cadence®

Tensilica® Instruction Extension (TIE) Language

User's Guide

For Xtensa Processor Cores

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

Issue Date:03/2017

RG-2016.5
PD-16--2002-10-02

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 RG-2017.5 release. Subsequent releases may contain updates for features in this release or additional features may be added.

- Updated Section 7.5, deleted reference to a discontinued feature.

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

- Added clarification for the retiming feature, see Section 5.8.1 and Section 5.8.2.

1. Overview

The Tensilica Instruction Extension (TIE) language is used to describe instruction set extensions for the Xtensa family of processor cores. This document describes the design methodology for extending Xtensa processors with custom instructions along with examples of main TIE features and a discussion of important TIE design considerations. For a comprehensive description of TIE constructs refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

Instruction set extensions accelerate data-intensive and compute-intensive functions within software programs that are commonly referred to as "hot spots". A hot spot is a region of code that is exercised heavily by the program and the efficiency of its implementation can have a disproportionate effect on the efficiency of the program as a whole. Figure 1–1 shows the percentages of total execution time consumed by four hypothetical functions and how TIE optimization can accelerate the hot spot functions. In this example, one hot spot function is consuming 50% of the run time of the application, and a second is consuming another 30%. When both functions are accelerated with TIE instructions, the total run time is reduced by more than half. The actual performance increase from using TIE instructions varies widely among applications and depends upon the parallelism inherent to the algorithms involved. Using TIE can increase performance by several orders of magnitude.

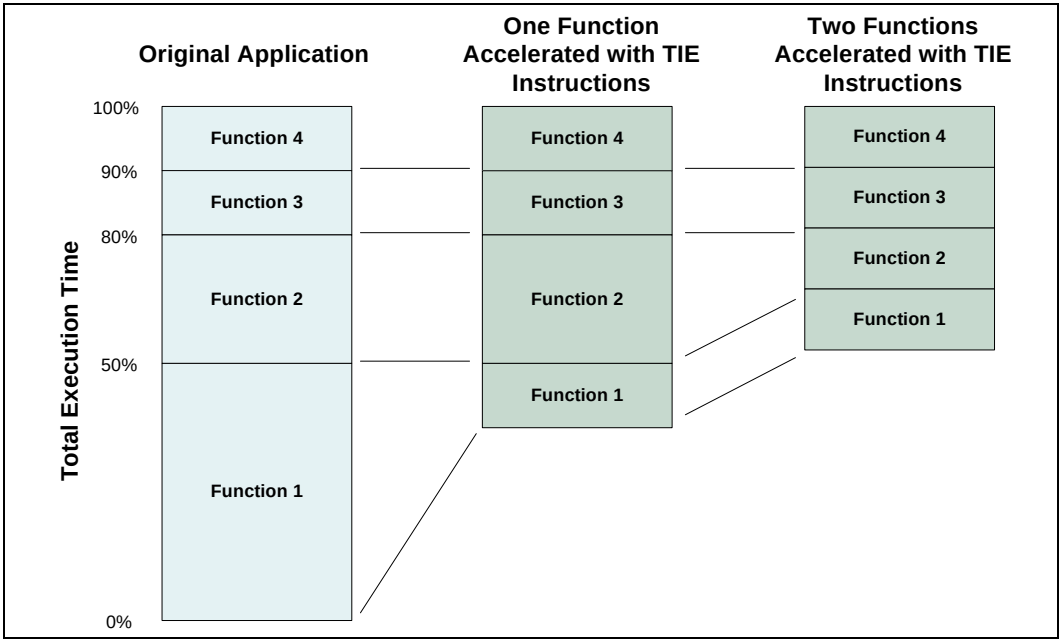

Figure 1–1. TIE Instructions Accelerate Compute-Intensive Functions

TIE descriptions do not compile into microcode or any other kind of intermediate machine language. Instead, they compile into hardware that is automatically integrated into the Xtensa processor along with all of the tools and scripts needed to develop software for it, simulate it and tape it out for production.

The purpose of this guide is to teach TIE design methodology and concepts. Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a more detailed explanation of TIE language constructs.

