number] {state-list | port-list}
name ::= a unique identifier
number ::= a unique number between 0-15
state-list ::= list of previously defined states and register files
port-list ::= list of previously defined import_wire, exported state,
queue and lookup
```

name is a unique symbolic name for the coprocessor. `number` is a unique number between 0 and 15, assigned to the coprocessor. The number must also be between 0 and (`n-1`) where n is the number of coprocessors allowed in your Xtensa processor. The number assigned to the coprocessor is only significant in that it is also the number of the bit in the `CPENABLE` register used to enable and disable the coprocessor. Specifying this number is optional. If it is not specified, it is automatically assigned by the TIE compiler. *state-list* is a comma separated list of designer-defined states and register files that belong to this coprocessor. *port-list* is a comma-separated list of `import_wire`, exported state, queues and lookups that belong to this coprocessor. Note that a *state-list* cannot contain any interfaces such as `import_wire`, and so forth. A *port-list* cannot contain any state that is not exported or register files. Thus a coprocessor can contain either TIE ports/queues/lookups or states & register files, but not both.

19.2 Use of Coprocessors for Context Switching

One use of the coprocessor construct is to speed up context switching. The main idea is to avoid unnecessary saving and restoring of state on a context switch in cases where only some of the contexts involve access to that collection of state or register files. Two common approaches to context switching of coprocessor state are to do it conditionally, or on-demand (“lazy” context switch).

Once a set of designer-defined states and register files are grouped together in a `coprocessor`, access to these states and register files is controlled through the appropriate bit of the CPENABLE register. If the bit is set to 1, the coprocessor is enabled, and any instruction accessing the state or register file executes normally. If the bit is set to 0, the coprocessor is disabled. Any instruction accessing a state or register file corresponding to a disabled coprocessor will take a coprocessor exception. For details about the CPENABLE special register and coprocessor exceptions, see the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

In a conditional context switch implementation, TIE state and register files mapped to a coprocessor are saved and restored during a context-switch, only if the relevant context is marked as making use of that coprocessor. This requires specifying upon creation of the context (e.g. thread) which coprocessor(s) it can use, but has the advantage that no space needs to be allocated for saving state of coprocessors the context does not use. A simple implementation may save coprocessor state when switching out of a context that uses it, and restore it when switching back into such a context. Note that this save and restore are unnecessary when switching back to the same such context from other context(s) that did not use that coprocessor, for example when only one context among many uses the coprocessor. At the cost of a little bit more overhead, an implementation may, upon switching from a context that uses a coprocessor to one that does not, skip saving the coprocessor's state and instead track the context that "owns" this state; upon later switching to a context that does use that coprocessor, a context switch of its state is done only if the new context does not already own it. Whatever approach is used, it is common to save and restore the CPENABLE register, such that access to a coprocessor is only enabled for contexts that use it: coprocessor exceptions can thus be handled as errors.

In a lazy or on-demand context switch implementation, TIE state and register files mapped to a coprocessor are not saved and restored at context switch time. Instead, the coprocessor is initially disabled so that if a new context accesses any state or register file, a coprocessor exception is raised. The exception handler enables the coprocessor, saves the coprocessor's state for its last owner, and restores its state for the new owner (that is trying to access it) before allowing the instruction to execute. If a context never accesses the coprocessor state, no coprocessor exception is raised and the coprocessor state is not saved and restored: the only overhead is the saving/restoring and/or setting of the CPENABLE register upon context-switch. There is a trade-off here, because taking a coprocessor exception typically involves a fair bit more overhead than checking at context-switch time whether to save and restore a coprocessor. Refer to the *Xtensa Microprocessor Programmer's Guide* and the *Xtensa System Software Reference Manual* for details on implementing a *lazy* context switch using the Xtensa coprocessor option and TIE coprocessor definitions.

For a designer-defined `state` to be included in a `coprocessor`, it must be mapped to a `user_register`. A state can be mapped to a `user_register` by using the optional `add_read_write` argument in its declaration. This mapping can also be provided as

described in Section 17 “User Register (user_register) Sections” on page 149. The user register construct allows multiple states to be mapped into a single user_register. For such a mapping, all states mapped to a particular user_register must belong to the same coprocessor.

A coprocessor consisting of TIE state or register files cannot contain any interfaces such as import_wire, export state, queue or lookups. It is not possible to save or restore the values of external interfaces. Interfaces grouped into a coprocessor for the purpose of protection as explained in Section 19.3 must use a different coprocessor.

19.2.1 Example

The following piece of code defines a coprocessor named COP1 and assigned the number 0. This coprocessor is enabled or disabled by setting or clearing bit 0 of the CPENABLE register.

```
state FOO 40 add_read_write
regfile XR 64 8 x

coprocessor COP1 0 {FOO, XR}

operation MV_FOO {out XR a} {in FOO} {
     assign a = {24'b0, FOO};
}
```

19.3 Use of Coprocessors for Protection

In microprocessor systems, it is common for different application code running on the processor to have different levels of access to system resources. A Memory Management Unit (MMU) is typically used to selectively grant access to different regions of memory and to memory mapped I/O devices.

In an Xtensa configuration with MMU, access to memory regions is controlled through this conventional mechanism. However, Xtensa processors also support interfaces such as import_wire, export state, queues and lookups that allow the code running on the processor to access resources external to the processor. The coprocessor construct is used to selectively grant or deny access to these resources.

In an Xtensa configuration with MMU, all import_wire, export state, queues, and lookups must be mapped to one or more coprocessors. Not doing so will result in a TIE compiler error when the file is compiled. The operating system can now disable or enable the coprocessor(s) that these interfaces are mapped to. This is done by setting or clearing the appropriate bit of the CPENABLE register. If the coprocessor is enabled, any application code running on the Xtensa processor has access to these interfaces and instructions that read or write any of these interfaces execute normally. If on the other hand, the co-

processor is disabled, any instruction accessing an interface belonging to the disabled coprocessor will take a coprocessor exception. The exception handler can then decide to grant access to the interface, or take other appropriate action as necessary. In a typical scenario, the operating system would decide which process has access to which coprocessor(s), and enable those coprocessors before executing the process.

A coprocessor consisting of interfaces such as queues, lookups, and so forth cannot contain any TIE state or register files. TIE state and register files grouped into a coprocessor for the purpose of implementing lazy context switching as explained in Section 19.2 must use a different coprocessor.

19.3.1 Example

The following piece of code defines three coprocessors named COP2, COP3 and COP4.

```
regfile XR 64 8 x

import_wire IMPW 32
queue OPQ 40 out
queue IPQ 40 in

coprocessor COP2 2 {XR}
coprocessor COP3 3 {IMPW}
coprocessor COP4 4 {OPQ, IPQ}

operation RD_IMPW {out XR a} {in IMPW} {
     assign a = {32'b0, IMPW};
}

operation RD_IPQ {out XR a} {in IPQ} {
     assign a = {24'b0, IPQ};
}

operation WR_OPQ {in XR a} {out OPQ} {
     assign OPQ = a[39:0];
}
```

COP3 protects accesses to the import_wire IMPW, and is enabled or disabled by setting or clearing bit 3 of the CPENABLE register. COP4 protects access to the input queue IPQ and output queue OPQ. This coprocessor is enabled or disabled by setting or clearing bit 4 of the CPENABLE register. If this coprocessor is disabled, the instruction RD_IPQ will not read any data from IPQ, and will take an exception. Similarly, the instruction WR_OPQ will not write any data to OPQ, and will take an exception. Coprocessor COP2 which consists of the XR register file is for lazy context save and restore, and is not related to protection.

19.4 Multiple Coprocessor Exceptions

If an instruction accesses multiple coprocessors and more than one coprocessor is disabled, the instruction will take multiple coprocessor exceptions. These exceptions are ordered by the number of the coprocessor, with the lowest coprocessor number having the highest priority. For example, if the instruction `RD_OPQ` in Section 19.3.1 executes when both coprocessor 2 and 4 are disabled, it will first take an exception for coprocessor 2 for context-switch of the register file `XR`, and then for coprocessor 4 for denied access to the queue `IPQ`.

19.5 Implementation Restrictions

Following are the implementation restrictions for the `coprocessor` sections:

- The Floating Point Unit option defines Coprocessor number 0. Thus TIE extensions written for an Xtensa processor configuration that includes the Floating Point Unit cannot use coprocessor number 0.
- The Vectra LX DSP Engine option defines Coprocessor number 1. The HiFi2/3 Audio Engine option also defines Coprocessor 1. ConnX D2 option defines Coprocessor 2. Thus, TIE extensions written for an Xtensa processor configuration that includes the Vectra LX DSP Engine or the HiFi2/3 Audio Engine cannot use coprocessor number 1. TIE extensions written for an Xtensa processor configuration that includes the ConnX D2 DSP Engine cannot use coprocessor number 2.
- An Xtensa configuration may contain a maximum of 16 coprocessors. The Floating Point Unit, the Vectra LX DSP Engine, the HiFi2/3 Audio Engine, and the ConnX D2 configuration options each declares a coprocessor that counts towards this limit.
- The coprocessor number assigned to a particular coprocessor in a TIE description must be in the range 0 to (`n-1`), where n is the maximum number of coprocessors supported by your Xtensa configuration. The maximum number of coprocessors supported is a configuration parameter you select on the Xtensa processor generator (XPG).
- One coprocessor can be declared multiple times for different states, registers, or ports. In this case, their coprocessor names must match exactly. If coprocessor number is provided for any of the coprocessor statements, all coprocessor statements must have the same number.

20. Function (function) Sections

A TIE function is similar to a function in C or Verilog. It encapsulates some computation, which can then be used in different places in a TIE description. TIE functions can be used for modular code development by providing one description of a function that gets instantiated in multiple places. Functions can also be used to share hardware (combinational logic) between multiple semantic blocks by declaring them as “shared.“ Functions also provide a mechanism for describing iterative instructions, which are instructions that use the same piece of hardware in multiple cycles of its execution.

20.1 Function Syntax

```
function-def ::= function [width] name (arg-list)
                          [shared | slot_shared] {computation}
name ::= a unique identifier
width ::= [from:to]
arg-list ::= [width] arg-name [, [width] arg-name]*
shared ::= optional keyword, indicating that there is a single copy
          of this hardware in the design
slot_shared ::= optional keyword, indicating that there is only one
               copy of this hardware per slot index
computation ::= see Chapter 28
from ::= integer bit index
to ::= integer bit index
arg-name ::= a unique identifier within the scope of the function
```

name is a unique identifier that can be used later to reference the function. It also serves as a variable inside the computation block for the return value of the function. *width* specifies the size of the function return value, and is optional if the return width is one. *arg-list* is a comma separated list of one or more input arguments to the function. Each input argument has a formal name (*arg-name*) and a width that specifies the size of that input. The *to* index of *width* is required to be zero. The specification of *width* is optional, and when not specified, it defaults to one. The optional keyword *shared* indicates that there is only one copy of this function in the hardware, and this is shared by all instructions that need it. Similarly, the optional keyword *slot_shared* indicates that there is one copy of this function (RTL logic) per slot index, and it is shared by all instructions belonging to that slot index. The computation section describes the behavior of the function, and is described in Chapter 28, “Computation Sections” on page 215.

Following are the valid variables inside a function’s computation block:

- The function name, in the LHS of an assignment
- All the input argument names, in the RHS of an assignment

- All wires declared explicitly in the computation
- TIE tables
- TIE modules and other functions

20.2 Simple Functions

Simple functions are used primarily to make the TIE code modular and more readable by encapsulating commonly used logic in a function. Every reference to a simple function instantiates a new copy (in hardware) of the logic implemented by the function. Simple functions can be nested; that is, a function can call another simple function. Consider the following example TIE code, which defines a function named addsub that computes either the sum or the difference of two 8-bit values. This function is invoked by another function, as8x4, which performs four 8-bit additions/subtractions in parallel. The function as8x4 is used in instructions ADD8X4 and SUB8X4.

```
function [7:0] addsub ([7:0] a, [7:0] b, add) {
     wire [7:0] tmp = add ? b : ~b;
     wire cin = add ? 0 : 1;
     assign addsub = TIEadd(a, tmp, cin);
}

function [31:0] as8x4 ([31:0] a, [31:0] b, add) {
     wire [7:0] t0 = addsub(a[ 7: 0], b[ 7: 0], add);
     wire [7:0] t1 = addsub(a[15: 8], b[15: 8], add);
     wire [7:0] t2 = addsub(a[23:16], b[23:16], add);
     wire [7:0] t3 = addsub(a[31:24], b[31:24], add);

     assign as8x4 = {t3, t2, t1, t0};
}

operation ADD8X4 {out AR sum, in AR in0, in AR in1} {} {
     assign sum = as8x4(in0, in1, 1'b1);
}

operation SUB8X4 {out AR diff, in AR in0, in AR in1} {} {
     assign diff = as8x4(in0, in1, 1'b0);
}
```

There are two references to the function as8x4; one each in the instructions ADD8X4 and SUB8X4. Thus, there are two copies of this function in hardware. Each instance of the function as8x4 has four references to the function addsub. Thus, the hardware has a total of eight instances of the function addsub. This example demonstrates the use of functions in TIE descriptions to improve readability and make the description more structured.

20.3 Shared Functions

A shared function is one for which there is a single copy or instance in the hardware, and this instance is shared among all of its users. Thus, shared functions provide a means to share hardware between different TIE instructions. The example in Section 20.2 on page 166 can be rewritten to make use of shared functions as follows:

```
function [7:0] addsub ([7:0] a, [7:0] b, add) {
     wire [7:0] tmp = add ? b : ~b;
     wire cin = add ? 0 : 1;
     assign addsub = TIEadd(a, tmp, cin);
}

function [31:0] as8x4 ([31:0] a, [31:0] b, add) shared {
     wire [7:0] t0 = addsub(a[ 7: 0], b[ 7: 0], add);
     wire [7:0] t1 = addsub(a[15: 8], b[15: 8], add);
     wire [7:0] t2 = addsub(a[23:16], b[23:16], add);
     wire [7:0] t3 = addsub(a[31:24], b[31:24], add);

     assign as8x4 = {t3, t2, t1, t0};
}

operation ADD8X4 {out AR sum, in AR in0, in AR in1} {} {
     assign sum = as8x4(in0, in1, 1'b1);
}

operation SUB8X4 {out AR diff, in AR in0, in AR in1} {} {
     assign diff = as8x4(in0, in1, 1'b0);
}
```

Note that the keyword `shared` has been added to the declaration of function `as8x4`. As a result, there is only one copy of the function `as8x4` in hardware, and this is shared between the instructions `ADD8X4` and `SUB8X4`. Thus, the amount of hardware required to implement the datapath is cut in half. The TIE compiler automatically generates the muxing logic to share this function's datapath between the two instructions. An alternate way to specify this sharing (without using shared functions) is to write a semantic section that implements the instructions `ADD8X4` and `SUBX4`. This is described in more detail in Chapter 9, "Instruction Semantics (`semantic`) Sections" on page 61. In general, the use of shared functions simplifies the specification of instructions that share logic and makes the code more readable. Writing semantic sections may make the code less readable but provides more flexibility in sharing logic between instructions.

Shared functions cannot be nested; that is, they cannot be called by other functions. They can only be called in `operation`, `reference`, or `semantic` sections.

20.3.1 Shared Functions and Resource Interlocks

When a function is declared as being shared, there is one copy of this function in hardware that is shared by multiple instructions. While this results in hardware area savings, it can potentially generate stall cycles due to resource interlocks. A resource interlock occurs when two instructions want to access the same shared function in the same clock cycle. Because the hardware cannot service both instructions at the same time, the second instruction has to stall.

Consider the effect of adding the following `schedule` statement for instructions ADD8X4 and SUB8X4 in a 5-stage pipeline configuration. Chapter 8, “Schedule (`schedule`) Sections” on page 47 discusses this construct in detail.

```
schedule schadd {ADD8X4} {
     use in0 1; use in1 1; def sum 1;
}

schedule schsub {SUB8X4} {
     use in0 2; use in1 2; def diff 2;
}
```

Without the preceding schedule statements, both instructions would take the default schedule, which is to use and def all the operands in cycle 1, or the E stage of the processor pipeline. However, with the preceding schedule, the SUB8X4 instruction has a use and def of 2 for its operands. As a result, the SUB8X4 instruction will use the shared function `as8x4` when it is in its M stage. Conversely, the ADD8X4 instruction will use the shared function when it is in its E stage. Consider the situation when the SUB8X4 instruction is executed followed immediately by the ADD8X4 instruction. If these two instructions were to go down the processor pipeline one after the other, the result would be the situation illustrated in Table 20–13.

Table 20–13. Resource Conflict Due to Shared Functions

Clock	Pipeline Stage			
Cycle	R	E	M	W
1	SUB8X4			
2	ADD8X4	SUB8X4		
3		ADD8X4	SUB8X4	
4			ADD8X4	SUB8X4
5				ADD8X4

As shown in Table 20–13, the SUB8X4 instruction is in its M stage and the ADD8X4 instruction is in its E stage during cycle 3. Thus, both instructions want to use the function as8x4 in this cycle, which is not possible. To prevent this from occurring, the hardware will stall the ADD8X4 instruction in the R stage for one cycle, thus avoiding the resource conflict. This is illustrated in Table 20–14.

Table 20–14. Pipeline Stalls to Avoid Resource Interlocks

Clock	Pipeline Stage			
Cycle	R	E	M	W
1	SUB8X4			
2	ADD8X4	SUB8X4		
3	ADD8X4		SUB8X4	
4		ADD8X4		SUB8X4
5			ADD8X4	

Note that if both ADD8X4 and SUB8X4 use the shared function in the same pipeline stage, this problem does not occur. Resource interlocks can also result in illegal bundles of FLIX operations as illustrated in Section 20.4.

20.4 Slot Shared Functions For LX cores

A slot shared function is a hybrid of simple functions and shared functions. In FLIX designs with multiple slots, a slot shared function is one that is shared between instructions in the same slot, but not across slots (or more precisely, shared between instructions belonging to the same slot index, but not across different slot indices). Consider the following FLIX design that has two slots. slotx implements the Xtensa ISA instructions L32I and BEQ, and the designer-defined instruction ADD8X4. sloty implements two designer-defined instructions ADD8X4 and SUB8X4.

```
length l64 64 {InstBuf[63:60] == 14}
format f64 l64 {slot_x, slot_y}

slot_opcodes slot_x {L32I, BEQ, ADD8X4}
slot_opcodes slot_y {ADD8X4, SUB8X4}
```

Assume that the instructions ADD8X4 and SUB8X4 are defined as in Section 20.3 (without the schedule statement). Both instructions use the function as8x4 that has the attribute shared, so there is only one instance of this function in hardware. There is no problem in sharing this hardware between the ADD8X4 and SUB8X4 instructions of slot_y, because only one can be issued at a time. However, pairing the ADD8X4 instruction of slot_x with either instruction of slot_y presents a problem. In this situation, two instructions need to use the same piece of hardware at the same time.

This is another example of a resource interlock, which is explained in Section 20.3.1 on page 168. Resource interlocks between two (or more) operations of the same FLIX instruction cannot be resolved with stall cycles because the entire FLIX instruction is viewed as one entity. If the instruction stalls, both operations stall, and if the instruction proceeds, both operations proceed. Thus, such a pairing of operations is illegal and will result in an error when the program is being assembled. As a result of this resource interlock problem, the ADD8X4 instruction cannot be issued from slot_x (except when it is paired with a NOP in slot_y).

One way to resolve this resource interlock problem is to *not* define the function as8x4 to be shared. It would then be a simple function as illustrated in Section 20.2. The ADD8X4 instruction of slot_x has its own copy of the hardware for this computation, and hence can be paired with either instruction of slot_y. Note that now there are three instances of the function as8x4 in the hardware; one in slot_x and two in slot_y. Only one of the two instances in slot_y is in use at any given time, thus resulting in wasted hardware.

The best solution for this scenario is to share the hardware between the two instructions of slot_y, but let slot_x have its own copy. To do so, declare the function as8x4 to be *slot_shared* as shown in the following example:

```
function [31:0] as8x4 ([31:0] a, [31:0] b, add) slot_shared {
     wire [7:0] t0 = addsub(a[ 7: 0], b[ 7: 0], add);
     wire [7:0] t1 = addsub(a[15: 8], b[15: 8], add);
     wire [7:0] t2 = addsub(a[23:16], b[23:16], add);
     wire [7:0] t3 = addsub(a[31:24], b[31:24], add);

     assign as8x4 = {t3, t2, t1, t0};
}
```

Slot shared functions are shared by all instructions that belong to the same slot index. The sharing is not based on slot names. Slot indices are explained in detail in Section 10.3 “Slot Indices of a Format” on page 89. To illustrate this point, consider the following example:

```
length l64 64 {InstBuf[3:0] == 15}
format f64a l64 {slot_a, slot_b, slot_a}
format f64b l64 {slot_p, slot_q}

slot_opcodes slot_a {ADD8X4, SUB8X4, JX}
slot_opcodes slot_b {BGEZ, ADDI}
slot_opcodes slot_p {L16SI, S16I, ADD8X4}
slot_opcodes slot_q {AND, XOR, ADDI}
```

In this example, format f64a has three slots and format f64b has two slots. Furthermore, slot_a is present twice in format f64a. In format f64a, the first instance of slot_a is assigned slot index 0, slot_b is assigned slot index 1, and the second instance of slot_a is assigned slot index 2. In format f64b, slot_p is assigned slot index 0 and slot_q is assigned slot index 1.

Assume that the instructions ADD8X4 and SUB8X4 are defined using the slot_shared function as8x4. The two instances of slot_a in format f64a will each have its own copy of the function as8x4, because they have a different slot index. Conversely, slot_p of format f64b has the same slot index (zero) as the first instance of slot_a in format f64a. Thus, these two slots will share a copy of the function as8x4.

20.5 Iterative Instructions

An instruction is defined to be iterative if it uses a shared function more than once in its semantic description. Because there is only one copy of the function in hardware, its use by the instruction is spread over multiple clock cycles. Consider the function mul8 and its use in the instruction MUL8X2 as defined in the following example:

```
function [15:0] mul8 ([7:0] m0, [7:0] m1) shared {
     assign mul8 = TIEmul(m0, m1, 1'b1);
}

operation MUL8X2 {out AR prod, in AR a, in AR b} {} {
     wire [15:0] p0 = mul8(a[ 7: 0], b[ 7: 0]);
     wire [15:0] p1 = mul8(a[23:16], b[23:16]);
     assign prod = {p1, p0};
}

schedule mul8x2_sched {MUL8X2} {
     use a 1; use b 1;
     def prod 2;
}
```

Because function mul8 is defined to be shared, there is only one copy of it in the hardware. However, the instruction MUL8X2 uses this function twice, with two different sets of operands. Thus, each use of this function has to occur in a different clock cycle. This makes it necessary to define a schedule for this instruction. The input operands a and b are available in cycle 1, and the result prod is defined in cycle 2. Thus, the first iteration of the multiply happens in cycle 1, when a[7:0] gets multiplied with b[7:0] and the result (assigned to wire p0) is held in a temporary holding buffer. The second iteration of the multiply happens in cycle 2, when a[23:16] gets multiplied with b[23:16]. In cycle 2; the results of these two multiply operations are concatenated to form the final result of the instruction.

Iterative instructions take less hardware as compared to fully pipelined, multi-cycle instructions. Conversely, iterative instructions result in processor pipeline stalls if a second instruction is issued while the first one is still executing. Thus, the choice between an iterative instruction and a fully pipelined multi-cycle instruction is based on the tradeoff between hardware area and execution performance.

By definition, iterative instructions are multi-cycle instructions and hence require a schedule statement. Chapter 8 provides more details about this TIE construct.

20.6 Scheduling of Functions

The TIE Compiler automatically schedules functions to be used in the appropriate pipeline stage based on the schedules of the instruction operands. The current TIE language syntax does not allow the explicit specification of the pipeline stages in which functions are used in an instruction. Consider the following example, which defines an instruction ADD8X4 that uses two shared functions, addsub8 and addsub16.

```
function [7:0] addsub8 ([7:0] a, [7:0] b, add) shared {
     wire [7:0] tmp = add ? b : ~b;
     wire cin = add ? 0 : 1;
     assign addsub8 = TIEadd(a, tmp, cin);
}

function [15:0] addsub16 ([15:0] a, [15:0] b, add) shared {
     wire [15:0] tmp = add ? b : ~b;
     wire cin = add ? 0 : 1;
     assign addsub16 = TIEadd(a, tmp, cin);
}

operation ADD8X4 {out AR sum, in AR a, in AR b} {} {
     wire [ 7:0] t0 = addsub8( a[ 7: 0], b[ 7: 0], 1'b0);
     wire [15:0] t1 = addsub16(a[23: 8], b[23: 8], 1'b0);
     wire [ 7:0] t2 = addsub8( a[31:24], b[31:24], 1'b0);
     assign sum = {t2, t1, t0};
}

schedule add8x4 {ADD8X4} {
     use a 1; use b 1; def sum 5;
}
```

In this example, the specific pipeline stages in which the addsub8 and addsub16 functions are used is not defined. The TIE compiler will schedule the use of the functions at an appropriate stage between 1 (when the inputs become available) and 5 (when the final result is required).[1]

1. Such multi-cycle TIE instructions require behavioral retiming during synthesis.

There may be specific situations where you want to control the stage in which a function is used by an instruction. To do so, provide appropriate def stages for wires in the semantic description as illustrated in the following example.

```
operation ADD8X4 {out AR sum, in AR a, in AR b} {} {
     wire [ 7:0] t0 = addsub8( a[ 7: 0], b[ 7: 0], 1'b0);
     wire [15:0] tmp_a0 = a[23:8];
     wire [15:0] tmp_b0 = b[23:8];
     wire [15:0] t1 = addsub16(tmp_a0, tmp_b0, 1'b0);
     wire [ 7:0] tmp_a1 = a[31:24];
     wire [ 7:0] tmp_b1 = b[31:24];
     wire [ 7:0] t2 = addsub8( tmp_a1, tmp_b1, 1'b0);
     assign sum = {t2, t1, t0};
}

schedule add8x4 {ADD8X4} {
     use a 1; use b 1;
     def t0 1;
     def tmp_a0 2; def tmp_b0 2;
     def t1 3;
     def tmp_a1 4; def tmp_b1 4;
     def sum 5;
}
```

The input operands a and b are not available until stage 1, and the wire `t0` has a def stage of 1. Thus, the first call to function `addsub8` has to be scheduled for stage 1. Wires `tmp_a0` and `tmp_b0` have a def stage of 2, which implies that their earliest use stage is 3. These wires are the input operands to the function `addsub16`, whose output `t1` has a def stage of 3. Thus, function `addsub16` is scheduled for stage 3. Wires `tmp_a1` and `tmp_b1` have a def stage of 4, which implies that their earliest use stage is 5. The second call to function `addsub8` is scheduled for stage 5, as the output of this function (`t2`) is required in stage 5 to create the output operand `sum`.

As another example, the execution of the `addsub16` function and the second iteration of the `addsub8` function are both scheduled for stage 5 with the following schedule:

```
schedule add8x4 {ADD8X4} {
     use a 1; use b 1;
     def t0 1;
     def tmp_a0 4; def tmp_b0 4; def tmp_a1 4; def tmp_b1 4;
     def sum 5;
}
```

The preceding examples use the schedules of the instruction operands and wires of the operation to indirectly schedule the pipeline stage in which the functions addsub8 and addsub16 are used. Note that it is not possible to assign use stages to the input arguments of a function or to assign a def stage to the return value of the function. It is also not possible to assign def stages to the wires declared within a function.

Every instantiation of a shared function in an operation or semantic description is assumed to be a call to that function. This is true even though it may be possible to perform the computation with only one call. Consider the following example TIE code:

```
state ADD8 1 add_read_write

function [7:0] addsub ([7:0] a, [7:0] b, add) shared {
     wire [7:0] tmp = add ? b : ~b;
     wire cin = add ? 0 : 1;
     assign addsub = TIEadd(a, tmp, cin);
}

operation ADDSUB2 {out AR sum, in AR a, in AR b} {in ADD8} {
     wire [7:0] tmp;
     assign tmp = ADD8 ? addsub(a[7:0], b[7:0], 1'b1) :
                         addsub(a[7:0], b[7:0], 1'b0);
     assign sum = {{31{tmp[7]}}, tmp};
}

schedule adsub {ADDSUB2} {
     use a 1; use b 1; def sum 1;
}
```

The intent of the preceding TIE code is to use the addsub shared function to perform either an addition or subtraction based on the value of the state ADD8. However, as written, this is implemented as two invocations of the addsub function, and the return value of one of them is selected based on the value of the state ADD8. Furthermore, because the operation now has two instances of a shared function, it becomes an iterative instruction that takes at least 2 cycles to execute. Thus the use 1 def 1 schedule for this instruction will generate an error message. To resolve this problem, write the operation description as follows:

```
operation ADDSUB2 {out AR sum, in AR a, in AR b} {in ADD8} {
     wire [7:0] tmp;
     assign tmp = addsub(a[7:0], b[7:0], ADD8);
     assign sum = {{31{tmp[7]}}, tmp};
}
```

Similarly, every shared function inside a semantic is instantiated, even if some operations that share the semantic only utilize a portion of the shared functions.

20.7 Implementation Restrictions

While simple functions can be nested (called from within other functions), shared (or slot_shared) functions cannot be nested. Shared functions can only be called from `operation`, `semantic`, or `reference` sections.

It is not possible to schedule the wires declared within a function. Furthermore, it is not possible to assign the pipeline stage in which a function is used. The use cycle of a function is derived based on the schedule of the instruction that uses the function.

Shared (or slot_shared) functions have to be single cycle functions. While the instruction that instantiates a shared function can be multicycle, the function itself has to execute in a single cycle. Behavioral retiming, which is used to balance the logic between registers of a multicycle TIE instruction, may not move registers into the logic encapsulated by a shared function.

21. Property (property) Sections

Property constructs describe optional attributes of other TIE constructs. These constructs may also describe the relationship between different TIE constructs. Property constructs are separated from the definition of the TIE construct so that the definition can be kept simple and elegant, and the attributes can be flexibly described. Depending on the TIE construct for which the property is described the syntax of the property differs slightly.

21.1 Specialized Operation Property

An instruction set may contain multiple operations performing the same computation, where their only difference is their operands: register file operands, or immediate operands of different sizes. The specialized operation property informs XCC of such relations between operations.

XCC may use this property information in two ways:

- If one operation contains an immediate operand, and the corresponding operand in the other operation is an AR register operand, XCC automatically chooses the operation with register operand when the immediate value does not fit in the immediate operand.
- If both operations contain immediates with different ranges, in cases where an immediate is supported by both variants, XCC will select between both in order to better optimize code. A typical example is using a larger immediate range for a non-FLIX variant and a smaller immediate range for an operation that can be issued in one slot of a 32-bit FLIX instruction. In cases where the immediate is out of range, XCC will correctly use the non-FLIX variant, but in cases where the immediate is in range, XCC will try to bundle the other variant into a FLIX instruction.

The syntax of specialized operation property and examples of uses are as illustrated below:

21.1.1 Property Syntax

```
property-specialized-op ::= property specialized_op oper-name1 oper-
name2
oper-name1 ::= a previously defined operation name
oper-name2 ::= a previously defined operation name
```

oper-name is a previously defined operation name. The sequence of operations do not matter, but note that only two operations can be named. XCC can automatically determine the immediate ranges of the operations. If one operation has AR operands, XCC can map the operand to the corresponding immediate operand of the other operation.

It is the designers responsibility to ensure the two operation perform the same computation without any side effect.

21.1.2 Example 1

In this example, two operations are supplied: SHFT and SHFT_IMM. SHFT takes two AR operands as input, while SHFT_IMM takes one AR operand and one immediate operand. Both operations perform the same computation. Property specialized operation is assigned to this pair of operations.

```
immediate_range immed 0 7 1
operation SHFT_IMM {out AR a, in AR b, in immed imm} {} {
     assign a = b[15:0] << imm;
}
operation SHFT {out AR a, in AR b, in AR c} {} {
     assign a = b[15:0] << c[4:0];
}
property specialized_op SHFT_IMM SHFT
```

Programmers may write SHFT_IMM intrinsic in the C file without calculating the valid immediate value. If it is out of range, XCC automatically converts it to a SHFT intrinsic.

```
a = SHFT_IMM(b, 8);
```

Because 8 is out of the immediate range, it will be converted to:

```
movi a12, 8;
shft a11, a11, a12;
```

21.1.3 Example 2

In this example, two operations are supplied: MYADD_SMALL and MYADD_BIG. The only difference between the two is the immediate range of each operation. Only MYADD_SMALL appears in slot0 and slot1 of the fmt format. This is because in a 32-bit format each instance of the MYADD_SMALL operation can only occupy 16 bits and only a limited immediate range can be encoded. However, the operation MYADD_BIG has a bigger immediate range and can be encoded in the 24-bit Inst slot. A property specialized operation is assigned to this pair of operations.

```
immediate_range immed1 0 12 4
immediate_range immed2 0 508 4
```

```
operation MYADD_SMALL {out AR a, in AR b, in immed1 imm} {} {
     assign a = b[15:0] + b[31:16] + imm;
}
operation MYADD_BIG {out AR a, in AR b, in immed2 imm} {} {
     assign a = b[15:0] + b[31:16] + imm;
}
property specialized_op MYADD_SMALL MYADD_BIG

format fmt 32 {slot0, slot1}
slot_opcodes slot0 {MYADD_SMALL}
slot_opcodes slot1 {MYADD_SMALL}
```

Software programmers can use the intrinsic `MYADD_BIG` in their application. If the immediate value fits to `immed1`, and two instances of the operation can be issued in parallel, XCC can automatically convert `MYADD_BIG` to `MYADD_SMALL` to utilize the parallel computation of `MYADD_SMALL` in the `fmt` format. If the immediate value is outside the range of `immed1`, XCC keeps `MYADD_BIG` operation in the program.

21.2 Ignore State Output Property

If two operations write to the same state, the XCC compiler generates code so that the two operations are executed in the same sequence as they appear in the source application code. In some situations, however, the exact order of the operations is irrelevant to the programmer. Providing XCC the freedom to move operations across state updates may generate more efficient code.

An example of such a situation is the use of a sticky overflow bit. Operations may set the overflow bit in the rare case that overflow happens, but otherwise leave it unchanged. However, the exact operation that causes the overflow may not be important. It is often sufficient to check the value of the overflow bit after multiple operations are executed. While the compiler must not move the computation operations to after the overflow check, the computation operations can be freely reordered amongst each other.

Note that if the results produced by an operation, other than the state, are not used, XCC may remove the operation to improve performance further.

21.2.1 Syntax

```
property-ignore-state-output ::= property ignore_state_output state-
name oper-list
state-name ::= a previously defined state name
oper-list ::= {oper-name [, oper-name]*}
oper-name ::= a previously defined operation name
```

state-name is the name of a state that is accessed by all the operations in *oper-list*. *oper-list* is a list of operations whose execution sequence can be reordered.

It is the user's responsibility to verify that reordering the instructions in *oper-list* may not result in incorrect behavior.

21.2.2 Example 1

In this example, state `overflow` is a 1-bit state indicating whether the operation generates an overflow. Operations `MYADD` and `MYSUB` may set the state in case an overflow happens. The property `ignore_state_output` informs XCC that `MYADD` and `MYSUB` may be reordered. Thus, the execution sequence of `MYADD` and `MYSUB` may be swapped. If the AR register output of the operations is not used, the operation may be eliminated. The TIE code is shown below:

```
state overflow 1 add_read_write
operation MYADD {out AR a, in AR b, in AR c} {out overflow} {
     wire [32:0] res= b + c;
     assign a = res[31:0];
     assign overflow = res[32];
     assign overflow_kill = !res[32];
}
operation MYSUB {out AR a, in AR b, in AR c} {out overflow} {
     wire [32:0] res = b - c;
     assign a = res[31:0];
     assign overflow = res[32];
     assign overflow_kill = !res[32];
}
property ignore_state_output overflow {MYADD, MYSUB}
```

An example C code that exercise the TIE operations is shown below:

```
int foo(int* a, int* b, int c, int d) {
     int i;
     for (i = 0; i < 100; i++) {
          a[i] = MYADD(a[i], c);
          b[i] = MYSUB(c, d);
     }
}
```

It is clear that the AR register result of `MYSUB(c, d)` is a loop invariant. By specifying the property `ignore_state_output`, XCC can hoist the result outside the loop, before the `MYADD` operation. XCC will not perform the optimization without the property.

21.2.3 Example 2

In this example, state overflow has shared_or attribute associated with it, which means multiple operations accessing this state may be scheduled to be executed in the same FLIX instruction. The state is set to the bitwise OR operation of the result of the individual operations. Please refer to Section 4.4 “ORing a State” on page 17 for detailed description of the shared_or attribute.

Instructions MYADD and MYSUB may set the overflow state. The software may check the value of the state and jump to an appropriate error handling routine. However, it is not important to identify the exact instruction that causes the overflow. Thus, the property ignore_state_output is applied to the instructions, which gives XCC permission to exchange the order of the instructions. The shared_or attribute of state overflow allows XCC to FLIX operations MYADD and MYSUB.

```
state overflow 1 1'b0 add_read_write shared_or
operation MYADD {out AR a, in AR b, in AR c} {inout overflow} {
     wire [32:0] res = b + c;
     assign a = res[31:0];
     assign overflow = overflow | res[32];
}
operation MYSUB {out AR a, in AR b, in AR c} {inout overflow} {
     wire [32:0] res = b - c;
     assign a = res[31:0];
     assign overflow = overflow | res[32];
}

property ignore_state_output overflow {MYADD, MYSUB}

format fmt 64 {slot0, slot1}
slot_opcodes slot0 {MYADD}
slot_opcodes slot1 {MYSUB}
slot_opcodes Inst {MYADD, MYSUB}
```

If the following sequence of intrinsics is specified in the C code:

```
c = MYADD(a, b);
d = MYSUB(a, b);
```

XCC may choose to implement it in one FLIX instruction:

```
{ myadd a3, a4, a8   mysub a2, a4 a8 }
```

Depending on the dependence information, by specifying the ignore_state_output property, XCC may also choose to reorder the instruction sequence:

```
mysub a2, a4, a8
myadd a3, a4, a8
```

21.3 Display Format Property

By default, Xplorer displays variables of TIE types as well as TIE registers and states as integer or hexadecimal numbers. For more complicated types, Xplorer allows you to define custom display formats for variables, registers or types that can be more legible. Using the display format property, these formats can also be attached directly in the TIE file to set the default formats used by Xplorer.

The display format property contains two related properties. The `display_format_name` associates the format string with a name. The `display_format_map` maps the states, user registers, or ctypes to the format defined in `display_format_name`.

21.3.1 Syntax

```
property-display-format-name ::= property display_format_name name
"display-format"
name ::= a unique identifier
display-format ::= format string to be displayed in Xplorer

property-display-format-map ::= property display_format_map
                        argument-list format-name "format-type"
argument-list := { argument [, argument]* }
argument := state-name | user-register-name | ctype-name
state-name := a previous defined state name
user-register-name := a previous defined user register name
ctype-name := a previous defined ctype name
format-name := a previous defined display_format_name identifier
format-type := mem_ctype | reg_ctype | state | user_register
```

name is a unique identifier that can be used later to reference the *display-format*. *display-format* is a quoted string that specifies the format to be displayed. The detailed syntax of the *display-format* can be found in Xplorer's on-line Help (in Reference/User Defined Formats). *argument-list* contains a list of ctypes, states or user_registers that can be displayed in the same format. All items in *argument-list* must be of the same type specified in *format-type*. *format-type* can be one of the following identifiers: `mem_ctype`, `reg_ctype`, `state` or `user_register`, where `mem_ctype` and `reg_ctype` are the ctype representation in memory and register respectively, because the ctype memory representation can be different from the register representation. This separation allows Xplorer to correctly display a variable whether it is currently in memory or in a register.

21.3.2 Example

In the following example, ctype `xr_16x2` is a SIMD type containing two 16 bit variables. The display format informs the Xplorer to display variables of `xr_16x2` type as two 16 bit values, separated by a space. Similarly, the state and user registers are all displayed as 16-bit chunks.

```
regfile XR 32 32 xr
ctype XR 32 32 XR default
ctype xr_16x2 32 32 XR

property display_format_name fmt_16x2 "%h[31:16] %h[15:0]"
property display_format_map {xr_16x2} fmt_16x2 "mem_ctype"
property display_format_map {xr_16x2} fmt_16x2 "reg_ctype"

state st 64
property display_format_name fmt_st "%h[63:48] %h[47:32] %h[31:16]
%h[15:0]"
property display_format_map {st} fmt_st "state"

user_register st_1 st[63:32]
user_register st_0 st[31:0]
property display_format_map {st_0, st_1} fmt_16x2 "user_register"
```

21.4 Lookup Memory Property

Lookup constructs can be used to access local memories with predefined latencies. The detailed description of lookup constructs are in Chapter 14, “Lookup (`lookup`) Sections” on page 109. To simulate a processor with such lookup constructs, designers may need to write XTMP or XTSC models of the memory and run XTMP or XTSC simulation, which may be inconvenient. If the only purpose of the lookup is to access a local memory, the lookup access behavior can be simulated using a stand-alone ISS simulator by adding the lookup memory property in the TIE file.

21.4.1 Lookup Requirements

The lookup construct must satisfy the following requirements to be treated as a memory.

- The output of the lookup is a concatenation of three parts: {enable, addr, data}
  - One bit write enable to indicate whether the lookup access is a read or write operation. A value of 1 indicates the operation is a write operation, bit 0 indicates a read operation.
  - The address of the data to be stored to or loaded from. This address needs to be between 1 and 32-bits wide.
  - The data to be stored to the memory. It is ignored in a load operation.
- The input to the lookup is the data that is loaded from the memory. It has the same width as the data part of the output of the lookup.
- The output of the lookup must be in W stage.

21.4.2 Syntax

```
property-lookup-memory ::= property lookup_memory lookup-name
lookup-name ::= name of a previously defined lookup
```

21.4.3 Example

In this example lookup `lkmem` is treated as a memory. Its output width is 65 bits, composed of 1-bit write enable, 32-bit address and 32-bit data. The input width is 32 bits composed solely of the data. The output stage is Wstage (the only allowed stage). The memory takes two cycles to return the data.

The operation `lkmem_store` saves data to the memory. The input interface `lkmem_In` is not used, but must be explicitly listed in the interface list. The operation `lkmem_load` reads data from the memory. The property `lookup_memory` informs ISS that the lookup can be modeled as a memory.

```
lookup lkmem {65, Wstage} {32, Wstage + 2} //output must be W stage
operation lkmem_store {in AR a, in AR addr} {out lkmem_Out, in
                          lkmem_In} {
     assign lkmem_Out = {1'b1, addr, a}; //concatination of
                                    // write enable, address, and data
}
operation lkmem_load {out AR a, in AR addr} {out lkmem_Out, in
                          lkmem_In} {
     assign lkmem_Out = {1'b0, addr, 32'b0}; //data part is not used
     assign a = lkmem_In;
}
property lookup_memory lkmem
```

21.5 Non-exact Instruction Map Property

The instruction map (`imap`) construct is used to enable the XCC compiler to infer the use of a single TIE instruction from patterns of other TIE or Xtensa ISA instructions. Details of `imap` construct can be found at Chapter 25, “Instruction Map (`imap`) Sections” on page 203. In general, the input instructions must perform a computation that is bit-exact to the inferred output instruction. However, under certain conditions, it is desirable to create imaps that may change the behavior of the application program. An example is rounding or saturation behavior of fix point or floating point calculations.For example, a multiply-add instruction might be more accurate than the sequence of a multiply followed by an add if the multiply and add instruction each round their results while the multiple-add might do only a single round at the end of both.

While sometimes the application might need the compiler to preserve bit-exact results, in most cases the inferred result is no worse, and might even be better, than the original. Given an inexact imap, XCC will only perform this optimization by default at optimization level `-O3`. It can also be turned on or off explicitly by specifying `-menable-non-exact-imaps` and `-mno-enable-non-exact-imaps` in the XCC command line, respectively.

21.5.1 Syntax

```
property-imap-nonexact ::= property imap_nonExact imap-name
imap-name ::= name of a previously defined imap
```

21.5.2 Example 1

In this example operations myfmul, myfadd, myfmac are for floating point multiply, add, and multiply-add, respectively. In floating point computation, a multiply-add operation does not always generate the same result as a multiply followed by an add operation. But in most cases, the application developer prefers that a multiply followed by an add is replaced with a multiply-add. The `imap_nonExact` property allows the XCC compiler to perform the instruction mapping even though their input pattern and output pattern are not bit exact.

```
operation myfmul {out AR a, in AR b, in AR c} {}
operation myfadd {out AR a, in AR b, in AR c} {}
operation myfmac {inout AR a, in AR b, in AR c} {}

imap myfmap {inout AR a, in AR b, in AR c}
{ {}
  { myfmac a, b, c; }
}
{ {AR t}
```

```
  {
    myfmul t, b, c;
    myfadd a, a, t;
  }
}

property imap_nonExact myfmap
```

21.5.3 Example 2

In this example, operation sadd2 performs addition with saturation on two input variables. Operation sadd3 performs addition with saturation on three input variables. Adding three variables using sadd3 is not the same as using two sadd2 sequentially. But the former provides higher resolution. Thus, in certain situations, it is desirable to map two sadd2 operations to one sadd3 operation.

```
operation sadd2 {out AR a, in AR b, in AR c} {} {
  wire [32:0] sum = {b[31], b} + {c[31], c};
  wire is_underflow = sum[32] & ~sum[31];
  wire is_overflow = ~sum[32] & sum[31];
  wire [31:0] maxval = 32'h7fffffff;
  wire [31:0] minval = 32'hffffffff;
  assign a = is_underflow ? minval : is_overflow ? maxval : sum[31:0];
}

operation sadd3 {inout AR a, in AR b, in AR c} {} {
  wire [33:0] sum ={{2{a[31]}}, a} + {{2{b[31]}}, b} + {{2{c[31]}}, c};
  wire is_underflow = sum[33] & ~(&sum[32:31]);
  wire is_overflow = ~sum[33] & (|sum[32:31]);
  wire [31:0] maxval = 32'h7fffffff;
  wire [31:0] minval = 32'hffffffff;
  assign a = is_underflow ? minval : is_overflow ? maxval : sum[31:0];
}

imap imap_sadd {inout AR a, in AR b, in AR c}
{ {}
  { sadd3 a, b, c; }
}
{ { AR t }
  {
    sadd2 t, a, b;
    sadd2 a, t, c;
  }
}

property imap_nonExact imap_sadd
```

21.6 Semantic Sharing Property

The semantic construct described in Chapter 9, “Instruction Semantics (`semantic`) Sections” on page 61 shares hardware logic among different operations within the same slot index. Two instances of semantics are generated when the operations of the semantic are instantiated in two different slot indices. When the operations of the semantic can be issued in parallel, separate semantic logic is required. This is the case that two instances of the operations are referenced in two slot indices of the same format. However, when the two instances of the operations are from different slot indices of different formats, those two instances cannot be issued in parallel. But two copies of semantic hardware are still generated if only semantic construct is used.

The `shared_semantic` property, when used together with the `semantic` property, instructs TC to generate one instance of semantic for two or more slot indices[1].

21.6.1 Syntax

```
property-shared-semantic ::= property shared_semantic semantic-name {
slot-index [, slot-index]+ }
semantic-name ::= a previously defined semantic name
slot-index ::= a slot index that the semantic appears in
```

21.6.2 Example 1

In the following example, operation `foo` is used in both `slot1` and `Inst` slot. Slot `Inst` belongs to slot index 0 of `Inst` format, while slot `slot1` belongs to slot index 1 of `fmt` format. Because both the slot index numbers and the formats are different, the two instances of the semantic of operation foo can be shared.