This guide assumes that you are familiar with the Xtensa processor architecture and the Xtensa Xplorer software development environment. Other documents most relevant to this guide are:

- *Tensilica Instruction Extension (TIE) Language Reference Manual*
- *Xtensa Instruction Set Architecture (ISA) Reference Manual*
- *Xtensa C and C++ Compiler User's Guide*
- Xtensa Xplorer documentation

2. The TIE Design Methodology and Design Tools

TIE can provide substantial performance improvements across a wide variety of applications. However, such performance gains are at the expense of additional processor gate-count, and sometimes reduced maximum operating frequency. A well-designed set of TIE instructions provides optimal performance with the least impact to processor area and operating frequency. This section provides an overview of the TIE design process and introduces the methodologies required to ensure optimal TIE designs.

2.1 Design Methodology Description

There are three main phases for a general TIE design methodology. The first phase, Functional Description, requires knowledge of the application or algorithm being executed. The other two phases, Hardware Optimization and Processor Area/Timing Optimization require knowledge of hardware optimization, and EDA implementation tools.

Following are descriptions for each of the methodology phases:

1. **Functional Description and Verification:** The first phase consists of identifying hot spots (software functions that consume the majority of processor cycles) in the application, specifying the functionality of TIE instructions to accelerate the hot spots, and verifying the functionality and performance of the application with the new TIE instructions. See Section 2.1.1 for more information about the functional description of TIE designs.
2. **Hardware Optimization**: The second phase of TIE design consists of analyzing the hardware area and timing requirements of the new instructions and making TIE optimizations to meet area and timing goals. See Section 2.1.2 for more information about optimizing TIE designs.
3. **Processor Area and Timing Optimization**: The third phase integrates the optimized TIE along with the Xtensa processor and verifies that the entire processor can meet area and timing goals. Section 2.1.3 contains information about the processor area and timing check for TIE designs.

2.1.1 Functional Description and Verification Phase

In the functional description phase, you locate hot spots in the application, specify TIE instructions that accelerate the performance of the hot spots, and verify the TIE instructions for functional correctness. Figure 2–2 is a flowchart of the functional description phase steps.

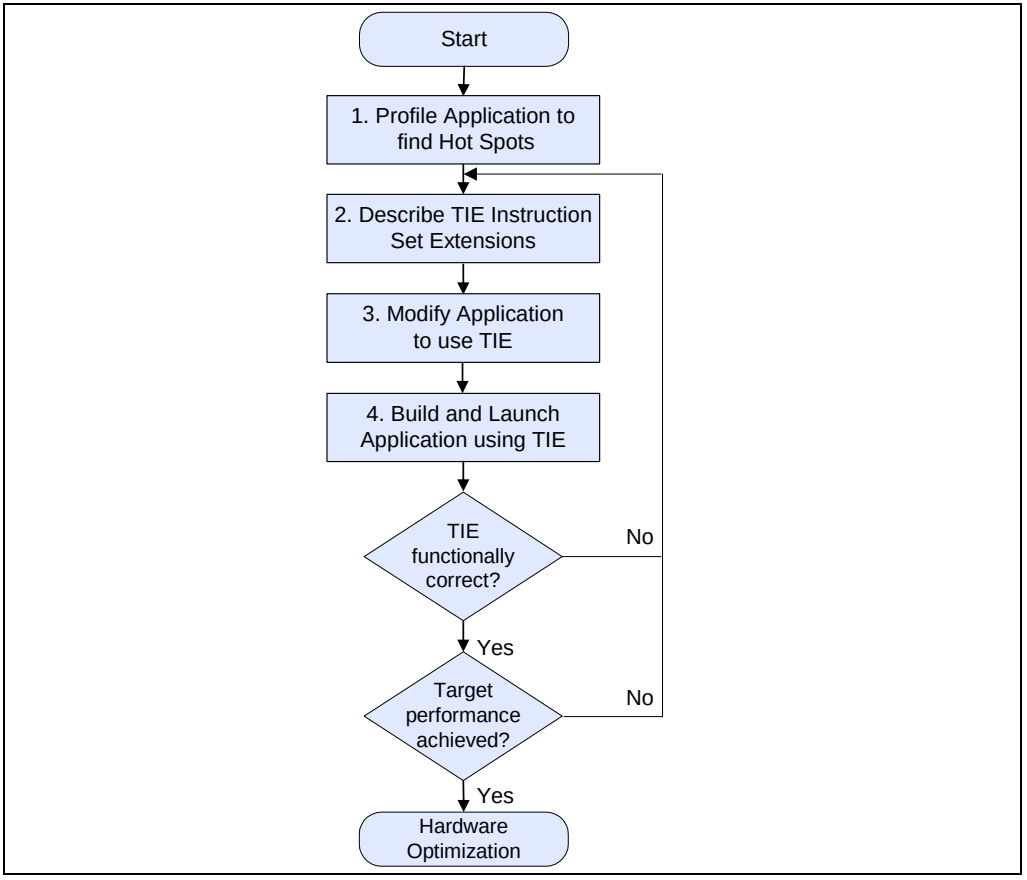

Figure 2–2. Functional Description Phase

1. Begin by running the application to obtain an execution profile. The execution profile reports how many processor cycles are spent in each function. Functions that are called frequently or require high processing bandwidth are considered hot spots and are good candidates for acceleration by developing TIE instructions.
2. Next, map the behavior of the hot spots to a set of new processor instructions. Define the new instructions using the TIE language.
3. TIE instructions are available for use by C/C++ programs through intrinsic functions and overloaded operators. Modify the application to call TIE intrinsic functions in place of the software code that originally represented the hot spots of the application.
4. After modifying the source code for TIE usage, compile and simulate the application. Verify the results of the TIE-enabled application against the results of the original source code.

5. If the results of the application using the TIE implementation do not match the original reference implementation, debug the TIE instructions. Once the application is functionally correct using the TIE implementation, check the execution profile to evaluate performance.
   - If the performance is not acceptable, enhance the TIE instructions to perform more work or add new TIE instructions until you get the desired performance.
   - If the performance is acceptable, develop additional C/C++ programs to verify correct functionality of TIE instructions under a wide variety of test cases. This step will ensure that there are no bugs in the TIE description before proceeding to the optimization phases. Once TIE instructions are functionally verified in simulation, begin the optimization phases.

2.1.2 Hardware Optimization Phase

The Xtensa tools provide a summary of the estimated hardware cost of the TIE specified in the functional description design phase. The summary provides the approximate area cost (as number of gates) for the TIE, with an error-band of ±50%. Use this estimate to get rapid feedback on the performance-area tradeoffs of a particular implementation without requiring a logic synthesis compilation. This estimate is also useful in determining relative area improvement amongst several area optimized TIE solution candidates.

This summary is useful to begin the second phase of the TIE methodology, optimization. Figure 2–3 displays a flowchart of this phase, where you optimize the functionally correct TIE for reduced area and improved timing, as follows:

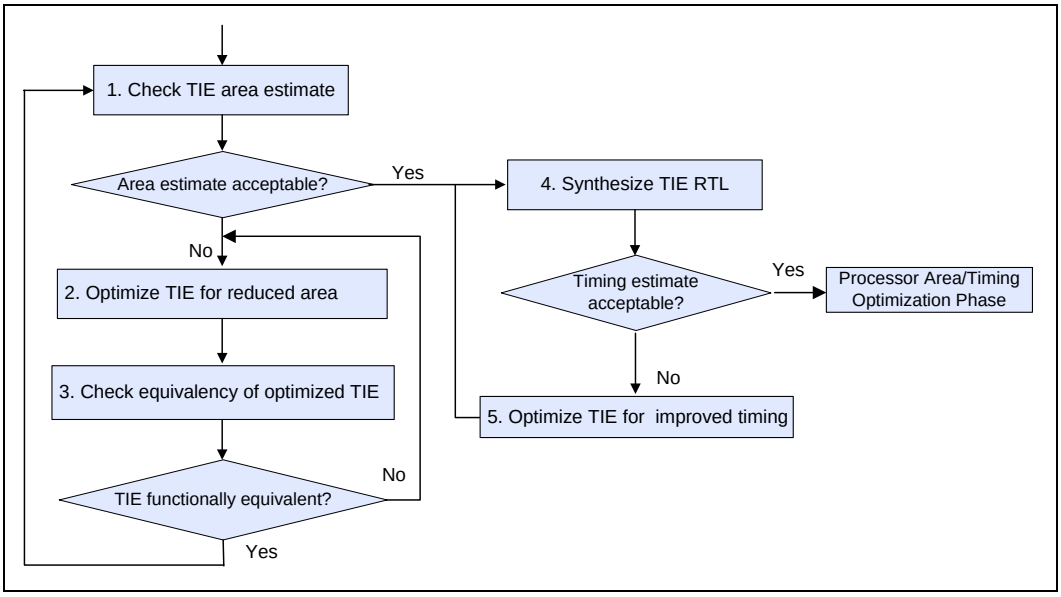

Figure 2–3. Hardware Optimization Phase

1. Check the TIE area estimate summary as described in Section 4.2 on page 26. If the area estimate is acceptable, proceed to Step 4 for the TIE Area and Timing Optimization phase.
2. Optimize the TIE instruction as described in Chapter 6, “TIE Optimization Strategies” on page 85. For example, use TIE features for logic sharing across different instructions.
3. After optimizing a TIE instruction, it is important to make sure that it provides the same behavior as the original TIE. While optimizing, bugs may be introduced to the TIE description. TIE provides a mechanism for describing an optimized version of a given TIE instruction and performing a formal equivalency check against the original version using the Incisive Conformal Equivalence checking tool from Cadence Design Systems.

   If the optimized TIE is equivalent to the initial TIE, check the area estimate again. If the TIE is not equivalent, correct it until it is equivalent. For more information about formal equivalency checking of TIE instructions, refer to Section 9.6.1.
4. After area goals are met, timing is checked. Tensilica’s TIE Compiler generates the RTL implementation for your TIE description. Design Compiler scripts are also provided so that you can synthesize the generated RTL. When synthesizing the RTL, set the timing goals to be slightly faster than your design target. This conservative measure will factor in timing degradation that may occur during place and route. For additional details on TIE synthesis, refer to Section 9.2.

   When TIE synthesis is complete, synthesis reports provide an estimate of worst-case timing for the TIE instructions alone, without considering the rest of the Xtensa processor core. Note that this method of generating a timing estimate is quick, but not 100% accurate. A more accurate timing estimate is generated in the next phase. If the timing reports show that timing goals are met, proceed to the Processor Area and Timing Optimization phase (see Section 2.1.3 “Processor Area and Timing Optimization Phase” on page 7).

   The synthesis reports will also provide a more accurate measure of the area of the TIE design. It is recommended that you correlate this number with the area estimate generated by the TIE compiler.
5. If the timing reports show that design constraints are not met, optimize your TIE instructions for improved timing. One way to improve TIE timing is to schedule the execution of a TIE instruction over multiple cycles. The Xtensa processor allows you to fine-tune TIE instructions through explicit partitioning of execution across clock cycle boundaries. Refer to Section 5.8 on page 54 for details on scheduling the execution of TIE instructions. Note that formal equivalency checking may be required, depending upon the optimizations performed in this step.

Several iterations of area and timing optimization may be necessary to obtain the desired results. However, the optimization process may reach a point of diminishing gains and it may be necessary to begin the initial TIE design phase again using a different approach to accelerate the hot spots with TIE instructions.

2.1.3 Processor Area and Timing Optimization Phase

In this phase, you integrate the optimized TIE code with the Xtensa base processor RTL. Tensilica's methodology guarantees that the processor RTL is accurately modeled in the instruction set simulator (ISS). Given that TIE functionality has been thoroughly verified in software simulation, it is not necessary to perform additional TIE functional verification on the processor RTL (note that hardware verification of the interaction between the processor and other SOC components is necessary, refer to the *Xtensa Hardware User's Guide* for further information on system verification). However, a final area and timing check of the processor RTL is required.

Figure 2–4 is a flowchart of the processor area and timing optimization phase. Check that the new RTL meets timing and area goals using more accurate estimation methods, as follows:

1. Using Xtensa Xplorer, build a new processor configuration that includes your TIE instructions. Within a few hours, the RTL for the new processor will be available.
2. Obtain detailed timing information by synthesizing the processor RTL using Design Compiler or RTL Compiler. The Xtensa package includes synthesis scripts that automatically generate timing reports. Use these reports to determine the maximum operating frequency of your processor. These reports also provide an estimate of area used to implement the processor.
3. Check that the timing and area goals are met by running the provided place and route scripts for SOC Encounter or IC Compiler. Synthesis reports do not factor routing area and signal propagation delays when providing area/timing reports. If place and route timing and area reports satisfy your design goals, the TIE design is complete.
4. If timing and area reports show that design goals are not met, optimize the parameters used by the synthesis and/or place and route scripts. The implementation scripts that are shipped with the Xtensa processor combine many implementation parameters into a single file (`CadSetup.file`, found in the `/Hardware/xda` directory of the processor installation). Optimize these parameters to improve timing and area (it may take several iterations of implementation trials to meet design goals).