```
format fmt 64 {slot0, slot1}
slot_opcodes slot0 {NOP}
slot_opcodes slot1 {foo}
slot_opcodes Inst {foo}

operation foo {out AR a, in AR b, in AR c} {} {
     assign a = b + (c << 1);
}

property shared_semantic foo {0, 1}
```

1. In release RD2010.0, the TIE compiler area estimator may not reflect the area reduction of sharing semantic instances. The synthesis area reflects the area reduction.

21.6.3 Example 2

In the following example, operation `foo` appears in slot index 0 of `Inst` format, slot index 1 of `fmt0` format, and slot index 2 of `fmt1` format. Thus, these three instances of the semantic of operation `foo` can be shared.

Operation foo also appears in slot index 3 of `fmt1` format and slot index 4 of `fmt0` format. These two instances of the semantic can also be shared.

Thus, two instances of semantics are generated when the shared_semantic property is specified. In contrast, five instances are generated when the property is not specified.

```
format fmt0 64 {slot00, slot01, slot02, slot03, slot04}
format fmt1 64 {slot10, slot11, slot12, slot13}

slot_opcodes Inst {foo}
slot_opcodes slot00 {NOP}
slot_opcodes slot01 {foo}
slot_opcodes slot02 {NOP}
slot_opcodes slot03 {NOP}
slot_opcodes slot04 {foo}
slot_opcodes slot10 {NOP}
slot_opcodes slot11 {NOP}
slot_opcodes slot12 {foo}
slot_opcodes slot13 {foo}

property shared_semantic foo {0, 1, 2}
property shared_semantic foo {3, 4}
```

21.6.4 Implementation Restrictions

Following are the guidelines for using `shared_semantic` property.

- The semantic name shall be used if a semantic is implemented for an operation. If no semantic is implemented for an operation, the operation name shall be used.
- It is illegal to share a semantic when multiple copies of it are instantiated in the same format.
- The semantics of load and store operations cannot be shared.
- Sharing semantics across slot indices may introduce muxes at the beginning of the use stage of each input and increase the number of fanouts at the end of the def stage of each output. It is the user's responsibility to assess the timing impact of sharing semantics.

21.7 Callee Saved Registers Property

Registers in a register file are either caller-saved or callee-saved. If a register is caller-saved, a callee is free to change the value of the register and a caller must assume that the register will lose its value during a call. If a register is callee saved, the callee can not use the register without first saving and later restoring it from memory. A caller is free to use it for variables that are live across a function call. By default, all TIE registers are caller saved. Therefore all registers can be freely used by a callee and any variable live across a function call must be saved to memory before the function call. The property `callee_saved` can be used to specify that a subset of the registers in a register file should instead be callee saved. The compiler will try to use these registers for variables that are live across function calls and will avoid these registers for local computation in a function.

21.7.1 Syntax

```
property-callee-saved ::= property callee_saved regfile-name number
regfile-name ::= name of a previously defined register file
number ::= number of register file entries that are reserved for callee
to save
```

21.7.2 Example

The register file is structured in such a way that the callee saved registers are always the last *number* of registers in the register file. For example, if the register file and the callee_saved property are defined as below:

```
regfile XR 16 16 xr
property callee_saved XR 5
```

The caller is responsible to save and restore registers with index from 0 to 10. The callee is responsible to save and restore registers with index from 11 to 15.

21.7.3 Implementation Restrictions

The callee saved registers have the following restrictions:

- If C application is implemented, the compiler does the analysis and automatically assigns variables to be caller saved or callee saved. If the application is written in assembly language, it is the designer's responsibility to save and restore live variables in the caller or callee.
- Register file entry with index 0 must be caller saved.

- The number of caller and callee saved registers for core register files is fixed and cannot be redefined.

21.8 Semantic Data Gating Property

The register file ports, states, and interfaces are usually connected to multiple semantics. When the data in the register file port, state, or interface change, the connected first stage logic in all semantics toggles. However, only one or a few semantics perform useful computation. The toggling in other semantics wastes energy. Please note the semantic logic in later stages do not toggle uselessly, because all flops inside semantics are clock gated.

The property **data_gate** is used by the designer to specify the semantic inputs to be data gated. Once specified, the input performs bitwise "and" operation with the semantic enable signal. Thus, the first stage logic is driven by the value zero when the semantic is not enabled, which reduces the toggling frequency.

21.8.1 Syntax

```
property-data-gate ::= property data_gate sem-name {arg-name [, arg-
name]*}
sem-name ::= name of a previously defined semantic
arg-name ::= oparg-name | opnd-name | st-name | intf-name
oparg-name ::= name of a register file operation argument that is
defined as an input to the semantic
opnd-name ::= name of a regfile operand that is defined as an input to
the semantic
st-name ::= name of a state that is defined as an input to the semantic
intf-name ::= name of an interface that is defined as an input to the
semantic
```

21.8.2 Example

The following example illustrates the usage of data gating property.

```
state st 32 add_read_write
immediate_range immed 0 60 4
operation special_load {out AR a, in AR *b, in immed imm} {in st, out
VAddr, in MemDataIn32} {
    assign VAddr = b + imm;
    assign a = MemDataIn32 + st;
}
property data_gate special_load {b, st, MemDataIn32}
```

In this example, the operation argument on AR register file `b`, the state `st`, and the interface `MemDataIn32` are all data gated. When the operation is not enabled, they all get value zero.

21.8.3 Usage Considerations

Care must be taken when selecting the inputs to data gate. If selected inappropriately, the power may not reduce.

- The `data_gate` property is only effective when the configuration sets the semantic data gating to be `user_defined`.
- The register file ports and states are all clock gated. If the register file port or the state is not used in the current cycle, its value is unchanged. Thus, if the register port or the state is only connected to one semantic, data gating the semantic input is not necessary. If multiple semantics are connected, care must be taken to select appropriate inputs to data gate.
- When evaluating the effectiveness of data gating, the power dissipated by the data gating gate needs to be taken into consideration. For details, refer to the "How to Identify Semantics for Data Gating" section in the *Xtensa Hardware User's Guide*.

21.9 Header Include Property

To use the intrinsics and ctypes defined in a TIE file, the designer needs to include a header file generated by the TIE compiler in the C program. The header file (`.h`) is given the same name as the TIE file (in case of multiple TIE files, the name is user specified) and is put in the `<path_to_TDK>/include/xtensa/tie` directory.

The `header_include` property provides a mechanism for the designer to add custom code to the header file the TIE compiler generates. It also enables the user to generate new header files and put them in the same directory. In many scenarios, it simplifies the software development process.

21.9.1 Syntax

```
property-header-include ::= property header_include header-name
"header-content"
header-name ::= name of a header file
header-content ::= content of the header file presented in a string
```

The *header-name* is the name of a header file. It can be the same name as the header file the TIE compiler generates. In this case the *header-content* specified in the property is appended to the end of the header file the TIE compiler generates.

The header file name can also be different from the header file name the TIE compiler generates. In this case, a new header file is generated and is put in the same directory.

Multiple `header_include` properties can be specified for the same header file. The order of the properties specified in the TIE file determines the order of the corresponding content in the header file.

21.9.2 Example

The following example illustrates the usage of the `header_include` property. The TIE file name is `mac2.tie`.

```
regfile XR2 32 32 xr2
operation mac2 {inout XR2 a, in XR2 b, in XR2 c} {} {
       wire [15:0] res0 = b[15:0] * c[15:0] + a[15:0];
       wire [15:0] res1 = b[31:16] * c[31:16] + a[31:16];
       assign a = {res1, res0};
}

property header_include mac2.h "#define NUM_MAC 2\n"
property header_include mac2.h "typedef XRN XR2\n"
property header_include mac2.h "#define macN mac2\n"
property header_include macn.h "#include <xtensa/tie/mac2.h>\n"
```

The TIE file describes operation `mac2` that performs two 16-bit multiply accumulations in parallel. With the usage of `header_include` property, three lines are appended to the header file `mac2.h` that the TIE compiler generates:

```
#define NUM_MAC 2
typedef XRN XR2
#define macN mac2
```

A new header file (`macn.h`) is also generated, which includes the other header file, `mac2.h`.

This provides a unified interface to the software developers. They do not need to worry about the SIMD factor. Instead, they can write the same code and use it in different processors. For example, they include the same header file: `macn.h`; they use the same type: `XRN`; and they reference the same intrinsic: `macN`. When a new processor with a SIMD factor of 4 is developed, it is easy to port the software application to the new processor.

22. Instruction Group (instruction_group) Sections

The `instruction_group` construct represents a list of instructions using one identifier, which designers can use in place of a sequence of instruction names. The instruction group names can be used in the `regbypass` construct, see Chapter 23, “Register File Bypass (regbypass) Sections” on page 195 for details. The instruction group names can also be used in the `slot_opcodes` construct to specify a group of instructions in a FLIX slot. A list of predefined instruction groups of Xtensa ISA instructions can be found in Section 11.3.1 “Instruction Groups” on page 98.

22.1 Instruction Group Syntax

```
instruction_group-def ::= instruction_group name {instr-name [, instr-name]*}
name ::= a unique name representing the instruction group
instr-name ::= an instruction name
```

name is a unique name representing the instruction group. Note that you are not allowed to define an instruction group name starting with "xt_". *instr-name* is the name of an instruction.

22.2 Example

The following is an example of an instruction group. Designers can use the identifier `vecALU` to indicate instructions `vec_add`, `vec_sub`, `vec_mul`, and `vec_neg`.

```
instruction_group vecALU {vec_add, vec_sub, vec_mul, vec_neg}
```

22.3 Restrictions

The following restriction applies to the instructions specified in an instruction group:

- Instructions specified using an `operation` construct cannot be mixed with instructions specified using `reference` and `iclass constructs`.
- Assigning a user-defined instruction group to a FLIX slot will result in the operations within the instruction group to also be encoded as part of the predefined Xtensa processor slot `Inst`.

23. Register File Bypass (regbypass) Sections

Designers can specify the `no_bypass` attribute to a register file to remove all direct bypasses of a register file, thereby adding one additional cycle to the minimum read-after-write delay. See Section 5.5 “Register File without Direct Bypass” on page 26. The register file bypass feature is only available for user-defined register files and operations using them. It cannot be used with pre-existing register files in Cadence Tensilica DSP instruction sets.

This feature is for advanced users, and should be used with extreme caution. The syntax allows a designer to add back a direct bypass from an operation's output operand to an operation's input operand. Identifying where to add direct bypass will typically require trial-and-error on the part of the designer, with very minimal tool support available to help identify candidates for direct bypass. Simply adding direct bypasses and profiling your software code is not sufficient. It may be necessary to generate the processor RTL, and run the physical implementation flow (i.e., synthesis, place & route, etc.) to ensure that the added direct bypasses do not adversely impact frequency. Adding back direct bypasses may reduce overall frequency compared to a design with no direct bypass.

23.1 Register File Bypass Syntax

```
regbypass-def ::= regbypass regfile-name src-arg dst-arg
regfile-name ::= name of a previously defined register file
src-arg ::= opnd-arg-name
dst-arg ::= opnd-arg-name
opnd-arg-name ::= [instruction-name | instruction_group-name],
[operand-name | arg-name]
instruction-name ::= name of a previously defined instruction
instruction_group-name ::= name of a previously defined instruction
group
operand-name ::= name of a previously defined operand
arg-name ::= the argument name of the operation specified in the
operation or instruction group composed of operations
```

regfile-name is the name of a register file. *src-arg* and *dst-arg* have the same structure. *src-arg* specifies the argument that produces the data, and *dst-arg* specifies the argument that consumes the data. The argument is a tuple. The first element is either an instruction name or an instruction group name. See Chapter 22, “Instruction Group (instruction_group) Sections” on page 193 for instruction group details. The second element is either an operand name or an operation argument name.

The `regbypass` specifies a direct bypass from the `src-arg` to the `dst-arg`. The entire set of specified direct bypasses can be found in the report file the TIE compiler generates.

23.2 Examples

In the following example, the register file XR does not contain any direct bypass, except the path from an output argument a of `foo` to the input argument b of `foo`. The data produced by argument a is available to b in the immediate next cycle. Note there is no direct bypass from argument a of `foo` to argument c of `foo`. Thus, the data produced by a is only available to c two cycles later.

```
regfile XR 32 16 xr no_bypass
operation foo {out XR a, in XR b, in XR c} {} {
          assign a = b + c;
}

regbypass XR {foo, a} {foo, b}
```

In the following example, the instruction group `alu_xr` is used to specify the direct bypass. The direct bypass is from argument a of instructions `add_xr`, `sub_xr` and `neg_xr` to the argument c of the same set of instructions.

```
regfile XR 32 16 xr no_bypass
operation add_xr {out XR a, in XR b, in XR c} {} {
        assign a = b + c;
}

operation sub_xr {out XR a, in XR b, in XR c} {} {
        assign a = b - c;
}

operation neg_xr {out XR a, in XR c} {} {
        assign a = ~c;
}

semantic alu_xr {add_xr, sub_xr, neg_xr} {
        wire [31:0] add0 = (add_xr || sub_xr) ? b : 32'b0;
        wire [31:0] add1 = add_xr ? c : ~c;
        wire carry = sub_xr;
        assign a = TIEadd(add0, add1, carry);
}

instruction_group alu_xr {add_xr, sub_xr, neg_xr}
regbypass XR {alu_xr, a} {alu_xr, c}
```

23.3 Implicitly Specified Bypasses

When a direct bypass is specified for an operation's input argument, input arguments of other operations also benefit from the direct bypass if they are mapped to the same register file read port and read the register file in the same pipeline stage. The instructions and the corresponding operation input arguments that can bypass data without explicitly specifying them are called *free riders*. Users can use the `regport` construct to control the operation arguments that are implicitly bypassed. See Chapter 24, "Regfile Port (regport) Sections" on page 199 for details.

Note that the same is not true for operation arguments sharing the same write ports. The designers need to explicitly specify the expected direct bypass for the output operation arguments.

Because those direct bypasses are implicit, the C compiler (XCC) may or may not know the existence of those bypasses. The C compiler understands the implicit direct bypass from a source operation argument to a destination operation argument if in all slot indices the source operation and destination operation exist, the bypasses from the source operation argument to the destination operation argument are all implicitly specified. If the C compiler does not know an implicit direct bypass, it schedules the instructions as if there is no direct bypass.

The following is an example illustrating the implicitly specified bypasses:

```
regfile XR 32 16 xr no_bypass


operation add_xr {out XR a, in XR b, in XR c} {} {
        assign a = b + c;
}

operation sub_xr {out XR a, in XR b, in XR c} {} {
        assign a = b - c;
}

regbypass XR {add_xr, a} {add_xr, b}

regport XR_rd0 XR in {add_xr, b, 0} {sub_xr, b, 0}
```

A direct bypass from the output argument `a` of operation `add_xr` to the input argument `b` of the same operation is specified. The argument `b` of `add_xr` and the argument `b` of `sub_xr` are mapped to the same stage of the same port. Thus, the argument `b` of `sub_xr` also gets the bypass from the output argument `a` of `add_xr`.

23.4 Bypasses on Register Groups

The `no_bypass` attribute specified on a register file also removes the direct bypasses for the register groups that are implemented on the register file. Users can add back selected direct bypasses for the register groups the same way as for the register files.

If a direct bypass is specified for a register group implemented using wide ports, all input operands whose widths are of equal or less size and that are mapped to the same read port and read port stage as the direct bypass can be implicitly bypassed. If an operand's width is wider than the width of the specified bypass, that operand is not implicitly bypassed.

If a direct bypass is specified for a register group implemented using multiple ports, all input operands mapped to the same read port and the same read port stage can be implicitly bypassed. An input operand specified for a register group can only be bypassed implicitly if bypasses are explicitly specified from a single source to all of the operand's mapped read ports.

23.5 Implementation Restrictions

The following implementation restrictions apply to direct bypass:

- Suppose a direct bypass from an output argument of a source operation to an input argument of a destination operation is specified. If the source operation or the destination operation exists in multiple slot indices, the direct bypasses exist from the source operation in any slot index to the destination operation in any slot index.
- When an instruction group is used to specify a group of instructions in a direct bypass, the instructions in the group must follow the following constraints:
  - All instructions in the group must have the same operation argument or operand that is specified in the direct bypass.
  - The operation argument or operand in all instructions of the group must have the same read or write stage.
  - All instructions in the group must be defined in the same slot indices.

24. Regfile Port (regport) Sections

The `regport` construct can be used to manually assign operands or operation arguments to the register file ports. The TIE compiler usually performs such an optimization. However, in some cases designers may want to specify the operands or the operation arguments that share the same port. For example, you may group the operands that expect to have the same bypasses to one port to reduce the number of direct bypasses of a register file.

24.1 Regfile Port Syntax

```
regport-def ::= regport port-name regfile-name dir [opnd-arg]+
port-name ::= a unique identifier
regfile-name ::= name of a previously defined register file
dir ::= in | out
opnd-arg ::= {opnd-arg-name, slot-index [, port-index]}
opnd-arg-name ::= operand-name | operation-name, arg-name
slot-index :: = an integer specifying the slot index
port-index :: = an integer specifying the port index
operand-name ::= name of a previously defined operand
operation-name :: name of a previously defined operation
arg-name ::= the argument name of the operation specified in operation-
name
```

port-name is a unique identifier to indicate the name of a register file port. *regfile-name* is the name of a register file. *dir* indicates the direction of the port on the register file. **in** means the port is a read port, and **out** means the port is a write port. A list of arguments specifying the operands or operation arguments follows.

One argument can be either an `operand`, specified by the *operand-name*, or an operation argument, specified by a tuple composed of *operation-name* and *arg-name.* Because one instruction may exist in multiple slot indices, the corresponding operand or the operation argument may appear in multiple slot indices. If multiple slots in a format contain the same operation argument or operand, they belong to a different slot index. Since they are used in the same cycle, they cannot be allocated to the same port. Thus, *slot-index* specifies the slot index number that the operand or operation argument is in. Note that if an operand or operation argument in different slot indices only belong to different formats, they can still be mapped to the same port. Because the instruction in all slots of the same slot index share hardware, their operand or operation arguments are always allocated to the same port.

port-index only refers to the port implementation of a register group. See Section 5.4 “Register Group” on page 23 for register group details. If a register group is implemented as multiple ports, one operation argument or operand can be mapped to multiple register file ports. port-index specifies the mapping between a port and a portion of an operation argument or an operand. The mapping can be arbitrary, but needs to be consistent across all mappings in a register group. This argument is optional. If not specified, a value of zero is assigned by the TIE compiler.

24.2 Examples

In the following example, the regport construct maps the argument z of ADD_XR in slot index 1, the argument z of ADD_XR in slot index 0, and the argument b of MOVE_XR in slot index 0 to the same read port. It also maps the argument a of MOVE_XR and the argument x of ADD_XR in both slot indices to the same write port. This mapping can be verified by examining the report file generated by the TIE compiler:

```
regfile XR 32 16 xr no_bypass
operation MOVE_XR {out XR a, in XR b} {} {
     assign a = b;
}

operation ADD_XR {out XR x, in XR y, in XR z} {} {
     assign x = y + z;
}

format fmt 64 {slot0, slot1}
slot_opcodes Inst {MOVE_XR}
slot_opcodes Inst {ADD_XR}
slot_opcodes slot0 {NOP}
slot_opcodes slot1 {ADD_XR}

regport myrd0 XR in {MOVE_XR, b, 0} {ADD_XR, z, 0} {ADD_XR, z, 1}
regport mywr0 XR out {MOVE_XR, a, 0} {ADD_XR, x, 0} {ADD_XR, x, 1}
```

The regport construct is useful when used in combination with the regbypass construct. Details of the regbypass construct can be found in Chapter 23, “Register File Bypass (regbypass) Sections” on page 195.

In the above example, users can specify one direct bypass from the output x of ADD_XR and a of MOVE_XR to input argument z of ADD_XR and b of MOVE_XR, shown as follows:

```
regbypass XR {MOVE_XR, a} {MOVE_XR, b}
regbypass XR {ADD_XR, x} {ADD_XR, z}
```

The specified `regport` construct ensures that only one direct bypass route is generated by the TIE compiler. It also controls the bypasses that are implicitly specified.

In the following example, the `regport` construct maps argument `b` of `ADD_XR2` in slot index 0 and argument `c` of `ADD_XR2` in slot index 1 to the same port. Note that `XR2` is a register group, with each entry composed of two `XR` entries. With the port implementation, two `XR` ports are required to compose each `XR2` port. The `regport` is specified twice, once for port index 0 and once for port index 1.

```
regfile XR 32 16 xr XR2=2 port
ctype XR 32 32 XR
ctype XR2 64 64 XR2 {XR a, XR b}

immediate_range immed 0 56 8
operation ld.XR2 {out XR2 a, in XR * addr, in immed imm} {out VAddr, in
MemDataIn64} {
        assign VAddr = addr + imm;
        assign a = MemDataIn64;
}
operation st.XR2 {in XR2 a, in XR * addr, in immed imm} {out VAddr, out
MemDataOut64} {
        assign VAddr = addr + imm;
        assign MemDataOut64 = a;
}
operation mv.XR2 {out XR2 a, in XR2 b} {} {
        assign a = b;
}
proto XR2_loadi {out XR2 a, in XR2 *b, in immediate imm} {} {
        ld.XR2 a, b, imm;
}
proto XR2_storei {in XR2 a, in XR2 *b, in immediate imm} {} {
        st.XR2 a, b, imm;
}
proto XR2_move {out XR2 a, in XR2 b} {} {
        mv.XR2 a, b;
}
operation ADD_XR2 {out XR2 a, in XR2 b, in XR2 c} {} {
        assign a = b + c;
}

format fmt 64 {slot0, slot1}
slot_opcodes Inst {ADD_XR2}
slot_opcodes slot0 {NOP}
slot_opcodes slot1 {ADD_XR2}

regport xr2_rd0 XR2 in {ADD_XR2, b, 0, 0} {ADD_XR2, c, 1, 0}
regport xr2_rd1 XR2 in {ADD_XR2, b, 0, 1} {ADD_XR2, c, 1, 1}
```

24.3 Features and Restrictions

The following are the features and restrictions for the `regport` construct:

- The `regport` construct with the same port name can be specified multiple times. Their operation arguments or operands are all mapped to the same port.
- Specifying different port names on the `regport` construct does not necessarily mean the specified operation arguments or operands are mapped to two unique ports. They can still be mapped to the same port if the TIE compiler can manage to do so.
- Multiple operations may share one semantic, and, thus multiple operation arguments may share one operand. In such cases, only one operation argument is needed to be specified in the `regport` construct. All other operation arguments are implicitly mapped to the same port.

25. Instruction Map (imap) Sections

Instruction map (imap) sections are used in TIE descriptions to enable the XCC compiler to infer the use of TIE instructions from patterns of other TIE or Xtensa ISA instructions. The XCC compiler uses the imap information to replace one sequence of instructions with another, so as to achieve better cycle count performance or better code density. A common case of this optimization is the use of a single fusion instruction that performs the same computation as a sequence of other instructions.

25.1 imap Syntax

```
imap-spec ::= imap name {arg-list} {output-pattern} {input-pattern}
name ::= a unique identifier
arg-list ::= imap-arg [, imap-arg]*
output-pattern ::= {[temp-arg-list]} {code-list}
input-pattern ::= {[temp-arg-list]} {code-list}
imap-arg ::= dir arg-type arg-name
dir ::= in | out | inout
arg-type ::= regfile-name | immediate
regfile-name ::= a previously defined regfile name
arg-name ::= an identifier unique to other arg-names in this imap
temp-arg-list ::= temp-arg [, temp-arg]*
temp-arg ::= regfile-name arg-name
code-list ::= code; [code;]*
code ::= inst | assignment
inst ::= instruction [argument] [, argument]*
instruction ::= a previously defined instruction
argument ::= arg-name | constant
constant ::= an integer constant | sized integer constant
assignment ::= arg-name = argument
```

name is a unique identifier for the `imap` construct. *arg-list* lists the register and immediate operands of the instruction sequence corresponding to the *output-pattern* and *input-pattern*. *arg-names* are "matched" between the *output-pattern* and the *input-pattern*, in the sense that the same *arg-name* implies the same operand. *code-list* lists a sequence of previously defined instructions and their arguments. This sequence may optionally contain assignment statements, in which the value of a register argument or a constant is assigned to another register argument. *temp-arg-list* lists temporary arguments of the code sequence. Temporary arguments are defined as those used in the *code-list* but not listed in the *arg-list* of the `imap` construct.

25.2 imap Usage

The primary purpose of the `imap` construct is to enable the XCC compiler to infer the use of TIE instructions. It implies that the instruction sequence specified in the *input-pattern* section should be replaced by the instruction sequence specified in the *output-pattern* section. In the current release of the TIE compiler, the *output-pattern* section is restricted to having one and only one instruction. Thus the instruction specified in the *output-pattern* section can be viewed as the fusion of the instructions specified in the *input-pattern* section.

When all the instructions in the *input-pattern* are Xtensa ISA instructions, the `imap` construct allows XCC to infer the TIE instruction in the *output-pattern* without the use of intrinsics in the C code. If one or more instructions in the *input-pattern* are designer defined TIE instructions, you will need to use the intrinsic for those instructions in the C code, or define `operator` constructs on those intrinsics. The XCC compiler will still be able to replace the *input-pattern* sequence with the instruction of the *output-pattern*, without the explicit use of the *output-pattern* instruction as an intrinsic.

The `imap` construct only indicates that a sequence of instructions can be replaced by another instruction. It does not define the implementation or the semantic description of any of the instructions. You are required to separately provide this for all non-Xtensa ISA instructions using the `operation`, `reference`, or `semantic` construct.

The XCC compiler searches for matches as an intermediate step of compiling an application. In order to successfully match a pattern, the pattern must be identical to the sequence of instructions generated by the compiler at the phase in the compilation process where matching occurs. It is not guaranteed that the pattens identified by visually inspecting the C/C++ code or assembly code would be preserved in the intermediate step. Thus, the *input-pattern* may not always be replaced by the *output-pattern*.

A possible problem in writing `imap` sections manually is ensuring that the *input-pattern* can be replaced by the *output-pattern* under all conditions. If this is not true, you may get incorrect compiled code. Neither the TIE compiler nor the XCC compiler can verify that the *output-pattern* is the correct and preferable replacement for the *input-pattern.* If you

manually define one or more imap sections in your TIE design, it is your responsibility to verify this property. If the *output-pattern* is not a bit-exact match of the *input-pattern*, but the *output-pattern* is a more preferred implementation, it is still possible to write the imap construct to suggest such replacements. You need to specify the imap_nonExact property described in Section 21.5 “Non-exact Instruction Map Property” on page 185. Depending on the optimization level, the XCC compiler may or may not perform such replacements.

25.3 Examples

The Xtensa ISA provides an 8-bit unsigned load instruction that zero-extends an 8-bit value from memory to 32 bits. There is no corresponding signed load instruction that would sign-extend the 8-bit value. Such an instruction can easily be created in TIE, and with an appropriate imap definition, will be inferred by the XCC compiler. The following TIE code illustrates the L8SI instruction that performs an 8-bit load, sign-extends the value to 32-bits and returns the result in an AR register.

```
immediate_range IMR 0 255 1
operation L8SI {out AR reg, in AR base_addr, in IMR offset}
               {out VAddr, in MemDataIn8} {
     assign VAddr = base_addr + offset;
     assign reg = {{24{MemDataIn8[7]}}, MemDataIn8};
}

imap l8si { out AR reg, in AR base_addr, in immediate offset}
{ { }
  {
     L8SI reg, base_addr, offset;
  }
}
{ { AR t0 }
  {
    L8UI t0, base_addr, offset;
    SEXT reg, t0, 32'h7;
  }
}
```

The imap description indicates that the designer defined L8SI instruction is equivalent to the Xtensa ISA L8UI instruction followed by a restricted use of the Xtensa ISA SEXT instruction. The SEXT instruction is used with a constant value of 32’h7 instead of an immediate operand. The L8UI instruction performs an 8-bit unsigned load, with its result assigned to the temporary argument t0. The SEXT instruction operates on t0, and sign extends the 8-bit value to 32-bits. This sequence is typically used by the XCC compiler to implement an 8-bit signed load. With the above TIE description, the XCC compiler will use the designer defined L8SI instruction instead of the L8UI, SEXT sequence. Without

the imap section, L8SI would still be a valid instruction that can be used as an intrinsic in your C/C++ code. But with the imap section, the XCC compiler understands its functionality and can automatically infer it from regular C code.

Note that it is not enough to just provide the imap section. The operation description of the L8SI instruction is also necessary. Furthermore, it is your responsibility to ensure that the operation description of L8SI is indeed equivalent to the L8UI, SEXT sequence.

The following TIE code provides another example, in which it is necessary to use an assignment statement in addition to instructions as part of the *input-pattern*.

```
operation fusion.MINU.SRLI {out AR r1, in AR r2} {} {
     wire [31:0] t2 = r2 > 32'h10000000 ? 32'h10000000 : r2;
     assign r1 = {1'b0, t2[31:1]};
}

imap fusion.MINU.SRLI {out AR r1, in AR r2}
{ {}
  {
    fusion.MINU.SRLI r1, r2;
  }
}
{ {AR t1, AR t2}
  {
    t1 = 32'h10000000;
    MINU t2, r2, t1;
    SRLI r1, t2, 1;
  }
}
```

The fusion.MINU.SRLI operation performs a computation equivalent to the sequence of the Xtensa ISA MINU (Unsigned Minimum) and SRLI (shift right logical with immediate) instructions. The MINU instruction takes all three operands from the AR register file, while the fusion instruction uses a constant (32'h10000000) as one of its operands. The *input-pattern* must use the MINU instruction with three AR register operands, so the constant value is first assigned to a temporary argument t1, which is then used as an operand of MINU.

Assignment statements are used to assign constants to register arguments, or to assign the value of one register argument to another register argument of the same type.

Note that while it is not possible to use a constant in place of a register argument, a constant can be used in place of an immediate argument. This was illustrated in the first example, where a constant value was used with the SEXT instruction.

25.4 Implementation Restrictions

- The *output-pattern* section of the `imap` definition must contain one and only one instruction. This also implies that the *temp-arg-list* of the *output-pattern* will always be empty.
- If the *input-pattern* section of the `imap` definition contains one instruction, the number of register operands of the instruction must be greater than the number of register operands of the instruction in the *output-pattern*. When calculating the number of register operands, operands with direction `in` or `out` are counted as one, while operands with direction `inout` are counted as two.
- If an immediate operand is used in the *input-pattern*, it may only be used with one instruction in the sequence that makes up the *input-pattern*.

The syntax of the `imap` construct in the current release may seem more verbose than necessary. This is because the syntax is designed to allow the *output-pattern* section to specify a sequence of instructions in a future release. The chosen syntax of `imap` allows this future enhancement to be made in a backward compatible manner.

26. Operator (operator) Sections

Programmers developing applications on an Xtensa platform can call TIE instrinsics in the C/C++ application code. This process, however, may reduce the readability of the application code. In many places, the intrinsic applies an operation (such as add, subtract) on user-defined ctypes. It is sometimes more intuitive to represent the operations as C/C++ operator symbols (such as "+", "-"). The `operator` construct maps C/C++ built-in operator symbols to the corresponding proto. With `operator` construct, programmers can treat user-defined ctypes the same as C built-in types, and thus improve the program's readability.

26.1 Operator Syntax

```
operator-def ::= operator "symbol" proto-name
symbol ::= a symbol defined in Table 26–15
proto-name ::= name of an already defined proto or TIE operation
```

symbol is a predefined C/C++ operator symbol. A complete set of supported symbols is defined in Table 26–15. *proto-name* is the name of an already defined proto that perform the computation defined in *symbol*. Note that for every operation in the TIE file, TIE compiler automatically generates a proto with the same name as the operation, thus the operation name can be used.

26.2 Supported Operators

The following operators are supported:

Table 26–15. Supported Operators

Operator	Intended Operation	Input Arguments
+	Add	2
-	Subtract	2
*	Multiply	2
/	Divide	2
%	Modulo	2
==	Equal	2
!=	Not equal	2
<	Less than	2
<=	Less than or equal	2

Table 26–15. Supported Operators

Operator	Intended Operation	Input Arguments
>	Greater than	2
>=	Greater than or equal	2
&	Bitwise and	2
|	Bitwise or	2
^	Bitwise ex-or	2
<<	Left shift	2
>>	Right shift	2
!	Logical not	1
-	Negate	1
~	Binary negate	1

By specifying the operators in Table 26–15, the following operators are derived and can be used in the C application:

```
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
```

Note these operators requires that the first input argument and the output argument of the proto are of the same ctype.

All operators have the same precedence as the corresponding C/C++ operators.

Some operators (such as /, %) require complex computation and often require control flow that can not be implemented using protos. Such operators can be overloaded by implementing a C function with stylized function names. Refer to the *Xtensa C and C++ Compiler User's Guide* for details.

26.3 Example

In this example, register file XR is created as well as the C datatype XR associated with it. The addition prototype (`MY_ADD`) is implemented. The operator maps symbol "+" to the addition prototype (`MY_ADD`).

The TIE file is:

```
regfile XR 32 16 xr
operation MY_ADD {out XR res, in XR in0, in XR in1} {} {
     assign res = in0 + in1;
}
proto MY_ADD {out XR res, in XR in0, in XR in1} {} {
     MY_ADD res, in0, in1;
```

```
}
operator "+" MY_ADD
```

In the C file, software designers can use "+" to perform an addition computation on two XR C datatypes. The XCC compiler automatically converts "+" into the proto `MY_ADD` based on the argument type and the `operator` construct in the TIE file.

```
XR a[128], b[128], c[128]
/* code to assign XR array a, b meaningful values */

for (i = 0; i < 128; i++) {
     c[i] = a[i] + b[i]; /* use "+" perform addition on user type XR */
}
```

26.4 Implementation Restrictions

Following are the implementation restrictions for the *`operator`* construct:

- All protos that are mapped to operators can have exactly one output argument. The allowed number of inputs is listed in Table 26–15. For operators with two inputs, the first input argument of the proto corresponds to the value on the left side of the operator; the second input argument of the proto corresponds to the value on the right side of the operator.
- TIE compiler (TC) does not verify the operators are used as intended. Programmers can assign the operators meanings distinct from the common meanings of the operators. However, we encourage the programmers to maintain a list of intended meaning of the operators. It is discouraged, for example, to assign operator "+" with meaning subtract.
- No implicit type conversion is allowed in any of the operator arguments. The input arguments passed to the operator need to be of the same type as the input arguments of the corresponding proto.
- The protos mapped to operators must have at least one user -defined ctype on one input argument.
- The operator overloading for the same operator can be defined multiple times with different protos. The XCC compiler matches protos associated with a given operator by the types of their input arguments. Therefore, to avoid ambiguity, if there are multiple protos associated with the same operator, there should be no two protos with exactly the same input argument types.
- The XCC compiler does not assume commutativity, associativity, and other standard properties on the overloaded operators.

27. Cstub Swap (cstub_swap) Sections

The cstub_swap construct is only used for cstub in native simulation. If the TIE instructions load or store multiple data values in one memory access, the TIE code may be endian dependent. That is, different results may be obtained on little-endian and big-endian processors. The cstub_swap construct provides a way to simulate applications for big-endian Xtensa processors on little-endian x86 host machines by specifying the swapping mechanism of MemDataIn<*n*> or MemDataOut<*n*> interfaces. Detailed information on the usage of cstub_swap construct can be found in the Simulating Applications for Big-Endian Xtensa Configurations section in the *Tensilica Instruction Extension (TIE) Language User's Guide*.

27.1 cstub_swap Syntax

```
cstub_swap-def ::= cstub_swap name opcode-list interface-concat
name ::= a unique identifier
opcode-list ::= { opcode-name [, opcode-name]* }
opcode-name ::= a previously defined opcode or operation name
interface-concat ::= { intf-slice [, intf-slice]* }
intf-slice ::= MemDataIn<n>[p:q] | MemDataOut<n>[p:q]
```

name is a unique symbolic name for the cstub_swap construct. *opcode-name* is a previously defined opcode name. If an instruction is defined using the operation construct, *opcode-name* is the same as the operation name. *interface-concat* is a concatenation of MemDataIn<*n*> or MemDataOut<*n*> interface. Only one interface can appear in the concatenation. That interface should be the one appearing in the instructions of *opcode-list*. Every bit of the interface should be included exactly once in the concatenation.

27.2 Example

The following example illustrates how to use the cstub_swap construct.

```
state st0 8 add_read_write
state st1 8 add_read_write
state st2 8 add_read_write
state st3 8 add_read_write

/* 31   24 23   16 15   8 7   0  <-- Bits of MemDataIn32
     st0     st1     st2   st3   <-- Loaded data */
operation simd_load {in AR *addr} {out VAddr, in MemDataIn32,
                                   out st0, out st1, out st2, out st3} {
```

```
        assign VAddr = addr;
        assign {st0, st1, st2, st3} = MemDataIn32;
}
```

Suppose the values read by the instruction `simd_load` are saved as four characters. The instruction is endian dependent. That is, the values in `st0 - st3` depend on the endianness of the processor configuration. When the application runs on a little-endian processor, no special care needs to be taken in native simulation. When the application runs on a big-endian processor, to achieve the same result in native simulation as in ISS, the following `cstub_swap` construct is provided:

```
cstub_swap myswap {simd_load} {MemDataIn32[ 7: 0],  /* st3 */
                               MemDataIn32[15: 8],  /* st2 */
                               MemDataIn32[23:16],  /* st1 */
                               MemDataIn32[31:24]   /* st0 */
                              }
```

With the `cstub_swap` construct, `myswap`, the `MemDataIn32` appearing in the operation body is the byte-swapped version of the `MemDataIn32` coming from the memory. The detailed steps in deriving the swapping mechanism can be found in the Simulating Applications for Big-Endian Xtensa Configurations section in the *Tensilica Instruction Extension (TIE) Language User's Guide*.

27.3 Implementation Restrictions

The `cstub_swap` construct is only used in simulating applications for big-endian processors in the cstub environment. It is ignored on little-endian processors. It has no impact on hardware or ISS. The following restrictions apply for `cstub_swap` constructs:

- Each instruction can only have one swapping mechanism and the swapping mechanism can be determined statically.
- If the `LoadByteDisable` or `StoreByteDisable` interface appears in an instruction, it can only disable reading or writing entire interface slices of the corresponding `cstub_swap` construct.
- On "Align address" configurations, the value of `VAddr` of all instructions that have a `cstub_swap` construct must be aligned to the `MemDataIn`<*n*> or `MemDataOut`<*n*> interface width.

28. Computation Sections

Computations are not TIE description sections in themselves, but they are used to specify computations in `operation` sections (see Chapter 7, “Instruction Operation (`operation`) Sections” on page 37), `semantic` sections (see Chapter 9, “Instruction Semantics (`semantic`) Sections” on page 61), `reference` sections (see Chapter 40, “Instruction Reference (`reference`) Sections” on page 281), and `operand` sections (see Section 34 “Instruction Operand (`operand`) Sections” on page 261). During hardware generation, computation descriptions are translated into synthesizable HDL descriptions. The statements are also translated into various C programs, to be used by the Instruction Set Simulator, the assembler and disassembler, and the compiler.

28.1 Computation Syntax

```
computation ::= statement [statement]*
statement ::= (wires | assignment)
wires ::= see Section 28.2.1
assignment ::= see Section 28.2.2
```

28.2 Statements

All computations consist of one or more statements. There are only two kinds of statements: `wires` and `assignment`.

28.2.1 Wires Statement

Wires are nets. They represent physical connections to circuit structures, such as operators (Section 28.3.4), which are the equivalent of gates in a circuit. Just as in real circuits, wires have values continuously driven by the devices to which they are connected. After being defined in a `wire` statement, a wire can be used in an assignment statement. Alternately, a wire can be defined as part of an assignment statement.

In specifying computations, wires are typically used as temporary variables. But unlike variables in programming languages, wires cannot be assigned multiple times. Assigning a wire multiple times is the equivalent of driving the wire with multiple gates, resulting in an undefined value.

Wires Syntax

Wires can represent single bits or multiple bits (vectors). Wire vectors are declared as an indexed range to represent multi-bit buses:

```
wires ::= wire [[from:to]] name1 [, name2]*;
```

name is a unique identifier for each wire declared in the statement. The size of a wire vector is specified as an index range, *from* the left-most index *to* the right-most index. *from* must be greater than or equal to *to*. The most-significant bit (msb) index is always the *from* index. Wire indices *from* and *to* can be negative integers. For single bit wires, the index range need not be specified.

Thus, a single wire uses the syntax:

```
wire name1 [, name2]*;
```

And a wire vector uses the syntax:

```
wire [from:to] name1 [, name2]*;
```

`name1 [, name2]*` indicates that multiple wires or wire vectors can be declared using a single wire statement. Thus, the statement:

```
wire [15:0] name1, name2, name3, name4;
```

is equivalent to the following four statements:

```
wire [15:0] name1;
wire [15:0] name2;
wire [15:0] name3;
wire [15:0] name4;
```

Examples

```
wire tmp;         // A single wire named “tmp”
wire [31:0] addr; // A 32-bit bus. The msb is addr[31].
wire [15:0] name1, name2, name3, name4; // multiple vectors
```

28.2.2 Assignment Statement

Assignment statements are the basic statements for describing computational data dependencies.

Assignment Syntax

```
assignment ::= assign lhs = expression;
lhs ::= (variable-concat | variable)
variable-concat ::= {variable [, variable]*}
variable ::= an out or inout type operand of the instruction or a
             previously defined wire
expression ::= see Section 28.3
```

The Left Hand Side (*lhs*) of an assignment statement can be any instruction operand of type `out` or `inout` or wires as defined in Section 28.2.1. It can also be a concatenation of operands and wires (see Section 28.3.4 for the concatenation syntax). *expression* corresponds to the result of an expression. The assignment is continuous, such that the nets assume the new values as soon as they are available. Another way to think of the assignment statement is to drive the *lhs* by the *expression* gate.

Examples

```
operation ASSGN {out AR res, in AR in0, in AR in1} {} {
     wire [31:0] w0, w1, w2;
     assign w0 = in0;
     assign w1 = in1;
     assign w2 = w0 + w1;
     assign res = w2;
}

state S1 16 add_read_write
state S2 16 add_read_write
operation COPY {inout AR a} {inout S1, inout S2} {
     assign {S1, S2} = a;
     assign a = S1 + S2;
}
```

28.2.3 Combining Wire Declaration and Assignment Statements

Wires are declared using the `wire` statement and assigned a value using the `assign` statement. You are also allowed to combine the declaration of a `wire` and an assignment to it in a single statement. For example, instead of:

```
wire [31:0] t;
assign t = s + b;
```

you can write the above as a single statement such as:

```
wire [31:0] t = s + b;
```

28.3 Expressions

An expression combines operators and operands to produce a result, which may be assigned to a variable. The operators are described in Section 28.3.4. Expression operands can include previously defined operands, constants, and previously defined wires (or wire vectors), and expressions can be recursive, including other expressions as operands.

The following rules apply to the evaluation of expressions:

- The TIE compiler does not perform implicit type conversions on operands prior to the computations.
- Operands are always zero-extended to the appropriate operand size before the computations. See Section 28.3.3 on page 219 for details on how an operand size is determined.
- Computations are always unsigned.
- It is the designer's responsibility to use two's complement representation to achieve the desired results on signed numbers.

28.3.1 Expression Syntax

```
expression ::= [operand-ex] operator operand-ex [operand-ex]
operand-ex ::= (expression | operand | constant | wire)
operand ::= a previously defined wire (see Chapter 34)
constant ::= see Section 28.3.3
operator ::= see Section 28.3.4
wire ::= a previously defined wire (see Section 28.2.1)
```

For details on the bit-width of an expression result, see Section 28.3.4 "Operators" on page 219.

28.3.2 Examples

```
r << (s & 32'hf)    // left-shift r by expression
(a || b) && c       // an expression with logical operators
(x & y) ? 1 : 0     // a conditional expression
```

The following pages contain additional examples of expressions.

28.3.3 Constants

Syntax

```
constant ::= (integer | sized-constant)
sized-constant ::= size-base-value
integer ::= integer constant
```

There are two kinds of constants: *integer* and *sized-constant*. Integer constants, when used in a context that requires knowing the bit-width of the constant, are 32 bits. Sized-constants can be specified in binary, octal, decimal, or hexadecimal formats by specifying the *size*, `base`, and *value*. *size* is the exact decimal number of digits in the constant. *base* starts with a single quote (‘) followed by a single character (“b” for binary, “o” for octal, “d” for decimal, or “h” for hexadecimal). *value* specifies the value of the constant, using characters consistent with the base specifier (binary values can be 0, 1, or `x`; octal values can be 0-7 or `x`; decimal values can be 0-9; and hexadecimal values can be 0-9, a-f, or `x`).

“x” means “don’t-care”. It is not possible to specify “don’t care” value-items in decimal format. All characters used in specifying constants are case-insensitive. An error results if the number of items in *value* is greater than *size*. If the number of items in *value* is less than *size*, the value still has *size* items and the missing high-order items are assumed to be zero (0).

Examples