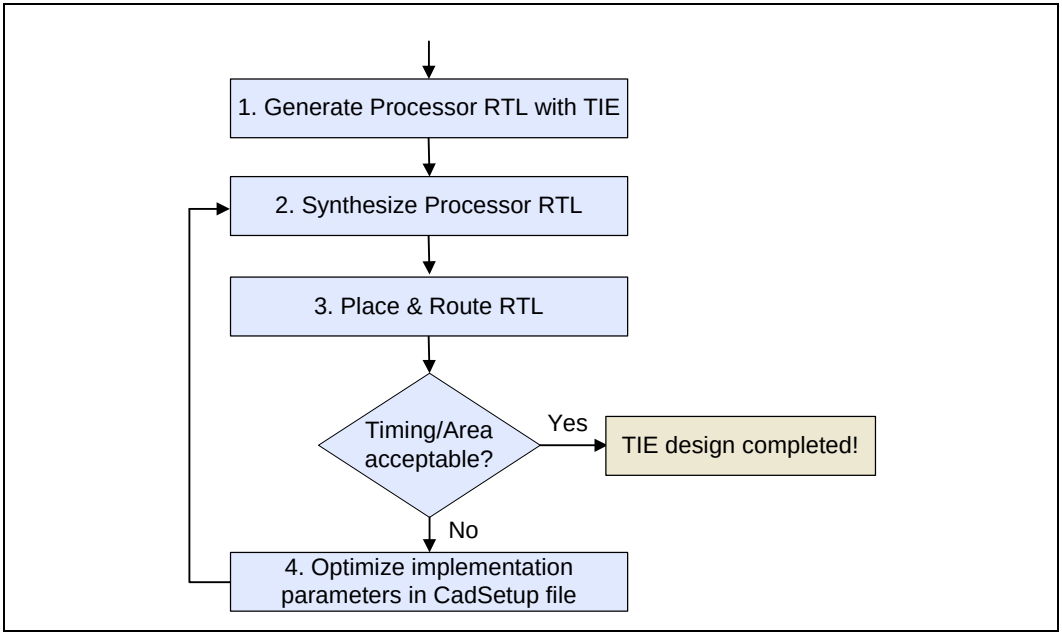

Figure 2–4. Processor Area and Timing Optimization Phase

2.2 Methodology and Design Summary

This section introduced the three phases of TIE design: Functional Description and Verification, Hardware Optimization, and Processor Area and Timing Optimization. Each phase, and each step within each phase, is key to developing a TIE design free of verification, area, and timing problems. Depending upon your design schedule and resources, your TIE design flow may differ from the exact flow shown in this section. Based upon design goals, you may choose to spend more effort in some steps and less effort on others. For example, if you have a high frequency design target, you may want to perform the processor timing check at regular intervals throughout the design cycle, instead of waiting until the end of the design cycle.

3. TIE Design Tools

Unlike a general purpose processor, the Xtensa processor hardware and supporting software tools were designed from the ground up to be highly extensible, enabling rapid development of custom processor extensions. With other processors, extending a general-purpose processor with custom instructions is a non-trivial task. There are many other considerations besides the RTL design of the instruction hardware. Integrating custom hardware with a processor requires in-depth understanding of micro-architectural details to ensure processor correctness under all data hazards, branches, and exceptions. The design and verification of pipeline control logic often requires a large proportion of the development time. Moreover, all the software development tools, such as the C/C++ compiler, assembler, debugger, profiler, and the Instruction Set Simulator (ISS), must be modified to support custom processor extensions.

The TIE design tools automate the entire process of extending the Xtensa processor, from providing a context-aware TIE editor, to automatically extending all the software development tools, to creating the hardware. The TIE design tools facilitate rapid prototyping of processor extensions, allowing the exploration of many optimization approaches in a short period of time. This section provides an introduction to TIE design tools.