```
1024           // integer
8'b111x        // same as 8'b0000111x
9'o77          // same as 9'b000111111
16'h7ffx       // same as 16'b011111111111xxxx
```

28.3.4 Operators

The TIE operators are similar to those used in the C programming language and the Verilog hardware description language. Table 28–16 summarizes the operators, by type. The pages immediately following the table describe their functions.

Table 28–16. Operators

Operator Type	Operator Symbol	Operation	Number of Operands
Arithmetic	+	Add	Two
	-	Subtract	Two
	*	Multiply	Two
Logical	!	Logical negation	One
	&&	Logical and	Two
	||	Logical or	Two
Relational	>	Greater than	Two
	<	Less than	Two
	>=	Greater than or equal	Two
	<=	Less than or equal	Two
	==	Equal	Two
	!=	Not equal	Two
Bitwise	~	Bitwise negation	One
	&	Bitwise and	Two
	|	Bitwise or	Two
	^	Bitwise ex-or	Two
	^~ or ~^	Bitwise ex-nor	Two
Reduction	&	Reduction and	One
	~&	Reduction nand	One
	|	Reduction or	One
	~|	Reduction nor	One
	^	Reduction ex-or	One
	^~ or ~^	Reduction ex-nor	One
Shift	<<	Left shift	Two
	>>	Right shift	Two
Extraction	[]	Bit extract or index	Two or Three
Concatenation	{ }	Concatenation	Any
Replication	{ { } } or { }	Replication	Any
Conditional	?:	Conditional	Three
built-in modules	<module-name>(...)	As defined in Chapter 29	As defined in Chapter 29
functions	<function-name>(...)	Designer-defined	Designer-defined

Arithmetic Operators

The binary arithmetic operators are add (+), subtract (-), and multiply (*). All arithmetic operators are unsigned. Signed multiply is not available. The designer is responsible for any required operand conversions, which must be done before the computations. Operands of different sizes are always zero-extended to the largest operand size before the computations.

Unary minus (-) operator is not supported. 1'b0-A is equivalent to -A.

Examples

```
(a + b) * c     // an expression with arithmetic operators
1'b0 - a // negate a, equivalent to -a
```

Relational Operators

The relational operators are greater than (>), greater than or equal (>=), less than (<), less than or equal (<=), equal (==), and not equal (!=). Operands are always zero-extended to the largest operand size before the comparison. The comparisons are always done with unsigned values. The relational operator always produces a 1-bit result, with 1 as true and 0 as false.

Example

```
a == b         // an expression with relational operator
```

Logical Operators

The logical operators are negation (!), AND (&&), and OR (||). An operand is logical 0 if all bits are 0's, and is logical 1 otherwise. The result of a logical expression is always 1 bit, with 1 as true and 0 as false.

Examples

```
(a || b) && c                // an expression with logical operators
(a != 1'b0 || b != 1'b0) && (c != 1'b0)  // an equivalent expression
```

Bitwise Operators

The bitwise operators are negation (~), AND (&), OR (|), XOR (^), and XNOR (^~ and ~^). Bitwise operators perform corresponding bit-by-bit Boolean operations on two operands. Operands are always zero-extended to the largest size (also the size of the results) before the operation.

Example

```
(a | b) & c     // an expression with bitwise operators
```

Reduction Operators

The reduction operators are AND (&), NAND (~&), OR (|), NOR (~|), XOR (^), and XNOR (~^, ^~). They perform Boolean operations on the bits of the single operand. The result is always 1 bit. The following identities should be used as the definition of reduction NAND, NOR, and XNOR:

```
~&A or ~(&A)           //two equivalent definitions of reduction NAND
~|A or ~(|A)           //two equivalent definitions of reduction NOR
~^A or ^~A or ~(^A)    //three equivalent definitions of reduction XNOR
```

Examples

```
assign A = 4'b1111;
assign B = 4'b0000;
assign C = 4'b0110;
assign D = 4'b1101;
assign X = &A;             // X is 1'b1
assign X = |B;             // X is 1'b0
assign X = ^C;             // X is 1'b0
assign X = ~&D;            // X is 1'b1
```

Shift Operators

The logical shift operators are left shift (<<) and right shift (>>). They perform shifts on the first (left-most) operand by the number of bits specified by the second operand. In both cases, the vacant bit positions are filled with zeros. The size of the result is always the same as that of the first operand.

Examples

```
(a << 4)           // Left-shift a by 4 bits
r << (s & 32'hf);  // Left-shift r by expression
```

In the second example, the first operand, `r`, is shifted by the amount specified by the second operand. The second operand is another expression specifying a bitwise AND of its first operand, `s`, and the 32-bit sized constant, `0000000F`.

Extraction Operator

The extraction operator, [ ], is used to either extract one or more consecutive bits from a wire vector or to access a value in a TIE table. When it is used to extract bits in a wire vector, a simple variable, a single integer constant, or a pair of integer constants separated by a colon (:) are allowed inside [ ]. When a pair of integer constants is used to specify a sub-range as *[from:to],* the first integer representing the *from* index has to be greater than or equal to the second integer representing the *to* index. When the extraction operator is used to get a value from a TIE table, only a simple variable is allowed inside [ ].

Examples

```
wire [7:0] A;
wire [2:0] B;
wire [2:0] C;
wire [4:0] D;
assign A = 8'b11110000;
assign B = A[5:3]; // B is 3'b110
assign D = A[0:4];  // Error: first integer should be greater than
                    // or equal to the second as in A[4:0].
assign C = 3'b010;
wire V = A[C]; // extract one bit of A indexed by C
table T 3 8 {...}
assign A = T[C]; // assign A with a value in table T indexed by C
```

Concatenation Operator

The *concatenation operator*, `{,}`, is used to append several operands to form wider bit vectors. The size of the result is the sum of the sizes of all the operands. The operands are ordered from the most significant to the least significant.

Examples

```
assign A = 2'b01;
assign B = 3'b101;
assign X = {A, B};               // X is 5'b01101
assign Y = {A[1], X, B[0]};      // Y is 7'b0011011
```

Replication Operator

The replication operator, `{n{exp}}`, specifies *n* concatenations of `exp`, where *n* must be an integer constant. While the above syntax for the replication operator is the recommended syntax, the TIE compiler also accepts `n{exp}` (that is, without the outer parenthesis) as an alternative syntax for backward compatibility reasons.

Example

```
{3{2'b01}}         //the result is 6'b010101
```

Conditional Operators

The conditional operator, *cond* `?` *true-exp* `:` *false-exp*, takes a logical expression and two operands. The result is *true-exp* if the logical expression *cond* evaluates to 1, and is *false-exp* otherwise. The two operands are zero-extended to the largest operand size, which is also the result size. If *cond* is not a logical expression, it is converted to the logical expression *cond* `!= 1'b0`.

Example

```
assign r = (x & y) ? 1 : 0;
```

Built-in Modules

Certain commonly used computations have no language-defined operators. However, using other language constructs is either very tedious to describe or very hard to implement efficiently. TIE provides a set of commonly used operators as built-in modules invoked in a way similar to function calls. These built-in modules are explained in Chapter 29, “TIE Built-in Modules” on page 233.

Example

The following semantic description uses the TIEadd built-in module to share an adder between ADD and SUB instructions:

```
assign arr = TIEadd(ars, SUB ? ~art : art, SUB);
```

Functions

A TIE function is similar to a function in C or Verilog. It encapsulates a specific computation and is used for modular code development. Functions also enable sharing of hardware between multiple instructions. Functions are described in more detail in Chapter 20, “Function (`function`) Sections” on page 165.

Operator Precedence

If no parentheses are used, an expression is evaluated according to the precedence specified in Table 28–17. For binary operators with the same precedence, the evaluation order is from left to right. All unary operators have the same precedence and are evaluated from right to left.

Note: Use parentheses specify the evaluation order and improve the readability and reliability of the code.

Table 28–17. Operator Precedence

Operators	Symbols	Precedence
Bit Extraction	`[]`	Highest
Concatenation, replication	`{}, {{}}`	
Unary and reduction	`~, !, &, ~&, ^, ^~, ~^, |, ~|`	
Multiply	`*`	
Add, subtract	`+, -`	
Shift	`<<, >>`	
Relational	`<, <=, >, >=`	
Equality	`==, !=`	
Bitwise and	`&`	
Bitwise xor and xnor	`^, ^~, ~^`	
Bitwise or	`|`	
Logical and	`&&`	
Logical or	`||`	
Conditional	`?:`	Lowest

28.3.5 Expression Width (Bit-Width)

To achieve consistent results, it is important to have precise rules for determining the number of bits in the evaluation-result of an expression, which we call the *expression width*. Because the TIE language is taken from a subset of Verilog HDL, the rules for computing expression widths are the same in both languages.

An expression's width is determined by its *computed-width*, shown in Table 28–18, and by its *required-width* (if any), shown in Table 28–19. All expressions have a computed-width, but not all expressions have a required-width. If an expression has no required-width, (that is, the expression's width depends only on the expression itself) then it is called a *self-determined* expression. If the expression has a required-width, (that is, the width also depends on the parent expression) then it is called a *context-determined* expression. The required-width of a *context-determined* expression is determined by the required-width of its parent expression or by the computed-width of the parent expression's operands, as shown in Table 28–19.

In determining an expression's width, the expression's computed-width is first determined, using the rules in Table 28–18. Then the expression's required-width is determined, using the rules in Table 28–19, starting from the highest parent level of any expression — an `assign` statement. In the case of a *context-determined* expression, the required-width takes precedence. If the required-width is different from the computed width, then the computed-width is either zero-extended or truncated (at its most-significant end) to match the required-width before the expression is evaluated and its result used in any parent expression (another expression that uses the expression as an operand).

Table 28–18 summarizes how an expression's computed-width is determined for the various types of expressions. In this table, `CW(x)` designates the computed-width of operand `x`, in bits. If `x` is a constant, then `CW(x)` is simply the width of the constant. In Table 28–18, the letters `i`, `j`, and `k` are used to denote expressions, the letter `n` is used to denote constants, and the letters `u` and `v` are used to denote variables.

Table 28–18. Computed-Width of Expressions

Type of Operation	Expression	Computed-Width of Expression[1]

two-operand arithmetic and bitwise	`i` op `j`, where op is:
	`+ - *`
	`& | ^ ^~`
		`max(CW(i), CW(j))`

two-operand logical and relational	`i` op `j`, where op is:
	`&& ||`
	`> < >= <= == !=`
		`1`

one-operand bitwise	`opi`, where op is: `~`	`CW(i)`
one-operand reduction	`opi`, where op is: `& ~& | ~| ^ ^~`	`1`
one-operand negation	`opi`, where op is: `!`	`1`
shift	`i` op `j`, where op is: `>> <<`	`CW(i)`[2]
conditional	`i ? j : k`	`max(CW(j), CW(k))`
concatenation	`{i, ..., j}`	`CW(i) + ... + CW(j)`
replication	`{ n {i, ..., j} }`	`n * (CW(i) + ... + CW(j))`
bit extraction	`u[v]` or `u[n]` `u[m:n]`	`1` `m - n + 1`
built-in modules	`<module-name>(...)`	as defined in Chapter 29
functions	`<function-name>(...)`	width of return value of the function as defined in Chapter 20

1. CW(x) = computed-width of operand x, in bits. If x is a constant, then CW(x) is simply the width of the constant.
2. The j operand is never truncated. A large j relative to i results in all bits in i being shifted out, leaving only zeros.

To determine whether an expression has a required-width, the rules in Table 28–19 are applied recursively, starting from the parent `assign` statement and working downward into the expression-nesting levels. As described in Section 28.2.2, `assign` statements consist of a wire or wire vector on the left side of their assignment operator and an expression on their right side.

The basic expression-width rule for `assign` statements, and the starting point for evaluating nested expressions, is:

> *Basic Rule For assign Statements:* The right side (expression) of an `assign` statement always has a required-width, which is equal to the statement's left side (wire or wire vector).

Thus, in the statement:

```
assign a = b << (c + d)
```

the `b << (c + d)` shift expression always has a required-width, because it is on the right side of an `assign` statement. Shift expressions are shown in row five of Table 28–19, and the operand widths for expressions that have a required-width (such as the shift expression) are shown in column two. This table entry shows:

- the `i` operand of all `i op j` shift expressions — in this example, the `b` operand — has a required-width equal to the required-width of the expression in which the operand occurs (this width is determined by the left side of the `assign` statement), and
- the `j` operand — in this example, the `(c + d)` expression — has no required-width, so its computed-width, given in Table 28–18 (row one, column three), is the maximum width of its two operands, `c` and `d`. If `c` and `d` are themselves expressions, then the third column of Table 28–19 is used — because the `(c + d)` expression has no required-width — to determine whether the operands of these expressions have a required-width.

To summarize, Table 28–19 specifies the required-width of each operand in an expression, based on whether the expression itself has a required-width. Starting from the fact that all expressions on the right side of an `assign` statement have a required-width, the second column of Table 28–19 determines whether the operands of that expression have a required-width. Then, if those operands are themselves expressions, continue using Table 28–19 — the second column for expressions that *have* required-width, and the third column for expressions that have *no* required-width — to determine the required-width of the nested operands. In Table 28–19, the letters `i`, `j`, and `k` are used to denote expressions, the letter `n` is used to denote constants, and the letters `u` and `v` are used to denote variables.

Table 28–19. Required-Width of Expression Operands

Expression	Operand Width When Expression Has Required-Width[1]	Operand Width When Expression Has No Required-Width[1]
i op j, where op is: + - * & | ^ ^~ ~^	RW(i)=max(RW(i op j), CW(i op j))[2] RW(j)=max(RW(i op j), CW(i op j))[2]	RW(i)=max(CW(i),CW(j)) RW(j)=max(CW(i),CW(j))
i op j, where op is: > < >= <= == !=	RW(i)=max(CW(i),CW(j)) RW(j)=max(CW(i),CW(j))	RW(i)=max(CW(i),CW(j)) RW(j)=max(CW(i),CW(j))
i op j, where op is: && ||	i: no required-width j: no required-width	i: no required-width j: no required-width
opi, where op is: ~	RW(i)=max(RW(op i), CW(op i))[2]	i:no required-width
opi, where op is: ! & ~& | ~| ^ ^~ ~^	i: no required-width	i: no required-width
i op j, where op is: >> <<	RW(i)=max(RW(i op j), CW i op j))[2] j: no required-width	i: no required-width j: no required-width
i ? j : k	i: no required-width RW(j)=max(RW(i ? j : k), CW(i ? i : j))[2] RW(k)=RW(i ? j : k)	i: no required-width RW(j)=max(CW(i),CW(j)) RW(k)=max(CW(i),CW(k))
{i, ..., j}	i: no required-width j: no required-width	i: no required-width j: no required-width
{ n {j, ..., k} }	n: no required-width j: no required-width k: no required-width	n: no required-width j: no required-width k: no required-width
u[v], u[n], u[m:n]	*no operands have required-width*	*no operands have required-width*
built-in modules	*no operands have required-width*	*no operands have required-width*
functions	*argument width as defined in function definition*	*argument width as defined in function definition*

1. RW(x) = required-width of operand x, in bits. CW(x) = computed-width of operand x, in bits. "no required-width" is equivalent to RW(x) = CW(x).
2. The operand's required-width is equal to maximum width of the computed width and the required-width of the expression of which the operand is a part.

As mentioned at the beginning of the chapter, the TIE language follows Verilog bit-width rules. Thus when operands of an expression have mismatched bit-widths, one or more of the operands are zero extended or truncated using the bit width rules defined in this section. These rules are convenient and do not pose problems in most cases. However, in some cases, strict bit-width matching may be desirable. For example, if the data represented is signed, then zero extension due to mismatched bit widths of the operands of an expression results in incorrect functionality. In such cases, the TIE compiler can be used to check mismatches in bit widths.

The TIE compiler can optionally generate a warning if the computed-width of the right-hand side of an assignment, generated by rules in Table 28–17, is not the same as the width of the left-hand side. Since the required-width of the right-hand side of an assignment is the width of the left-hand side, this mismatch indicates that when evaluating the right-hand side, some operations may be subject to truncation or zero-extension of the operand. This deserves the designer's attention to determine if it should be allowed. The TIE compiler will also issue a similar warning when the two operands of an operation other than an assignment are not the same width, and the operand widths are not self-determined.

In the following examples, the evaluation of expressions using bit-width rules is illustrated. In addition, the optional warnings that are generated by the TIE compiler in each case are also shown.

Example 1

```
wire [2:0] a;
wire [3:0] b;
wire [4:0] c;
assign c = a + b;
```

In this example, the required-width of the expression `a + b` is 5 bits, by the basic rule for `assign` statements. The `a` and `b` operands both have a required-width of 5 bits, by the rule in Table 28–19 (row one, column two), which states that the required-widths of both operands are equal to the required-width of the `a + b` expression itself. So, `a` and `b` are both zero-extended to 5 bits, `a + b` is evaluated using 5 bits, and the result is assigned to `c`.

If this code is compiled with the option to generate warnings on bit-width mismatches, the TIE compiler will generate two warning messages. To do the bit-width checking, the right-hand side of the assignment is evaluated using computed-width only. Using Table 16 (row one, column two), the computed-width of the right-hand side expression `a + b` is 4 bits, which is not the same as the width of the left-hand side `c`, and this will generate one warning. The second warning is generated because the two operands of the + operation are not of the same width.

Example 2

```
wire [2:0] a;
wire [3:0] b;
wire [4:0] c, d;
assign d = (a == b) + c;
```

In this example, the required-width of `(a == b) + c` is 5 bits, by the basic rule for `assign` statements. The `(a == b)` and `c` operands both have a required-width of 5 bits, by the rule in Table 28–19 (row one, column two), as in the previous example. The `a` and `b` operands both have a required-width of 4 bits, by the rule in Table 28–19 (row two, column two), and this required-width takes precedence over the expression's computed-width of 1 bit. So, the `(a == b)` expression is evaluated by first zero-extending `a` to 4 bits, comparing `a` with `b` to produce a 1-bit result, zero-extending the comparison result to 5 bits, performing the addition in 5 bits, and assigning the result to `d`.

If this code is compiled with the option to generate warnings on bit-width mismatches, the TIE compiler will generate a warning about the two operands of the + operation not being the same size. Using Table 16 (row two), the computed-width of the expression `(a == b)` is 1 bit. Using Table 16 (row one), the computed-width of the right-hand side expression `(a == b) + c` is 5 bits, which is the same as the width of the left-hand side `d`. Therefore, there is no warning about the assignment.

Example 3

To compute the sum of two 16-bit numbers and assign the high-order 16 bits to another variable (truncating the least-significant bit), the following statements would produce the *wrong* result:

```
wire [15:0] m, a, b;
assign m = (a + b) >> 1;
```

Because `m` is 16 bits, the required-width of the shift expression is 16 bits, and by the rule in Table 28–19 (row two, column two) the required-width of the `(a + b)` operand in the shift expression is also 16 bits. By shifting `(a + b)` 1 bit to the right, the least-significant bit is lost and the most-significant bit of the result assigned to `m` is always 0.

This is an example of a case in which the TIE compiler would not be able to detect that the code does not generate the correct result according to the designer's intent, since the rules for bit width are not violated.

The following statements correctly compute the result:

```
wire [16:0] tmp;
wire [15:0] m, a, b;
assign tmp = (a + b) >> 1;
```

```
assign m = tmp;
```

Because `tmp` is 17 bits, the required-width of `(a + b)` is 17 bits. So, the most-significant bit is not lost.

Example 4

All the operands of a designer-defined function have a required width equal to the width of the argument in the function definition. This is illustrated in the following example.

```
function [7:0] donop([7:0] a) {
     assign donop = a;
}

operation FOO {out AR result, in AR inp} {} {
     wire bit0 = inp[0];
     assign result = donop(~bit0);
}
```

The function `donop` is defined with an input argument that is 8-bits wide, and hence the required width of `~bit0` and `bit0` in the expression "`donop(~bit0)`" is also 8. Thus `bit0` is first 0-extended to 8-bits, this 8-bit expression is then inverted to compute `~bit0`, and this value is passed as an argument to the function `donop`. If the instruction `FOO` is executed with its input operand having the value 0x0, the output has the value 0xFF. Note that the result would have been 0x1 if the function did not impose a required width on its arguments, and the expression "~bit0" was evaluated as a single bit wide value and then 0-extended to 8-bits.

If this code is compiled with the option to generate warnings on bit-width mismatches, the TIE compiler will generate a warning, to indicate that the width of the function argument does not match that of the function input.

29. TIE Built-in Modules

Certain commonly used computations have no language-defined operators. However, using other language constructs is either very tedious to describe or very hard to implement efficiently. TIE provides a set of commonly used operators as built-in modules invoked in a way similar to function calls. The following are the rules common to all built-in modules:

- Each built-in module has its own definition of computed-width. The definition is not affected by the required-width of the context.
- The inputs to the built-in modules have no required-width.
- In the case where an input does not have enough bits as needed by the definition of the built-in module, it is 0-extended.

29.1 TIEadd

Synopsis

```
sum = TIEadd(a, b, cin)
```

Definition

```
sum = a + b + cin
```

Computed-width

```
CW(sum) = MAX(CW(a), CW(b)) + 1
```

Description

Addition with carry-in. It is an error if the `TIEadd` does not have three arguments and the computed-width of the last argument is not 1. If the computed-width of a and b are different, the narrower input is evaluated in its computed-width and then 0-extended.

29.2 TIEaddn

Synopsis

```
sum = TIEaddn(A0, A1, ..., An-1)
```

Definition

```
sum = A0 + A1 + ... + An-1
```

Computed-width

$CW(sum) = MAX(|Ai|) + ceil(log_2(n))$

Description

The `TIEaddn` module is an n-number addition. There must be at least three arguments to the built-in module. If the computed-width of the inputs are different, the narrower inputs are evaluated in their computed-width and then 0-extended.

The advantage of using the `TIEaddn` built-in module is that the underlying RTL implementation is much more timing and area efficient than simply using the + operator.

29.3 TIEcsa

Synopsis

```
{carry, sum} = TIEcsa(a, b, c)
```

Definition

```
carry = a & b | a & c | b & c
```

```
sum = a ^ b ^ c
```

Computed-width

```
CW(carry) = CW(sum) = MAX(CW(a), CW(b), CW(c))
```

Description

Carry-save adder (`csa`). `TIEcsa` must have exactly three arguments. If the computed-width of the inputs are different, the narrower inputs are evaluated in their computed-width and then 0-extended. The computed-width of `TIEcsa` is twice the maximum input width. The lower half of the result represents the sum bits and the upper half, the carry bits. To add the sum and carry, the carry must be shifted to the left by 1 bit.

This module is provided to allow efficient implementation of adding or subtracting several numbers, accomplished by a series of csa reductions followed by a single adder.

Example

```
state S 32 add_read_write
opcode SUB2 op2=0 CUST0
iclass subsub {SUB2} {out arr, in ars, in art} {in S}
semantic sem {SUB2} {
     wire [31:0] carry, sum;
     assign {carry, sum} = TIEcsa(ars, ~art, ~S);
     assign arr = TIEadd({carry[30:0], 1'b1}, sum, 1'b1);
}
reference SUB2 {
     assign arr = ars - art - S;
}
```

This description implements a SUB2 instruction that subtracts `art` and S from `ars` and returns the result in `arr`, as defined in the reference. However, two subtractions typically require two cycles. Thus, the semantics of SUB2 is written to perform a csa reduction followed by an addition. Furthermore, it makes use of a property of 2's complement number representation: `-a = ~a + 1.`

29.4 TIEcmp

Synopsis

```
{lt, le, eq, ge, gt} = TIEcmp(a, b, signed)
```

Definition

```
lt = (a < b)

le = (a <= b)

eq = (a == b)

ge = (a >= b)

gt = (a > b)
```

Computed-width

```
CW(lt) = CW(le) = CW(eq) = CW(ge) = CW(gt) = 1
```

Description

Signed and unsigned comparison. It is an error if the `TIEcmp` does not have three arguments and the computed-width of the last argument is not 1. The input operands a and b are required to have the same computed-width. If the input `signed` is true, the comparison is for signed data. Otherwise, the comparison is for unsigned data.

29.5 TIElzc

Synopsis

o = TIElzc(a)

Definition

```
o = a[w-1] ? 0 : a[w-2] ? 1 : ... a[0] ? (w-1) : w
```

where `w` is the computed width of a

Computed-width

$CW(o) = ceil(\log_2(CW(a) + 1))$

Description

This module performs a leading zero count. It takes only one argument as its input, and returns a count of the number of leading zeros in this input wire.

Example:

```
state CNT1 6 add_read_write
operation LZC {out AR cnt0, in AR val0, in AR val1} {out CNT1} {
     wire [3:0] tmp = TIElzc(val0[9:0]);
     assign cnt0 = {28'b0, tmp};
     assign CNT1 = TIElzc(val1);
}
```

There are two leading zero computations performed in this example. The first, whose input is a 10-bit wire (`val0[9:0]`) returns a 4-bit result with legal values in the range 0 and 10. The second, whose input is a 32-bit wire (`val1`), returns a 6-bit result with legal values in the range 0 to 32.

29.6 TIEmac

Synopsis

```
o = TIEmac(a, b, c, signed, negate)
```

Definition

```
o = negate ? c - a * b : c + a * b
```

Computed-width

```
CW(o) = MAX(CW(a)+CW(b), CW(c))
```

Description

Multiply-accumulate. The multiplication is signed if `signed` is true and unsigned otherwise. The multiplication result is subtracted from the accumulator `c` if `negate` is true, and added to the accumulator `c` otherwise. If the computed-width of the multiplication is less than the width of the accumulator, the multiplication result is sign-extended if `signed` is true and 0-extended otherwise. If the computed-width of the accumulator is less than the computed-width of the multiplication, the accumulator is sign-extended if `signed` is true and 0-extended otherwise. The computed-width of `signed` and `negate` must be 1. Note that the computed-width of the `TIEmac` module is defined to be the maximum of the computed width of the multiplication and the computed width of the accumulator. This implies that if the output of `TIEmac` is assigned to a wire that is wider than this computed width, the `TIEmac` output will be 0-extended regardless of the value of the `signed` bit.

To avoid slowing down the clock frequency of the Xtensa processor, any instruction that uses `TIEmac` should be allocated at least two cycles to execute. However, by carefully choosing the instruction schedule, it is possible to achieve a throughput of one `TIEmac` operation per cycle.

Example:

```
state ACC 40
operation MAC {in AR m0, in AR m1} {inout ACC} {
     assign ACC = TIEmac(m0[15:0], m1[15:0], ACC, 1'b1, 1'b0);
}
schedule sched {MAC} {
     use m0 1;
     use m1 1;
     use ACC 2;
     def ACC 2;
}
```

This description implements an instruction MAC that performs signed 16-bit by 16-bit multiply and accumulates the result in a 40-bit state ACC. The MAC schedule specifies that m0 and m1 are used in stage 1, the accumulator ACC is used in stage 2, and the result is written into ACC in stage 2. With this schedule, the Xtensa processor can issue a MAC instruction every cycle.

29.7 TIEmul

Synopsis

```
prod = TIEmul(a, b, signed)
```

Definition

```
prod = a * b
```

Computed-width

```
CW(prod) = CW(a) + CW(b)
```

Description

The TIEmul module performs a multiply operation. The multiplication is signed if signed is true and unsigned otherwise. The computed-width of signed must be one. Note that the computed width of the TIEmul module is defined to be the sum of the computed widths of the multiplier and the multiplicand. This implies that if the output of TIEmul is assigned to a wire that is wider than this computed width, the TIEmul output will be 0-extended regardless of the value of the signed bit.

To avoid slowing down the clock frequency of the Xtensa processor, any instruction that uses TIEmul should be allocated at least two cycles to execute. Because multicycle TIE instructions are implemented in a fully pipelined fashion, this will still allow you to achieve a throughput of one TIEmul operation per cycle.

29.8 TIEmulpp

Synopsis

```
{p0, p1} = TIEmulpp(a, b, signed, negate)
```

Definition

```
p0 + p1 = negate ? - a * b : a * b
```

Computed-width

```
CW(p0) = CW(p1) = CW(a) + CW(b)
```

Description

The `TIEmulpp` performs a partial-product multiply operation. This module returns two partial products of the multiplication. The multiplication is signed if `signed` is true and unsigned otherwise. The sum of the two partial products equals the product. If `negate` is true, the sum equals the negative of the product. The definition does not give specific meaning to the individual partial product. The computed-width of `signed` and `negate` must be one.

This module is provided to allow efficient implementation of certain algebraic expressions involving multiplications, additions, and subtractions. For example, the following expression:

```
o = a * b + c * d + e
```

can be efficiently computed using the `TIEmulpp` and `TIEaddn` module as:

```
{p0, p1} = TIEmulpp(a, b, 1'b1, 1'b0)
{p2, p3} = TIEmulpp(c, d, 1'b1, 1'b0)
o = TIEaddn(p0, p1, p2, p3, e)
```

Restriction

In the current TIE compiler implementation, the resulting partial products cannot be 0-extended, 1-extended or sign-extended. The partial products must be summed and assigned to a wire of the same width as the computed width of the partial products. Violating this constraint will result in incorrect hardware implementation. Cadence recommends that the partial products be summed using the `TIEadd`, `TIEcsa`, or `TIEaddn` modules; that all the operands of these modules have a width equal to the computed width of the partial products; and that the output be assigned to a wire of the same width.

To accumulate the results of a multiply operation in an accumulator with more bits, the only correct way is to 0 or sign extend the multiplier inputs such that the width of the resulting partial products matches that of the accumulator.

29.9 TIEmux

Synopsis

```
o = TIEmux(s, D0, D1, ..., Dn-1)
```

Definition

```
o = s == 0 ? D0 : s == 1 ? D1 : ... : s == n-2 ? Dn-2 : Dn-1
```

Computed-width

```
CW(o) = MAX(CW(Di))
```

Description

The `TIEmux` module is an n-way multiplexor. This module returns one of the n data depending on the value of the select signal. The number of data, n, must be power-of-2. The width of the select signal must be log2(n).

29.10 TIEpsel

Synopsis

```
o = TIEpsel(S0, D0, S1, D1, ..., Sn-1, Dn-1)
```

Definition

```
o = S0 ? D0 : S1 ? D1 : ... : Sn-1 ? Dn-1 : 0
```

Computed-width

```
CW(o) = MAX(CW(Di))
```

Description

The `TIEpsel` module is an n-way priority selector. This module selects one of n input data according to the values and priorities of the select signals. The first select signal has the highest priority and the last the lowest. If none of the selection signals are active, the result is 0. The width of select signals must be 1.

29.11 TIEsel

Synopsis

```
o = TIEsel(S0, D0, S1, D1, ..., Sn-1, Dn-1)
```

Definition

```
o = (size{S0} & D0) | (size{S1} & D1) |... (size{Sn-1} & Dn-1);
```

where `size` is the computed width of o as defined below.

Computed-width

```
CW(o) = MAX(CW(Di))
```

Description

The `TIEsel` module is an n-way, 1-hot selector. This module selects one of n input data values according to the values of the select signals. The select signals are expected to be 1-hot. If none of the selection signals are active, the result is 0. If multiple select signals are active, the output is implementation-specific and should not be relied on. The width of select signals must be 1.

30. TIEprint

TIEprint is similar to the *printf* function in the C programming language. It provides a mechanism to print the variables in a block of statements of your TIE code when simulating using the Xtensa ISS. TIEprint can be used to print any valid variable in an operation, reference, semantic, or function body, and can thus be useful when debugging a TIE instruction. This construct only affects simulation and is completely ignored for hardware generation. It thus has no impact on the area, timing, or power consumption of your hardware.

The output of TIEprint is written to the trace log of the ISS. By default, this printing is disabled and needs to be enabled by setting a command line flag (or enabling the option in Xtensa Xplorer) for the ISS. Refer to the *Xtensa Instruction Set Simulator (ISS) User's Guide* for details on this option.

30.1 TIEprint Syntax

```
TIEprint-def ::= TIEprint([qualifier,] format_string[, arg-list] );

qualifier ::= identifier | expression
format_string ::= quoted character string with
conversion_specification
arg-list ::= print-arg[, print-arg]*
conversion_specification ::= %[[0]width]conversion_specifier
conversion_specifier ::= u | d | x
print_arg ::= identifier | expression
expression ::= see Chapter 28
identifier ::= a unique identifier within the scope of the TIEprint
width = [0-9]+
```

qualifier is an optional argument to the TIEprint statement that, if provided, is evaluated to determine if the TIEprint should be active. If it evaluates to logical "TRUE" (nonzero), then the TIEprint will be active. The qualifier can be any expression or identifier within the current scope. If no *qualifier* is specified it defaults to 1; the TIEprint is always active.

format_string is a quoted character string that specifies the format of the TIEprint. It is similar to the format string for *printf* in the C programming language. The *format_string* can contain valid alphanumeric characters and conversion specifications that start with the % character. The characters that do not start with % are directly reproduced in the TIEprint output. The conversion specification may contain an optional integer following the % character. This integer specifies the minimum number of characters to be printed. If this number is larger than the size of the value string, the value string is pad-

ded with blanks and printed. There may also be an optional 0 character immediately following the % character and before the integer. If the 0 character is present, the value string is padded with 0's instead of blanks. A conversion specification must end with a conversion specifier. Following are valid conversion specifiers for `TIEprint`:

- *u* (unsigned integer)
- *x* (unsigned integer in hexadecimal format)
- *d* (signed integer)

`arg-list` is a comma separated list of one or more arguments to the `TIEprint`. The number of arguments must match the number of conversion specifiers in the format string. Each argument can be a valid identifier or expression in the current scope.

30.2 TIEprint Description

`TIEprint` is designed for easy debug of TIE code by inserting formatted print statements into the code, similar to what a C programmer might do to debug a C program. The output of `TIEprint` will appear in the trace log when a program running on the ISS executes the TIE instruction or function that contains the `TIEprint` statement. `TIEprint` has no impact on the hardware and is not a hardware debug mechanism. `TIEprint` can be used in operation, reference, semantic, or function sections.

A `TIEprint` statement specifies a format string and a number of arguments. In addition, there is an optional argument, which acts as a run time qualifier for the `TIEprint`. If a qualifier is provided, the `TIEprint` is printed only when the qualifier evaluates to logical "TRUE" (non-zero). If no qualifier is provided, the `TIEprint` will always be printed when the statements are executed. The format string can contain any characters. If a character starts with %, it marks the beginning of a conversion specification. Valid conversion specifiers in `TIEprint` are unsigned integer in hexadecimal (x), unsigned integer in decimal (u), and signed decimal integers (d). Following are examples of valid conversion specifications:

```
%u        // unsigned integer
%08x      // unsigned hexadecimal, minimum length 8 characters, with 0
          // padding
%4x       // unsigned hexadecimal, minimum length 4 characters, no
          // padding
```

The arguments to `TIEprint` can be any valid expression with a valid identifier in the scope. Following are the valid identifiers that can be in a `TIEprint` argument or qualifier:

- All the input argument names, including input states.
- All instruction names that are implemented in the semantic, or the name of the current operation, which are implicit inputs.

- All wires declared explicitly in the computation.

Output arguments cannot be used in `TIEprint` directly because outputs cannot be read from, and `TIEprint` arguments are considered to be read in the semantic. You can indirectly print the value of an output by using a temporary wire, as illustrated in the example of Section 30.2.1. If an inout argument is used as a `TIEprint` argument, the input value of the argument is printed.

The size (bit-width) of the arguments to `TIEprint` must be less than or equal to 32 bits. The TIE compiler will flag an error if the size exceeds 32 bits. Arguments that are wider than 32 bits must be broken up into 32-bit values for printing.

A block of statements can have any number of `TIEprints`. The output of the `TIEprint` is printed in lexical order in the simulation output. This is illustrated in Section 30.3.

30.2.1 Example: Simple TIEprint

This is a simple example illustrating the use of `TIEprint` in an operation body.

```
operation MYADD {out AR sum, in AR in0, in AR in1} {} {
     wire [31:0] result = in0 + in1;
     assign sum = result;
     TIEprint("MYADD( %d, %d ) = %d", in0, in1, result);
}
```

In this example, the two input values of the instruction and their sum are printed as integers. There is no qualifier in this `TIEprint`, so the string is printed every time the instruction executes. Note that since the output argument `sum` cannot be printed directly, a temporary wire `result` is declared and used in `TIEprint`.

30.2.2 Example: TIEprint with Expressions

In this example, the `TIEprint` arguments are expressions that break up wide variables to 32 bit slices for printing.

```
state ACCUM 56
operation MAC24 {in AR a, in AR b} {inout ACCUM} {
     wire [47:0] product = TIEmul(a[23:0], b[23:0], 1'b0);
     assign ACCUM = ACCUM + product;

     TIEprint("mul %x * %x = %04x_%08x\n",
               a[23:0], b[23:0], product[47:32], product[31:0]);

     TIEprint("Input value of accumulator is %06x_%08x\n",
                                  ACCUM[55:32], ACCUM[31:0]);
```

```
        TIEprint("Upper bits of a & b are %4x\n",{a[31:24], b[31:24]});
    }
```

The 48-bit product of the multiply and the 56-bit accumulator are both printed in this example by splitting them into 32-bit wide variables. Furthermore, these values are printed with leading zero padding, using hexadecimal format. Note that the state ACCUM is an in-out operand, and the value printed will be its input value. It is also possible to concatenate multiple wires and print them as a single value, as illustrated by the concatenation of the upper 8 bits of the two input operands.

30.2.3 Example: TIEprint with qualifier

This TIEprint example illustrates the use of a qualifier to define a run-time condition under which a TIEprint should be activated.

```
function [31:0] addsub ([31:0] a, [31:0] b, sub) {
     wire [31:0] tmp = sub ? ~b : b;
     wire [32:0] addresult = TIEadd(a, tmp, sub);
     assign addsub = addresult[31:0];

     // print when performing a subtract operation, and the
     // carry bit of the output is also 1
     TIEprint(sub && addresult[32],
          "Overflow in subtraction %d - %d\n",a, b);
}

operation MYADD{out AR sum, in AR in0, in AR in1} {} {
     assign sum = addsub(in0, in1, 1'b0);
}

operation MYSUB {out AR diff, in AR in0, in AR in1} {} {
     assign diff = addsub(in0, in1, 1'b1);
}
```

The function addsub has a TIEprint with a qualifier. The qualifier is an expression that does a logical OR of two variables, sub, which is an input to the function, and the most significant bit of addresult, which indicates if there is a carry from the addition operation. If these two conditions are true, then the qualifier evaluates to logical "TRUE", and the TIEprint will be printed. This function is called from the MYADD and MYSUB operations, but sub is only true for MYSUB. So the TIEprint will only be active when the MYSUB instruction executes and the subtract operation results in a carry.

The qualifier provides a method to do a run-time check or assertion by activating a TIEprint in the simulation under certain error or corner case conditions. It can also be useful when writing TIEprint in a semantic that implements many instructions and the

TIEprint is only specific to a particular instruction. In this case, the instruction name can be the qualifier, so that the TIEprint is active only when the semantic executes that particular instruction.

30.3 TIEprint Output

The output of TIEprint is written to the trace log of the Xtensa ISS, when enabled by the appropriate command line option for the ISS. The TIEprint output is written only for instructions that are committed in the processor pipeline; it is not written for speculatively executed instructions that subsequently get killed or replayed. The output is seen in the lexical order in which the TIEprint statements appear in the TIE description. This may be different than the order in which the TIEprint statements get evaluated in the processor pipeline, as illustrated by the following example.

```
function [31:0] adder ([31:0] a, [31:0] b) {
     wire [31:0] result = TIEadd(a, b, 1'b0);
     assign adder = result;
     TIEprint("adder(%u, %u) = %u\n", a, b, result);
}

function [31:0] mult16 ([15:0] a, [15:0] b) {
     wire [31:0] result = TIEmul(a, b, 1'b0);
     assign mult16 = result;
     TIEprint("mult16(%u, %u) = %u\n", a, b, result);
}

operation MYMAC { inout AR accum, in AR a, in AR b } {}
{
     wire [31:0] mult;
     TIEprint("Multiplier result %u added to %u\n",
                                          mult, accum);
     assign mult = mult16(a[15:0], b[15:0]);
     assign accum = adder(accum, mult);
}

schedule mymac { MYMAC } { use a 1; use b 1; use accum 2; def accum 2;}
```

In this example, the operation MYMAC has a TIEprint followed by two function calls, each of which have a TIEprint. When this instruction executes on an Xtensa processor with a 5-stage pipeline, operands a and b are read in the E stage of the processor pipeline, and the function mult16 executed. The operand accum is read in the next cycle, in the M stage of the processor pipeline, and then the function adder is executed. Thus the functions mult16 and adder will execute their TIEprint in the E and M stages respectively. The TIEprint statement in the operation MYMAC body prints the multiplier result from E stage and the operand accum read in the M stage, and can therefore

be executed in the M stage. However, the actual output will appear in the lexical order; that is, the output of the `TIEprint` in the operation body of `MYMAC` will appear first, followed by the output of the `TIEprint` statements in the functions `mult16` and `adder`.

Following is an example C program and the output from simulating this program. Note that the output of `TIEprint` and the output of the printf from the C program are mixed. This can be controlled with options to the ISS.

```
void main(void) {
     unsigned int a, b, acc;

     a = 16; b = 16;
     acc = 128;
     MYMAC(acc, a, b);
     printf("Accumulator is %d\n", acc);
}

// Program output in trace log
Multiplier result 256 added to 128
mult16(16, 16) = 256
adder(128, 256) = 384

Accumulator is 384
```

If operations with `TIEprint` are in a FLIX instruction with more than one slot, the `TIEprint` outputs from each slot are tagged or prefixed with the slot index.

30.4 Implementation Restrictions

- The arguments to `TIEprint` can only be up to 32 bits wide.
- A single `TIEprint` statement can have up to 32 arguments.
- Verilog style binary format printing is not allowed.
- Trigraphs are not supported.

31. TIE Preprocessor

Certain language features of TIE are provided by means of a preprocessor. Conceptually, preprocessing is the first step in TIE compilation. The TIE preprocessor is based on the Perl text-manipulation language. TIE provides two simple ways of embedding Perl code inside a TIE description: embedding Perl statements and embedding Perl expressions.

31.1 Embedding Perl Statements in TIE

Any block of Perl code can be embedded in a TIE description. If a line starts with a semicolon (;), the rest of the line is treated as a Perl statement. The first character must be a semicolon (;) with no leading blank characters (spaces or tabs) preceding it. Otherwise, the TIE compiler fails.

31.2 Embedding Perl Expressions in TIE

Any Perl expression can be embedded in a TIE description. Anything enclosed inside backquotes (`‘...‘`) is treated as a Perl expression.

31.3 TIE Preprocessing

During TIE preprocessing, a TIE description is converted to a Perl program. The output from the evaluation of that program is another TIE description, which is passed on to the second stage of compilation. The conversion of a TIE program to a Perl program follows these rules:

- An embedded Perl statement (preprocessing line) is copied to the output with the leading semicolon removed.
- A line not starting with a semicolon (normal line) is converted to a Perl `print` statement that prints the original line.
- An embedded Perl expression in a normal line is evaluated and replaced by its value before the line is printed.

31.3.1 Example

The following TIE segment declares a table of the first 32 even numbers without using preprocessing.

```
table even 18 32 {
      0,  2,  4,  6,  8, 10, 12, 14,
     16, 18, 20, 22, 24, 26, 28, 30,
     32, 34, 36, 38, 40, 42, 44, 46,
     48, 50, 52, 54, 56, 58, 60, 62
}
```

Using TIE preprocessing constructs, the same description can be written as:

```
table even 18 32 {
; for ($i = 0; $i < 31; $i++) {
     ‘$i * 2‘,
; }
     62
}
```

31.4 Predefined TIE Preprocessor Variables

One of the major motivations for using the TIE preprocessor is to write configurable TIE descriptions compatible with the configurable Xtensa processor. For this reason, the TIE preprocessor defines a set of standard variables to the values defined by the Xtensa core configuration. Table 31–20 summarizes the predefined TIE preprocessor standard variables.

Table 31–20. Predefined TIE Preprocessor Variables

Name	Legal Values	Description
DataMemWidth	32, 64, 128, 256, 512	Data memory access width.
IsaMemoryOrder	LittleEndian, BigEndian	Memory order of Xtensa processor.
IsaUseBooleans	0, 1	Whether Xtensa core Boolean option is available.
IsaCoprocessorCount	0 – 16	Number of coprocessors the Xtensa core supports.
IsaMaxInstructionSize	4, 8, 12, 16	Maximum instruction size in bytes.

31.4.1 Example

If a certain part of a TIE description is memory-order-dependent, it can be written as:

```
; if ($IsaMemoryOrder eq "LittleEndian") {
     ...Little-Endian TIE Code...
; } else {
     ...Big-Endian TIE Code...
; }
```

32. Instruction Field (field) Sections

Instruction fields are defined with field description sections. Fields are subsets or concatenations of bits within a slot. New fields can also be defined as subsets or concatenations of previously defined fields.

In general, field definitions are not necessary for instructions created using the operation construct described in Chapter 7, “Instruction Operation (operation) Sections” on page 37. In earlier versions of the TIE language, fields were typically used to specify opcode encodings as described in Chapter 33, “Instruction Opcode (opcode) Sections” on page 257 and to create instruction operands as described in Section 34 “Instruction Operand (operand) Sections” on page 261.

32.1 Instruction Field Syntax

```
field-def ::= field name (sub-range | concatenation | size)
name ::= a unique identifier
sub-range ::= ident[from:to] | ident[index]
concatenation ::= {ident [, ident]*}
size ::= an integer representing the width of the field in bits
ident ::= slot | another-field
slot ::= a previously defined slot
another-field ::= a previously defined field
```

name is a unique identifier for the new field defined by this statement. A new field is defined by specifying either a *sub-range* of bits, a *concatenation* of bits, or by just specifying the size of the field. For a *sub-range* specified as *[from:to]*, the *from* index has to be greater than the *to* index. In the case of *concatenation*, the bits are ordered, left-to-right, from most-significant to least-significant. The bits in the newly defined field are ordered, left-to-right, from *n*-1 (most-significant) to 0 (least-significant), where *n* is the *size* of the field.

When a field is defined as a *sub-range* or *concatenation* of bits, all the bits in the specification must belong to the same slot. When a field is defined as a *concatenation* of other fields, all of them must be previously defined as fields in the same slot.

When a field is defined using the *size* argument, the TIE compiler automatically selects the actual bits of the instruction word that represent the field. Such a field would typically be used to create immediate operands of an instruction. The field is created in all the slots in which this instruction is defined.

The 24-bit instructions of the Xtensa ISA are part of a predefined slot called `Inst`. This slot also contains a number of predefined fields. All predefined fields in the current Xtensa architecture are listed in the *Instruction Formats and Opcodes* chapter of the *Xtensa Instruction Set Architecture (ISA) Reference Manual.*

The predefined fields depend on the byte order (little-endian or big-endian) for which your processor has been configured. The predefined fields list in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* includes both little-endian and big-endian byte orders. When defining new fields, it is important to pay special attention to the byte order of your Xtensa processor. In general, defining a new field as a subset of bits in a slot is always byte-order dependent. A new field defined as a concatenation or subset of bits of predefined fields is byte-order independent. Fields defined using the `size` argument are also byte-order independent.

32.2 Examples

The following example defines a 5-bit field named `foo` in the predefined slot named `Inst`.