3.1 The TIE Compiler

The TIE Compiler (tc) is a processor synthesis tool that extends the Xtensa processor with custom processor extensions. The TIE compiler works from a TIE source file that describes the behavior of processor extensions in the TIE language. The TIE language provides the developer with a concise way of extending the Xtensa processor to accelerate application performance. The TIE language contains constructs that describe the behavior of TIE instructions while abstracting most complex Xtensa microprocessor details. This allows the developer to focus primarily on TIE performance and optimization instead of verifying processor correctness (correct behavior of TIE instructions under all pipeline conditions such as exceptions, branches, and data hazards). The TIE developer simply describes the behavior of processor extensions using the TIE language and saves it in a TIE file (a text file with the `.tie` suffix).

The TIE compiler takes the TIE description of processor extensions and performs consistency checks to assure compatibility with the existing Xtensa processor. From the TIE description, the TIE compiler automatically generates a TIE Development Kit (TDK). The TDK contains files that allow Tensilica's Xtensa C/C++ compiler and the ISS to support the processor extensions. TIE instructions are available to software developers as assembly mnemonics or intrinsic C/C++ functions. The TDK also extends the ISS, providing cycle-accurate benchmark results when simulating code utilizing TIE. The TIE compiler generates the TDK within a matter of seconds for basic TIE, and may take up to several minutes for more complex TIE.

In addition to software components, the TDK also contains hardware components for use in area and timing analysis during the TIE optimization phases. An area estimate is generated that provides a quick "ballpark estimate" of the gates needed to implement TIE.

The TDK also contains the TIE RTL module and synthesis scripts for use in detailed timing (and area) analysis.

Given that the Xtensa tools and processor installation process (refer to the *Xtensa Development Tools Installation Guide* for details) has set up the path to the Xtensa tools, you may invoke the command-line TIE Compiler utility as follows:

```
tc <options> <tie_file>
```

Where <*tie_file*> is the name of a text file that describes the TIE.

Some of the commonly used TIE compiler options are listed below. Refer to Chapter 11, “Compile Options for the TIE Compiler” on page 185 for a complete list of TIE compiler options.

- `tc -help`

  Invoke the TIE Compiler with the `-help` option to provide a full list of options and their descriptions.

- `tc -lint foo.tie`

  The `-lint` option runs a fast consistency and error check on the `foo.tie` file without generating a TDK. Use this option to save time when incrementally fixing syntax errors in a TIE file.

- `tc -d <tdk_dir> foo.tie`

  The `-d` option creates the TDK in a sub-directory named <*tdk_dir*> relative to the current operating directory. Otherwise, all TIE compiler output is generated in the current operating directory. You can create separate output directories for each version of your TIE, making it easy to switch between different TIE versions.

  For example, the following commands will compile and simulate the `bar.c` code using the TDK in directory <*tdk_dir*>.

  ```
  >xt-xcc bar.c -o bar --xtensa-params=<tdk_dir>
  ```

  ```
  >xt-run bar --xtensa-params=<tdk_dir>
  ```

  Use the `xtensa_params` command line option to locate the TDK. Alternatively, the TDK directory can also be passed to the software tools through an environment variable, XTENSA_PARAMS.

For details on all the command line options for the TIE compiler, refer to Chapter 11, “Compile Options for the TIE Compiler” on page 185.

3.2 The Xtensa Xplorer

Xtensa Xplorer is an integrated development environment (IDE) for configuring Xtensa processors, TIE development, and application software development using the C/C++ programming language and Xtensa assembly language. Xtensa Xplorer integrates the TIE compiler, allowing you to invoke the TIE compiler and get results without leaving the Xplorer environment. This allows TIE applications to be created with syntax-aware editing and automatic support for building and running applications that use TIE.

In the first phase of TIE design, Functional Description, you can specify and compile TIE in Xplorer's integrated TIE Editor Source view as shown in Figure 3–5.

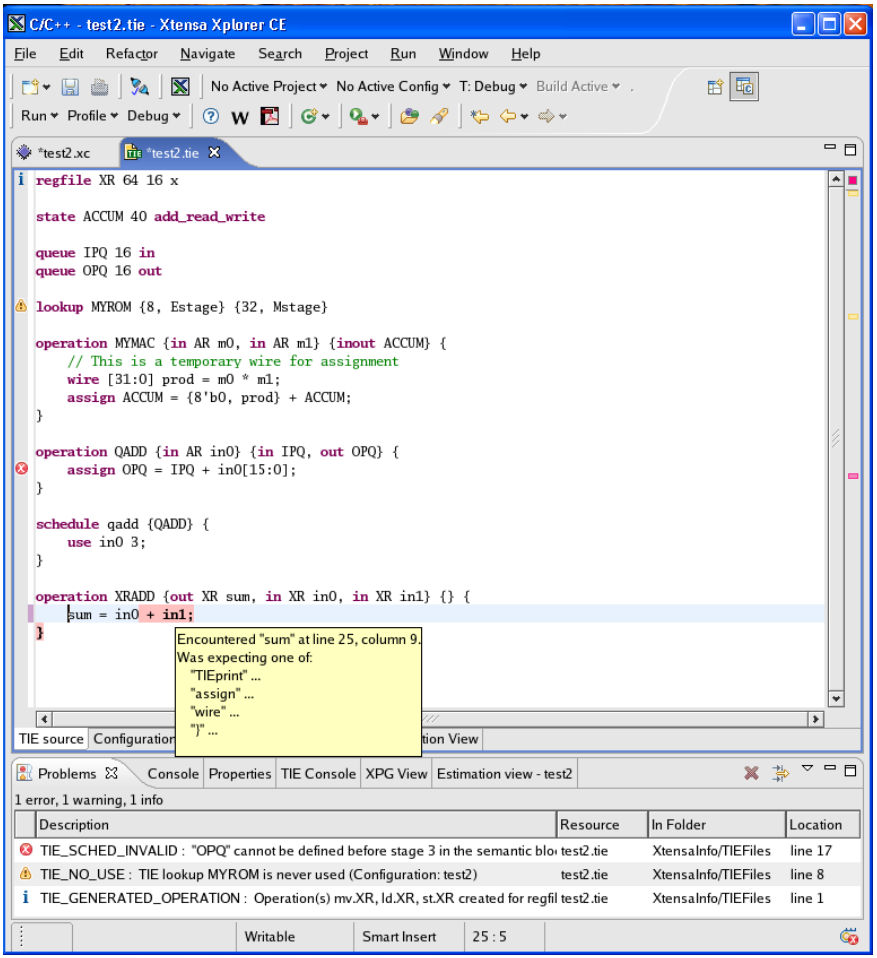

Figure 3–5. Xtensa Xplorer's TIE Editor Page - TIE Source View

The TIE editor is syntax aware, and provides automatic highlighting of TIE language keywords, appropriate indentation and on-the-fly syntax help. If errors, warnings, or messages are encountered in the compilation of your TIE description, the left margin of the TIE editor displays icons against the offending lines.

Xplorer also provides other useful views of your TIE description. One such view is the **TIE Instruction View** shown in Figure 3–6. When you select an instruction in the **Outline** view, the **Instruction View** is populated with the C syntax of the instruction, the various operands of the instruction and a gate count estimate of the hardware implementation of the instruction.