```
field foo Inst[5:1]      // bit foo[0] is Inst[1]
```

The following example defines an 8-bit field as the concatenation of the predefined `r` and `t` fields:

```
field rt {r, t}
```

The actual bits of `rt` in `Inst` change depending on the byte order of the Xtensa processor; that is, they are {`Inst[15:12], Inst[7:4]`} in little-endian byte order and {`Inst[11:8], Inst[19:16]`} in big-endian byte order.

The following example defines two 4-bit fields, `x` and `y`, in a designer-defined slot named `slot_b`. It then creates an 8-bit field, `xy`, as the concatenation of the `x` and `y` fields:

```
length l32 32 {InstBuf[3:0] == 14}
format f32 l32 {slot_a, slot_b}

slot slot_a f32[31:20]
slot slot_b f32[19:4]

slot_opcodes slot_a {ABS}
slot_opcodes slot_b {NEG}

field x   slot_b[7:4]    // bit x[0] is slot_b[4] or f32[8]
field y   slot_b[11:8]   // bit y[3] is slot_b[11] or f32[15]
field xy  {x, y}         // bit xy[5] is slot_b[5] or f32[9]
```

The following example defines a 5-bit field without specifying the actual bits that make up the field. The field is then used to create an operand for the instruction BAR. Instruction BAR is present in the slots `slot_x` and `Inst`. The TIE compiler picks five available bits in slot `slot_x` and in slot `Inst` to represent the field `foo`.

```
length l64 64 {InstBuf[3:0] == 15}
format f64 l64 {slot_x, slot_y}

slot_opcodes slot_x {ADD, BAR}
slot_opcodes slot_y {NEG, ABS}
slot_opcodes Inst {BAR}

field foo 5
opcode BAR Inst
iclass bar {BAR} {out arr, in ars, in foo}
reference BAR {
     assign arr = ars + foo;
}
```

33. Instruction Opcode (opcode) Sections

Instruction opcodes are defined with `opcode` description sections. An opcode declaration typically defines the name or the mnemonic of a new, designer-defined TIE instruction. It can also define a sub-opcode of another (higher level) opcode. This allows the designer to create a hierarchical organization of opcodes that simplifies the management of opcode space and enhances code readability.

Instructions that are defined using `operation` description sections are not required to have `opcode` sections. If an opcode section is not specified for such an instruction, the TIE compiler automatically assigns an appropriate field encoding for the instruction. Refer to Chapter 7, “Instruction Operation (`operation`) Sections” on page 37 for more information on `operation` sections. Instructions that are not defined using `operation` description sections are required to have `opcode` sections.

An opcode is defined by assigning a specific value (encoding) to a specific field of the instruction word. An instruction opcode can also be defined by specifying a slot name and no field encoding. For such an opcode declaration, the TIE compiler automatically assigns an appropriate field encoding in the specified slot, consistent with the operands and usage of the instruction corresponding to the opcode.

33.1 Opcode Syntax

```
opcode-def ::= opcode name (encoding-list | slot-name)
name ::= a unique identifier
encoding-list ::= encoding [encoding]* [opcode-name]
slot-name ::= a previously defined slot name
encoding ::= field-name=sized-constant
opcode-name ::= a previously defined opcode
field-name ::= a previously defined field name
```

name is a unique identifier that can later be used to reference this opcode. *encoding-list* is a space-separated list of one or more encoding specifications, followed by an optional *opcode-name*. An encoding is a *field-name* and *sized-constant* pair, which assigns a specific value to a previously defined field. All encoding in the encoding-list must agree in all of their bit positions. A *sized-constant* must be specified using the form *size-base-value*, as defined in Section 28.3.1 “Expression Syntax” on page 218.

The *field-name* used in an opcode encoding must be the name of a previously defined field, as described in Chapter 32, “Instruction Field (`field`) Sections” on page 253. Alternatively, the field can be a predefined field of the Xtensa architecture. Similarly, *opcode-name* must be the name of a previously defined opcode or one of the predefined opcodes of the Xtensa architecture. All predefined opcodes and fields of the current Xtensa architecture are listed in the *Instruction Formats and Opcodes* chapter of the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

An alternative syntax for defining opcodes is to specify the name of the slot in which the instruction is desired, and to not specify the field encoding. When an instruction opcode is defined this way, the TIE compiler automatically chooses an appropriate field encoding based on the number and type of operands of the corresponding instruction. This feature automates the task of managing the opcode space and makes it easy to port the TIE code from one Xtensa configuration to another. When using this syntax, the *slot-name* can either be a previously defined slot in the TIE description, or the predefined slot `Inst`. Opcodes assigned to the slot `Inst` will be encoded as 24-bit instructions. It should be noted that while the TIE compiler will always generate a legal encoding, it is not guaranteed to be the same from one run of the compiler to another. In some special situations, the TIE compiler may be unable to perform the automatic encoding of opcodes. This typically happens when it is not able to identify enough unused bits in the instruction word to encode all the designer-defined instructions.

The following predefined opcodes are intended for customer use in defining TIE instruction extensions:

```
opcode CUST0 op1=4'b0110 QRST
opcode CUST1 op1=4'b0111 QRST
```

When an explicit opcode encoding is specified for designer-defined TIE instructions in the `Inst` slot, it should be a sub-opcode of the CUST0 or CUST1 opcodes, except when the opcode represents a load or store instruction. Load/Store instructions are generally encoded as sub-opcodes of the predefined opcodes LSCI and LSCX as explained below. If the opcodes provided by `CUST0` and `CUST1` are not enough, the opcodes reserved for some of the Xtensa configuration options can be used for designer-defined TIE instructions. However, doing so makes the TIE incompatible with an Xtensa processor configured with that option. Certain opcodes that are unallocated in the Xtensa ISA can also be used for TIE instructions. However, the unallocated opcode space may be used by various coprocessor packages. It may also be used for new Xtensa configuration options that may become available in future releases. Thus, use of opcodes outside of the `CUST0` and `CUST1` opcode space results in less robust TIE descriptions, which may be incompatible with current or future Xtensa configuration options and coprocessor packages. The current Xtensa processor configuration options and opcode maps are fully described in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*. Consider defining 32-bit or 64-bit wide instructions if there is no more opcode space in the `Inst` slot.

Designer-defined load/store instructions are generally defined as sub-opcodes of the predefined opcodes `LSCI` and `LSCX`. Load/Store instructions with immediate addressing may use the LSCI opcode space, which provides room for an 8-bit immediate field. Indexed load/store instructions may use the LSCX opcode space.

33.2 Examples

The following example shows how to define two new opcodes, `acs` and `adsel`, without specifying the opcode encoding. These opcodes will be automatically encoded by the TIE compiler in the `Inst` slot.

```
opcode acs     Inst
opcode adsel   Inst
```

The exact field encoding in this case is determined by the TIE compiler, based on the instruction operands and available bits of the instruction word.

The following example defines the same two instructions, but with the encodings specified by the TIE developer. Note that the definition uses the `CUST0` reserved opcode space provided by the Xtensa ISA.

```
opcode acs     op2=4'b0000 CUST0
opcode adsel   op2=4'b0001 CUST0
```

These opcode specifications work because the following `field` and `opcode` sections are part of the core Xtensa ISA specification. For little-endian byte ordering, define the fields as:

```
field op0 Inst[3:0]
field op1 Inst[19:16]
field op2 Inst[23:20]
opcode QRST op0=4'b0000
opcode CUST0 op1=4'b0110 QRST
```

Putting these two parts together — the predefined `field` and `opcode` specifications shown above, and the designer-defined specifications for the `acs` and `adsel` opcodes — the TIE compiler generates the following instruction decoding logic to detect the `acs` and `adsel` instructions, respectively:

```
Inst[23:0] = 0000 0110 xxxx xxxx xxxx 0000
Inst[23:0] = 0001 0110 xxxx xxxx xxxx 0000
```

It is possible when specifying the opcode encoding to use `x`'s (don't-cares). For example:

```
opcode mode op2=4'b01xx CUST0
```

In this case, the instruction bits corresponding to the `x`'s are not considered as part of the opcode and can therefore be used for other purposes. The only purpose of using `x`'s is to simplify the descriptions. For instance, the above opcode example is equivalent to the following description without using `x`'s:

```
field op2_h op2[3:2]
opcode mode op2_h=2'b01 CUST0
```

34. Instruction Operand (operand) Sections

Instruction operands are defined with `operand` description sections. Operands are register values or immediate constants that instructions operate upon. For instructions that are defined with the `operation` construct, it is not necessary for the designer to create operands. The TIE compiler can automatically generate operands based on the operation arguments as described in Chapter 7, “Instruction Operation (`operation`) Sections” on page 37. Previous versions of the TIE compiler required designers to define operands for use in `iclass` and `reference` description sections.

Instruction operands are created from instruction fields. The field can either be automatically picked by the TIE compiler, or manually defined. Instructions to define a field are in Chapter 32, “Instruction Field (`field`) Sections” on page 253. Alternatively, it is possible to use a predefined field (as listed in the *Instruction Formats and Opcodes* chapter of the *Xtensa Instruction Set Architecture (ISA) Reference Manual*) to create an operand. The field used to create an operand represents the bits of the instruction word used to encode the operand.

Register operands are typically created using a field to index into the register file. Thus, the `field` provides the address of the register, and the register contents correspond to the `operand`. Because of this, register operands are always as wide as the register file from which they are created.

Operands that are immediate constants can be created in multiple ways. Fields can be used to create more complex immediate operands as explained below. Immediate operands can be created to represent values from an `immediate_range`. Refer to Chapter 2, “Immediate Range (`immediate_range`) Sections” on page 9 for more information. Immediate operands can be created to represent values from a previously defined constant table. Constant tables are defined with `table` sections, as described in Chapter 3, “Constant Table (`table`) Sections” on page 11. In the case of table operands, the field provides the index into the table, and the table value corresponds to the operand. Immediate operands are always treated as 32-bit constants. If they are not explicitly defined as 32 bits wide, the definition is zero-extended to 32 bits. It is illegal to define immediate operands that are wider than 32 bits.

34.1 Operand Syntax

```
operand-def ::= operand name (field | *) (type | dec-enc-expr)
name ::= a unique identifier
field ::= name of a previously defined field
* ::= symbol used to indicate auto generated field
type ::= (regfile | table |immediate_range | operand_sem)
dec-enc-expr ::= decoding_expr encoding_expr [hw_decoding_expr]
regfile :: name of a previously declared regfile
table ::= name of a previously declared table
operand_sem ::= name of a previously declared operand semantic
immediate_range ::= name of a previously declared immediate range
decoding_expr ::= {expression}
encoding_expr ::= {expression}
hw_decoding_expr ::= {field}
```

name is a unique identifier that can later be used to reference the operand. *field* is the name of a previously defined field. It is possible to use the symbol * instead of explicitly specifying the field. This indicates to the TIE compiler that it should automatically create an appropriately sized field that can be used to represent the operand being defined.

A light-weight operand can be defined by specifying the name of an *operand_sem* in as its type. operand_sem describes the computation portion of an operand, and can be shared among multiple operands that perform the same computation. Detailed description of operand_sem can be found at Chapter 35, “Operand Semantic (operand_sem) Sections” on page 269.

To define a register file operand, the name of that *regfile* is used as the *type* of the operand. Similarly, to define a *table* or *immediate_range* operand, the corresponding name is used as the *type* of the operand. The *type* argument is used as a convenient and simple way to indicate how the field should be used to create the operand. Alternatively, it is possible to use a decoding and encoding expression to indicate this information in a more explicit, but verbose manner.

decoding_expr is a computation defining how the *field* is decoded into a value used for computation. For immediate operands, the decoded value is a 32-bit integer constant. It is illegal to specify a decoding expression wider than 32 bits, and if the width of the expression is narrower than 32, the operand value is zero-extended to 32 bits. For register operands, the decoded value is the value read from the register file entry addressed by the field.

encoding_expr is a computation defining how a constant is encoded in the *field*. The encoding expression is required for immediate operands, and it describes how to pack an operand value into the field. Encoding is not allowed for register operands.

The optional *hw_decoding_expr* is a computation defining how the *field* is decoded into a value that is made available to the instruction semantic in hardware. When this optional computation is specified, the *decoding_expr* only refers to the decoding as used by the software tools, while the *hw_decoding_expr* is the decoding expression as used in hardware. This construct is typically used for immediate operands, to pass the raw value of the *field* to the semantic, instead of the expanded, 32-bit operand. The use of this construct is illustrated with an example in Section 34.4 on page 265.

`expression` is described in Section 28.3 “Expressions” on page 218. Valid variables for a decoding expression are *field* names and `table` names. Valid variables for an encoding expression are `table` names and `operand` names.

When defining an operand using a decoding expression, the *field* must be explicitly specified. When defining an operand using the `regfile`, `table`, or `immediate_range` name, it is possible to either explicitly specify a field, or allow the TIE compiler to automatically select an appropriately sized field. Note that under some circumstances, the TIE compiler may not be able to pick the field automatically. This typically happens when it cannot identify enough unused bits in the instruction word to represent the field.

34.2 Example: Immediate Operands

The following example defines an operand, `imm3x2`, that can take any value in the range -8 to 6, incrementing in steps of two. The field used to encode this operand is not specified, and is thus automatically chosen by the TIE compiler.

```
immediate_range imr0 -8 6 2
operand imm3x2 * imr0
```

Such an operand can also be created by explicitly specifying the field and the decoding and encoding expression. Note that eight different values need to be represented, so a 3-bit field named `imm3` is picked and the operand is specified as follows:

```
field imm3 Inst[23:21]
operand imm3x2 imm3
     {{{28{imm3[2]}}, imm3, 1'b0}}
     {imm3x2[3:1]}
```

The decoding expression that determines the value of the `imm3x2` operand is specified as a concatenation of one zero bit, the 3-bit field `imm3` and 28 replications of the sign bit of `imm3`. This creates a 32-bit, sign-extended value equal to the value of `imm3` times two. The encoding logic does the inverse computation by extracting 3 bits starting from bit 1.

The only input used in the decoding expression is the field name imm3. The only input used in the encoding expression is the operand name, imm3x2. The imm3x2 operand is computed from the decoding expression, with imm3 as the input. A given value of the imm3x2 operand can be packed into the imm3 field using the encoding expression.

34.3 Example: Constant Table Operands

An immediate operand can also be created to represent the values shown in the following table. The operand prime_s can take any of the values specified in the table prime. The field used to encode the operand is not specified, and hence automatically picked by the TIE compiler. In the case of a table operand, the field encodes the index into the table, while the operand refers to the table value.

```
table prime 8 16 {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53
}
operand prime_s * prime
```

This table operand can also be created by explicitly specifying the field and the decoding and encoding functions as shown in the following table. Because the table has 16 entries, and the field provides an index into the table, choose the 4-bit, predefined field s. The encoding block describes how the assembler encodes a value in the field s:

```
table prime 8 16 {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53
}
operand prime_s s {
  prime[s]
} {
  prime_s == prime[0]  ? 4'b0000 :
  prime_s == prime[1]  ? 4'b0001 :
  prime_s == prime[2]  ? 4'b0010 :
  prime_s == prime[3]  ? 4'b0011 :
  prime_s == prime[4]  ? 4'b0100 :
  prime_s == prime[5]  ? 4'b0101 :
  prime_s == prime[6]  ? 4'b0110 :
  prime_s == prime[7]  ? 4'b0111 :
  prime_s == prime[8]  ? 4'b1000 :
  prime_s == prime[9]  ? 4'b1001 :
  prime_s == prime[10] ? 4'b1010 :
  prime_s == prime[11] ? 4'b1011 :
  prime_s == prime[12] ? 4'b1100 :
  prime_s == prime[13] ? 4'b1101 :
  prime_s == prime[14] ? 4'b1110 :
                         4'b1111
}
```

34.4 Example: Use of Decoding Expression for Hardware

For immediate operands, you have to specify a decoding expression and an encoding expression. The encoding expression specifies how the operand gets encoded as a field in the instruction word. The decoding expression specifies the reverse computation; to convert the field into the operand. There are certain situations in which it is desirable to specify different decoding expressions for use by the software tools and for use in the hardware implementation. For example, you may want the raw field in the hardware implementation, instead of the expanded 32-bit immediate operand. The operand construct allows for this scenario as illustrated in the following example.

Consider two instructions, `INST1` and `INST2`, both of which use the 4-bit `s` field to encode an immediate operand that represents values from -8 to 7. They perform the same computation.

```
operand myop1 s {{{28{s[3]}}, s}} {myop1[3:0]}
operand myop2 s {{{28{s[3]}}, s}} {myop2[3:0]} {s}

operation inst1 {out AR a, in AR b, in myop1 c} {} {
     assign a = b + c;
}

operation inst2 {out AR a, in AR b, in myop2 c} {} {
     assign a = b + {{28{c[3]}}, c};
}
```

For both `INST1` and `INST2`, the assembly syntax of the instruction uses values in the range -8..7 to specify the immediate operand. This is dictated by the decoding expression in the operand definition. Operand `myop2` specifies a hardware decoding expression for its operand, which is equal to the field itself. Thus in the reference description of instruction `INST2`, operand `myop2` is a 4-bit value equal to the value of the `s` field. Conversely, operand `myop1` does not specify a hardware decoding expression. Thus in the reference description of instruction `INST1`, operand `myop1` is a 32-bit immediate value as defined by its decoding expression. In the generated Verilog file, the entire 32 bits of operand `myop1` are flopped from R to E in the semantic for `INST1`, while only 4 bits of operand `myop2` are flopped from R to E in semantic for `INST2`. In `INST1`, the sign extension of field `s` happens in R stage, while in `INST2`, the sign extension happens in E stage. Thus, `INST2` uses less flops than `INST1`. The choice of `INST1` or `INST2` depends on the timing analysis of the logic in R and E stages.

In the current release, the hardware decoding expression must be the field without any computation.

34.5 Example: Register Operands

Consider a register file, VR, which is 128 bits wide and has sixteen entries. The following example defines an operand `vrr`, which represents a register of this register file. The field used to encode the operand is automatically chosen by the TIE compiler.

```
regfile VR 128 16 v
operand vrr * VR
```

The same operand can be defined with an explicitly chosen field and a decoding expression. The register file has sixteen entries, so a 4-bit field is needed to address it. In the example below, choose the predefined `r` field for this purpose. The decoding logic must be of the form *R[F]*, where `R` is a previously defined register file name and `F` must be the field name. Encoding logic is not allowed in register operand definitions.

For example:

```
regfile VR 128 16 v
operand vr r { VR[r] }
```

34.6 Xtensa Core Register Operands

The Xtensa core address register file (AR regfile) can be accessed in TIE code using a set of predefined register operands. There are three such operands named `arr`, `ars`, and `art`. These operands are defined using the `r`, `s`, and `t` fields respectively to index into the AR register file. Cadence recommends to not create additional operands to access the AR regfile.However, if you use the `operation` construct and do not specify the operand for an operation argument that accesses the AR register file, the TIE compiler will generate additional operands.

The Xtensa processor also has an option to use Boolean registers (see the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details). The base definition for Boolean registers is a 1-bit 16-entry register file with a name BR and short name `b`. The Boolean register file can be accessed in TIE code using a set of predefined register operands `br`, `bs`, and `bt`. These operands are defined using the `r`, `s`, and `t` fields respectively to index into the register file.

The Boolean register file can also be viewed as a 2-bit eight-entry register file, 4-bit four-entry register file, 8-bit two-entry register file, or 16-bit one-entry register file. Correspondingly, the predefined operands `br2`, `bs2`, and `bt2` access it as a 2-bit eight-entry register file; operands `br4`, `bs4`, and `bt4` access it as a 4-bit four-entry register file; operands `br8`, `bs8`, and `bt8` access it as a 8-bit two-entry register file; and operands `br16`, `bs16`, and `bt16` access it as a 16-bit single-entry register file. The operands `b[r,s,t]2` need only three bits to encode, operands `b[r,s,t]4` need only two bits to encode, and operands `b[r,s,t]8` need only one bit to encode. The operands `b[r,s,t]16` do not need any bits to encode. Users can create additional operands to the BR regfile as well as different views of the register file.

35. Operand Semantic (operand_sem) Sections

Operand semantic construct describes the computation portion of an operand. Multiple operands can share the same operand semantic. Its purpose is to separate the computation portion of an operand from the field of an operand. This separation makes the operand construct light weight. If the operand semantic construct is not specified, the TIE compiler rewrites the operand constructs to use the operand semantic construct.

35.1 Operand_sem Syntax

```
operand-sem-def ::= operand_sem name type {out_name, out_width}
{in_name, in_width} (reg-statements | dec-enc-expr)
name ::= a unique identifier
type ::= (regfile | immediate)
regfile :: name of a previously declared regfile
out_name ::= an identifier indicating the output of the statements or
dec-enc-expr
out_width ::= an integer indicating the width of the output
in_name ::= an identifier indicating the input of the statements or
dec-enc-expr
in_width ::= an integer indicating the width of the input
dec-enc-expr ::= decoding_expr encoding_expr [operand_hw_decode]
reg-statements :: {statements}
decoding_expr ::= {expression}
encoding_expr ::= {expression}
```

name is a unique identifier that can later be used to reference the operand semantic. If the operand semantic is referred in a register file operand, its *type* is the register file name. If the operand semantic is referred in an immediate or table operand, its *type* is a keyword **immediate**. *out_name* is an identifier to indicate the output of the operand semantic that is used in the statements or the decoding/encoding expression. *out_width* is the width of the *out_name*. Similarly, in_name is an identifier to indicate the input of the operand semantic that is used in the statements or the decoding/encoding expression. in_width is the width of the *in_name*, which is the same as the field width of the operand it is referred in.

To define an operand semantic for a register file operand, the name of that *regfile* is used as the *type* of the operand semantic. The *out_width* is the same as the field width of the operand it is referred in. Only *reg-statements* is allowed in the description, which must be a direct assignment from *in_name* to *out_name*.

To define an operand semantic for an immediate or table operand, the keyword **immediate** is used as the *type* of the operand semantic. The *out_width* is always 32. Only *dec-enc-expr* is allowed in the description. *decoding_expr* is a computation defining how the *in_name* is decoded into a value used for computation. *encoding_expr* is a computation defining how a constant is encoded in the *out_name*. The optional **operand_hw_decode** indicates that the *in_name* is used directly in the semantic computation. When this optional computation is specified, the *decoding_expr* only refers to the decoding as used by the software tools. This construct is used to pass the raw value of the *field* of an operand to the semantic, instead of the expanded, 32-bit operand.

`expression` is described in Section 28.3 “Expressions” on page 218. Valid variables for a decoding expression are *field* names and `table` names. Valid variables for an encoding expression are `table` names and `operand` names.

35.2 Examples

The following example illustrates the usage of an operand semantic for multiple immediate operands.

```
operand_sem opnd_sem_12x8 immediate {out_12x8, 32} {in_12x8, 12}
                                {in_12x8 << 32'h3} {out_12x8[14:3]}
operand uimm12x8 imm12 opnd_sem_12x8
operand fimm12x8 fimm12 opnd_sem_12x8
```

The following example illustrates the usage of an operand semantic with operand_hw_-decode. In this example, the number of flops staged from R stage to E stage is 4, instead of 32.

```
operand_sem opnd_sem immediate {o, 32} {i, 4} {{{28{i[3]}}, i}}
                                {o[3:0]} operand_hw_decode
operand opnd s opnd_sem
operation myaddi {out AR a, in AR b, in opnd o} {} {
        assign a = b + {{28{o[3]}}, o};
}
```

The following example illustrates the usage of an operand semantic for multiple register file operand.

```
regfile XR 32 16 xr
operand_sem XR_sem XR {o, 4} {i, 4} { assign o = i; }
operand xrs s XR_sem
operand xrt t XR_sem
operand xrr r XR_sem
operation XRadd {out xrr a, in xrs b, in xrt c} {} {
     assign a = b + c;
}
```

36. Operand Map (operand_map) Sections

Early in the design cycle, designers focus on the functionality and performance of TIE instructions, caring less for the encoding of the instruction and its operands. Thus, the operation definition is usually written using register file names or immediate range names in an operation's argument list. When such a TIE file is compiled, the TIE compiler does automatic operand assignment for these TIE operations. The TIE compiler may use different operand assignments every time the TIE is compiled, especially if there are changes in the TIE file or in the Xtensa configuration against which it is compiled.

Late in the design cycle, the designer may want to "freeze" the operand assignment so that it is fixed and does not change. This required defining instructions fields (described in Chapter 32) and operands based on these fields (as described in Chapter 34). One option is to then modify the operation definition to use these operands in the argument list, but this involves making many error prone changes to the TIE code. It also requires every use of an operand in different operations to share the same name. This may be inconvenient when the operations have already been written with different names. The `operand_map` construct can instead be used to achieve the same objective. The `operand_map` construct associates the operation arguments of an operation with operands, thus specifying the encoding of the operands without having to rewrite the TIE code.

Note that in order to fully specify the encoding of the TIE operation, in addition to the `operand_map`, you will need to specify the opcode encoding using the opcode construct described in Chapter 33.

36.1 Operand Map Syntax

```
operand-map-def ::= operand_map operand-name operation-name
                                operation-arg-name
operand-name ::= a previously defined operand name
operation-name ::= a previously defined operation name
operation-arg-name ::= an argument name appeared in the argument list
                       of the operation
```

`operand-name` is the name of a previously defined operand. For details on `operand` construct, see Chapter 34, “Instruction Operand (`operand`) Sections” on page 261. `operation-name` is the name of a previously defined operation. For details on `operation` construct, see Chapter 7, “Instruction Operation (`operation`) Sections” on page 37. `operation-arg-name` is the name of an argument which appears in the argument list of the operation.

36.2 Example

Consider the following piece of TIE code which defines the operation `foo` using register file names (`XR`) and immediate range names (`immed`). When this code is compiled, the TIE compiler selects appropriate instruction fields to encode the two register and one immediate operand of the instruction.

```
regfile XR 16 16 xr
immediate_range immed 0 15 1
operation foo {out XR a, in XR b, in immed imm} {} {
     assign a = b + imm;
}
```

Later in the design cycle, if you want to explicitly specify the encoding for these operands, you can define the operands and associate them with the operation arguments using the `operand_map` construct as illustrated below. Note that if you want to fully specify the encoding, you also need to specify the opcode encoding (using the `opcode` construct) in addition to specifying the `operand_map`.

```
/* original TIE code */
regfile XR 16 16 xr
immediate_range immed 0 15 1
operation foo {out XR a, in XR b, in immed imm} {} {
     assign a = b + imm;
}

/* code to specify encoding of foo operation */
opcode foo op2= 4'b0000 CUST0
operand xrt t {XR[t]}
operand xrr r {XR[r]}
operand immeds s {{28'b0, s}} {immeds[3:0]}

operand_map xrt foo a
operand_map xrr foo b
operand_map immeds foo imm
```

37. Length (length) Sections For LX cores

Instruction length is defined as the number of bits required to encode an Xtensa processor instruction. All Xtensa processor instructions have a fixed, predefined length. The base Xtensa processor instruction set consists of various 24-bit and 16-bit wide instructions. Designer-defined instructions can be 24- to 128-bits wide with an increment of 8 bits. The widest width must be the same as the maximum instruction size supported by the configuration. The TIE language requires that the underlying machine has an instruction buffer that is large enough to hold the longest instruction supported in a particular configuration. This instruction buffer is represented by the TIE keyword `InstBuf`. The width of `InstBuf` is the maximum instruction size supported by the configuration.

A `length` statement specifies the decoding logic for determining a particular instruction length based on values of certain bits in the instruction buffer. `length` sections are only required in a TIE description when you want to specify the decoding bits. They are optional if you would like the TIE compiler to automatically assign these bits for you.

37.1 Length Syntax

```
length-def ::= length name length decoding-expr
name ::= a unique identifier
length ::=  an integer describing the length
decoding-expr ::= {InstBuf[from:to] == sized-constant}
from ::= integer bit index
to ::= integer bit index
```

name is a unique identifier that can later be used to reference the length. *length* specifies the instruction length in bits, and is currently restricted to the values from 32 to 128 in stripes of 8. *decoding-expr* is an expression that specifies the decoding logic for determining the instruction length. Only use the first 8 bits of `InstBuf` in the decoding expression. The specific bits depend on the endianness of the Xtensa processor configurations:

- For little-endian configurations, it is `InstBuf[7:0]`.
- For big-endian configurations having a maximum instruction size of N-bits, it is `InstBuf[N-1:N-8]`. Thus, for 32-bit configurations, it is `InstBuf[31:24]`, for 64-bit configurations, it is `InstBuf[63:56]`, and for 128-bit, it is `InstBuf[127:120]`. If multiple wide lengths are specified on the processor, they all need to use the first 8 bits of `InstBuf`. Preprocessing variable `$IsaMaxInstructionSize` is defined to describe the maximum instruction size in the number of bytes. Thus, `InstBuf[N-1:N-8]` can be described as `InstBuf['$IsaMaxInstructionSize * 8 - 1':'$IsaMaxInstructionSize * 8 - 8']`.

Multiple length statements can have the same instruction length. The designers can define multiple length statements of different names, but with the same instruction length and the same decoding expression. Those length statements are identical.

37.2 Predefined Xtensa Processor Instruction Lengths

The following length statements describe the predefined 24-bit and 16-bit Xtensa processor instruction lengths in a little-endian configuration:

```
length l24     24   {InstBuf[3] == 1'b0}
length l16a    16   {InstBuf[3:2] == 2'b10}
length l16b    16   {InstBuf[3:1] == 3'b110}
```

For little-endian Xtensa processor configurations that include the 16-bit instructions (density option), the available encoding space is `{InstBuf[3:0] == 14}` and `{InstBuf[3:0] == 15}`. For little-endian Xtensa processor configurations that do not include the 16-bit instructions, the entire opcode space corresponding to `{InstBuf[3] == 1}` is available for specifying the instructions' length. The opcode space corresponding to `{InstBuf[3] == 0}` is not available in any Xtensa configuration. Similarly, for big endian Xtensa processor configurations that include the 16-bit instructions, the available encoding space is `{InstBuf[N-1: N-4] == 14}` and `{InstBuf[N-1: N-4] == 15}`. For big endian Xtensa processor configurations that do not include the 16-bit instructions, the entire opcode space corresponding to `{InstBuf[N-1] == 1}` is available.

Note that 16- and 24-bit length statements are not allowed in designer-defined TIE code. The above definitions are for the Xtensa core ISA, and are shown here for illustration only. You can create 24-bit (non-FLIX) TIE instructions in any Xtensa configuration without having to define a length. 16-bit TIE instructions are not supported.

37.3 Examples

The following statement defines a 64-bit instruction length for little endian Xtensa processor configurations:

```
length l64 64 {InstBuf[3:0] == 14}
```

The following statement defines a 32-bit instruction length for big endian Xtensa processor configurations with a maximum instruction width of 64:

```
length l32a 32 {InstBuf[63:60] == 15}
```

The following statement defines a 96-bit instruction length for little-endian Xtensa processor configurations. The number of bits to decode the length is 6 bits.

```
length l96 96 {InstBuf[5:0] == 6'b001110}
```

38. Instruction Slot (slot) Sections

A slot is a subset of bits of a FLIX instruction that encodes one operation. A typical FLIX instruction has several slots and therefore is capable of performing several operations in parallel.

Slot sections are only required in a TIE description when the designer wants to specify the bits used to encode a slot in a format. They are optional for slots where the TIE compiler automatically assigns the bits.

38.1 Slot Syntax

```
slot-def ::= slot name (sub-range | {sub-range_concat})
name ::= a unique identifier
sub-range ::= format-name | format-name[from:to]
format-name ::= name of a previously defined format
from ::= integer bit index
to ::= integer bit index
sub-range_concat ::= sub-range [, sub-range]*
```

name is a unique identifier for the slot. *format-name* must be the name of a previously defined format. When using a concatenation of sub-ranges, all the format names used in a slot definition must be the same. All bit indices must be greater than or equal to zero, and less than the length of the referenced format. The length of a format is the number of bits that remain in the instruction word after accounting for the bits needed to encode the length and the format. A bit in a format cannot belong to more than one slot.

Note that slot names have to be declared in the slot-list of the format that the slot belongs to. Thus, the only purpose of the slot declaration is to specify the bits of the format that make up the slot. If you do not wish to specify the slot encoding, simply omit the slot declaration, as it is optional. When a slot name is listed in the slot-list of a format, but there is no slot declaration for it, the TIE compiler automatically assigns the bits corresponding to that slot.

Unlike the names for most other sections in TIE descriptions, a slot name can be defined multiple times, in the same format or in different formats. When a slot is defined multiple times, all the opcodes in that slot are identically replicated in all the instances of that slot. The only restriction is that all the definitions for a given slot name must have the same size (number of bits).

38.2 Examples

The following example defines three slots in a 64-bit, little-endian configuration. In this example, the designer explicitly specifies the bits corresponding to each slot. Bits [3:0] of the instruction word are used to specify the length of the instruction. Thus, format f64a has 60 bits available to distribute between its slots.

```
length len64 64 {InstBuf[3:0] == 15}
format f64a len64 {slot0, slot1, slot2}
slot slot0 f64a[27:4]
slot slot1 f64a[47:28]
slot slot2 f64a[63:48]
```

In the following example, two formats are created, one containing two slots and the other containing three slots. Note that slota is defined as a concatenation of disjointed bits of format myfmta. Furthermore, note that there are two instances of slotc in format myfmtb. In this example, bits [63:60] specify the length, and bit[59] specifies the format within the length. Thus, there are 59 bits available to encode the slots within each format.

```
length mylen 64 {InstBuf[63:60] == 14}
format myfmta mylen {slota, slotb} {InstBuf[59] == 0}
format myfmtb mylen {slotc, slotc, slotd} {InstBuf[59] == 1}
slot slota {myfmta[58:50], myfmta[9:0]}
slot slotb myfmta[49:10]
slot slotc myfmtb[58:48]
slot slotc myfmtb[47:37]
slot slotd myfmtb[36:0]
```

The following example creates two slots in a 32-bit, big-endian configuration. There is no slot declaration in this example, which means that the TIE compiler will automatically select the bits corresponding to each slot. In this example, bits [31:28] of the instruction word specify the length, and bits [27:0] encode the two slots.

```
length l32 32 {InstBuf[31:28] == 14}
format f32 l32 {slot0, slot1}
```

39. Instruction Class (iclass) Sections

The iclass construct is deprecated. This section is provided for backward compatibility purpose. Please use operation construct (Chapter 7) instead.

Instruction classes, which associate opcodes with operands, are defined with iclass description sections. Instructions that are defined using operation description sections do not have an iclass section. The argument-list of an operation serves as an iclass declaration. This is described in Chapter 7, “Instruction Operation (operation) Sections” on page 37. Instructions that are not defined with operation sections are required to have iclass sections.

All instructions defined in an iclass have the same assembly format and operand usage. Before defining an instruction class, first define its opcodes and operands.

39.1 Iclass Syntax

```
iclass-def ::= iclass name
     opcode-list operand-list [state-list [interface-list] ]
name ::= a unique identifier
opcode-list ::= {opcode-name [, opcode-name]*}
opcode-name ::= a previously defined opcode name
operand-list ::= { [dir operand-name [, dir operand-name]*] }
state-list ::= { [dir state-name [, dir state-name]*] }
interface-list ::= { [dir interface-name [, dir interface-name]*] }
dir ::= in | out | inout
operand-name ::= a previously defined operand name
state-name ::= a previously defined state name
interface-name ::= a previously defined interface name
```

name is a unique identifier for the instruction class. *opcode-list* is a list of one or more previously defined opcode names. *operand-list* is a list of zero or more operand specifications, each of which defines whether the operand is a source (in), destination (out), or both (inout) for the computation performed by the instruction. The order of operands in *operand-list* is significant only in that it defines the order of the operands in the assembly (or C/C++ intrinsic) syntax for the instruction. *state-list* is a list of zero or more processor state specifications, each of which defines whether the state is read, written, or both for the computation performed by the instruction. *interface-list* is a list of zero or more interface signal specifications, each of which defines whether the interface is read, written, or both by the instructions of the iclass.

For information about how opcodes are defined, see Chapter 33, “Instruction Opcode (opcode) Sections” on page 257. For information about how operands are defined, see Section 34 “Instruction Operand (operand) Sections” on page 261. For information on how interface signals are declared, see Chapter 6, “Interface Signals (interface) Sections” on page 29. For information about how states are defined, see Chapter 4, “Processor State (state) Sections” on page 15.

39.2 Example 1

The following example defines two opcodes, acs and adsel, which take two register operands, art and ars, as sources, write a result to a third register operand, arr, and write a state ERROR:

```
opcode   acs      op2=4'b0000   CUST0
opcode   adsel    op2=4'b0001   CUST0

iclass   viterbi  {adsel, acs}   {out arr, in art, in ars} {out ERROR}
```

(The predefined arr, art, and ars operands reference the AR register file, as defined in Section 34.5 “Example: Register Operands” on page 266.) Following is an example of the assembly language syntax resulting from this declaration:

```
acs a4, a5, a6
```

This example invokes the acs instruction to read AR[5] and AR[6] and write AR[4]. (The syntax for an operand referencing AR[*n*] is “a*n*” because “a” is the short-name defined for the AR register file. See Chapter 5, “Register File (regfile) Sections” on page 21). State registers, such as ERROR in this example, are implicit operands and are not reflected in the assembly (or C/C++ intrinsic) syntax of the instruction.

39.3 Example 2

The following example defines an iclass for a 64-bit immediate memory load instruction for a designer-defined register file XR:

```
opcode load r=4'b0000 LSCI   // declare opcode
regfile XR 64 16 x           // a 64-bit register file
operand xrt t { XR[t] }      // a register operand
iclass load {load} {
  out xrt, in ars, in imm8   // instruction operands
} {} {
  out VAddr, in MemDataIn64  // interface signals
}
```

Load instructions use the interface signal VAddr to specify the memory address of the load. The data read from memory is returned on the interface signal MemDataIn64 in this example. Interface signals are also implicit operands, and are not reflected in the assembly (or C/C++ intrinsic) syntax of the instruction. Thus, an example assembly language syntax of the above instruction would be:

```
load x8, a7, 0
```

40. Instruction Reference (reference) Sections

reference construct is deprecated. This section is provided for backward compatibility purpose. Please use operation construct (Chapter 7) instead.

The purpose of a reference section is to clearly describe the behavior of an instruction, without concern for implementation efficiency. When a new instruction is defined using the operation construct, the computation section of the operation provides the same functionality as a reference section. Hence, instructions created using operation sections cannot have a reference section. Because of its many advantages, Tensilica recommends writing TIE descriptions using operation sections instead of reference sections. Refer to Chapter 7, “Instruction Operation (operation) Sections” on page 37 for more information on the operation construct.

Unlike a semantic description, a reference section describes the behavior of a single instruction. When an instruction has both a reference description and a semantic description, the semantic description is used for the implementation by the TIE compiler. The TIE compiler supports formal equivalence checking between reference and semantic descriptions. Several TIE instructions may share a single semantic block for efficient implementation, but the reference description cannot be shared between instructions.

40.1 Reference Syntax

```
reference-section ::= reference opcode-name {computation}
opcode-name ::= a previously defined opcode name
computation ::= see Chapter 28
```

opcode-name is a previously defined opcode name. The instruction behavior is specified in *computation*. For details on the computation section, see Chapter 28, “Computation Sections” on page 215. With the exception of the opcode name, the valid variables for a computation in a reference section are the same as those for a semantic section.

40.2 Example

This example shows the reference description of two TIE instructions, ADD8_4 and SUB8_4. The first instruction adds four 8-bit values that are packed into one 32-bit register and the second instruction performs a subtraction on the same data type. The following reference descriptions are written for ease of readability, without regard to hardware implementation considerations.

```
reference ADD8_4 {
   assign arr = { ars[31:24] + art[31:24],
                  ars[23:16] + art[23:16],
                  ars[15:8]  + art[15:8],
                  ars[7:0]   + art[7:0] };
}

reference SUB8_4 {
    assign arr = { ars[31:24] - art[31:24],
                   ars[23:16] - art[23:16],
                   ars[15:8]  - art[15:8],
                   ars[7:0]   - art[7:0] };
}
```

A. Guidelines and Restrictions for FLIX TIE

This appendix lists several guidelines and restrictions that you should keep in mind when creating FLIX TIE designs. The Xtensa C/C++ compiler is aware of the restrictions listed in this appendix, and will generate code consistent with these restrictions. The assembly language programmer should pay special attention to these guidelines and restrictions.

A FLIX format consists of a combination of one or more slots. Each slot supports a set of instructions that are defined using the `slot_opcodes` construct. In this discussion, the word *operation* is used to refer to the “instructions” within a slot, and the word *instruction* to refer to the entire FLIX instruction consisting of one or more operations.

A.1 Reading and Writing State and Register Files

When creating a FLIX instruction, you cannot have operations from different slots write to the same TIE state or the same entry of a register file[1]. Doing so will result in an assembly error when you try to assemble the program. It is legal to read from the same state or register file entry in multiple slots.

Consider the following TIE description, which defines two slots with various Xtensa ISA instructions defined in those two slots:

```
format f64 64 {slot0, slot1}
slot_opcodes slot0 {ADD, AND}
slot_opcodes slot1 {SUB, XOR}
```

For this TIE description, the following are examples of legal and illegal FLIX instructions:

```
// Illegal because both slots write to a4
{ADD a4, a5, a6; SUB a4, a10, a11}

// Legal example. Reading from the same register in multiple slots is
allowed.
{AND a4, a5, a6; XOR a7, a5, a6}
```

From a programming standpoint, a FLIX instruction that reads and writes the same state or register file entry always follows the semantics that “all reads happen before any writes.” This is independent of the actual use and def stages of the register file by their respective operation.

1. States marked with `shared_or` attribute are exceptions. Details see Section 4.4 “ORing a State” on page 17

Consider the following TIE description:

```
format p64 64 {slota, slotb}
slot_opcodes slota {ADDP}
slot_opcodes slotb {SUBP}

regfile XR 64 16 x
operation ADDP {out XR a, in XR b, in XR c} {} {
     assign a = b + c;
}
schedule addp {ADDP} {
     use b 5;
     use c 5;
     def a 5;
}

operation SUBP {out XR a, in XR b, in XR c} {} {
     assign a = b - c;
}
schedule subp {SUBP} {
     use b 1;
     use c 1;
     def a 1;
}
```

Now consider the following FLIX instruction:

```
{ADDP x1, x2, x3; SUBP x2, x4, x5}
```

The ADDP instruction uses the register file entry x2 as an input, while the SUBP instruction writes to the same entry. Furthermore, the SUBP instruction has a def schedule of 1 while the ADDP instruction has a use schedule of 5. When this instruction executes, the value of x2 used by the ADDP instruction is the value prior to the execution of this instruction. It is not the value generated as a result of the execution of the SUBP instruction in `slotb` of the instruction, even though that may appear to be the case from the use/def schedule.

A.2 Shared Functions and FLIX Instructions

When creating a FLIX instruction, you cannot have operations from different slots use the same (globally) shared function in the same cycle. Doing so will result in an assembly error when you try to assemble the program. Such an instruction is legal if the shared function is used in different clock cycles by the respective operations. This is illustrated in the following example:

```
format x64 64 {slot0, slot1}
slot_opcodes slot0 {ADD64, L32I}
slot_opcodes slot1 {SUB64, BEQ}
regfile XR 64 16 x

function [63:0] addsub([63:0] a, [63:0] b, sub) shared {
     wire [63:0] tmp = sub ? ~b : b;
     assign addsub = TIEadd(a, tmp, sub);
}

operation ADD64 {out XR a, in XR b, in XR c} {} {
     assign a = addsub(b, c, 1'b0);
}

operation SUB64 {out XR a, in XR b, in XR c} {} {
     assign a = addsub(b, c, 1'b1);
}
```

In the preceding TIE code, operations ADD64 and SUB64 both use a shared function named addsub in cycle 1. Hence, the following FLIX instruction is illegal and will result in an assembly error:

```
{ADD64 x0, x1, x2; SUB64 x4, x5, x6}
```

Now consider modifying the SUB64 instruction to use the shared function in cycle 2 by writing a schedule as follows:

```
schedule sub64 {SUB64} {
     use b 2;
     use c 2;
     def a 2;
}
```

With this modification, SUB64 uses the addsub function in cycle 2. Because the ADD64 instruction uses the shared function in cycle 1, it is now legal to create a FLIX instruction with these two operations.

A TIE semantic section may describe the implementation of more than one operation. If a semantic uses a shared function, the shared function resource is considered to be in use when any operation implemented in that semantic is being executed. This is true even if one or more of the operations do not actually use the output of the shared function. Thus it is not possible to bundle any operation from such a semantic with any other operation that uses the same shared function.

A.3 Load/Store Operations in FLIX Instructions

In an Xtensa LX processor configuration with one load/store unit, all load/store instructions must be assigned to slots with slot index 0. In a configuration with two load/store units, these instructions may be assigned to slots with slot index 0 and to slots with one other slot index.

In an Xtensa configuration with two load/store units, it is possible to issue two store operations, or a store and a load operation, as part of the same FLIX instruction. In this situation, the two operations must access different locations in memory. Xtensa processor hardware detects the situation in which both operations access the same byte(s) in memory (with consideration to the `StoreByteDisable` and `LoadByteDisable` values) and takes an exception.

Load/store operations to IRAM are only allowed from slot index 0. If the address of a load or store issued from any other slot index evaluates to an IRAM address, the processor will take an exception.

A.4 Branch Operations in FLIX Instructions

It is illegal to generate a FLIX instruction in which more than one slot contains a branch (or jump) instruction. Note that it is legal to include a branch instruction in more than one slot of any FLIX format. This allows you to issue a branch instruction from any slot of the format. However, you cannot have two or more branches in the same FLIX instruction.

When assigning branch instructions to FLIX slots, Cadence recommends that you assign a branch and its inverse to the same slot. For example, if you would like to use the “Branch if Equal (BEQ)” instruction in a particular slot, it is advisable to also include the “Branch if Not Equal (BNE)” instruction in the same slot. While this is not mandatory, it is recommended because it allows the Xtensa C/C++ compiler and the assembler to generate better code.

A.5 TIE Queues and Lookups in FLIX Instructions

It is illegal to have more than one operation in a FLIX instruction that accesses the same TIE queue. This is true of both input as well as output queues. It is also illegal to have more than one operation in a FLIX instruction that accesses the same TIE lookup.

Index