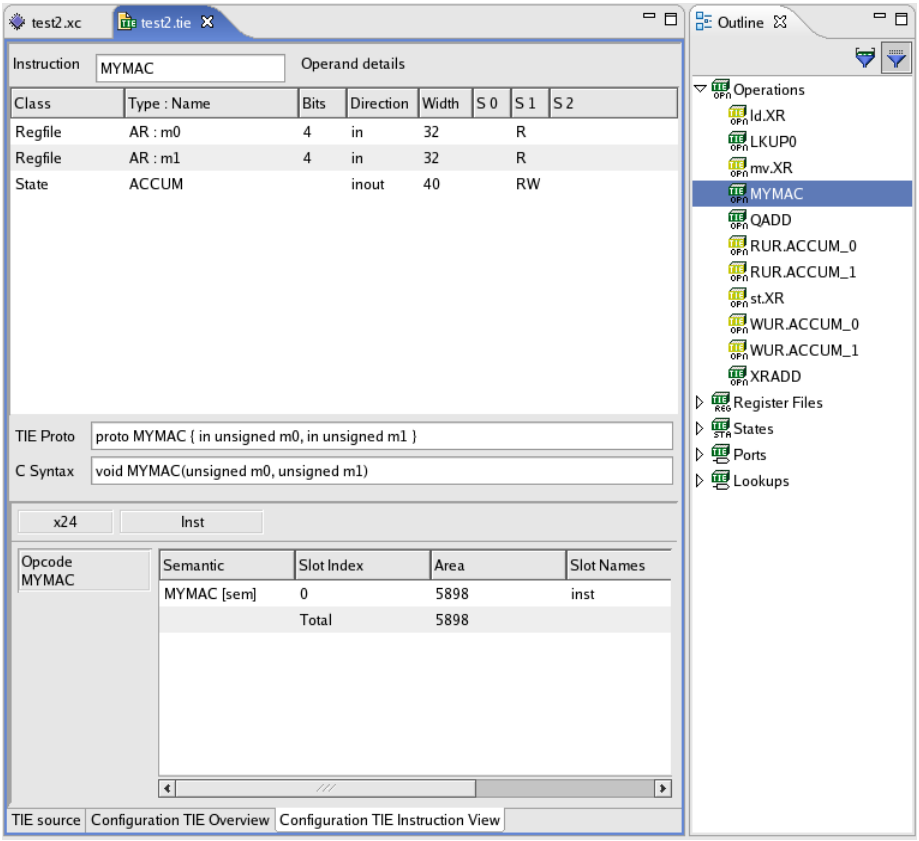

Figure 3–6. Xtensa Xplorer's TIE Instruction View

Xplorer also provides a summary of the gate-count estimate of all your TIE instructions in the **Configuration TIE Overview** page as shown in Figure 3–7. The TIE area estimator allows TIE developers to implement TIE area optimizations and to immediately see how these optimizations affect area.

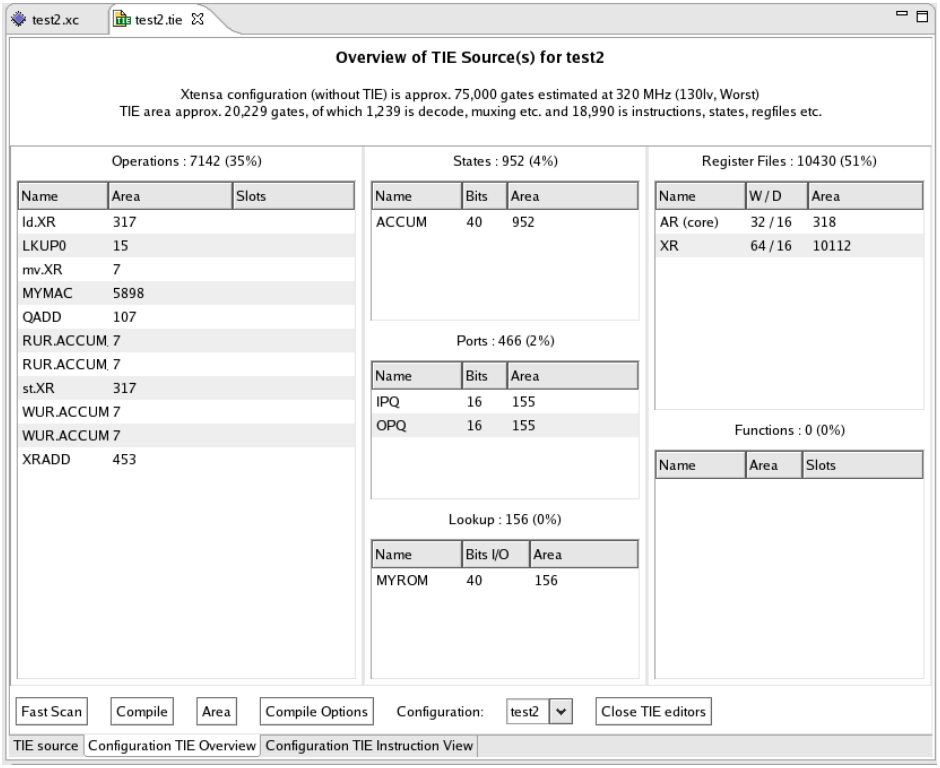

Figure 3–7. Xtensa Xplorer's Configuration TIE Overview Page

To populate the different TIE views, you must first compile your TIE description using the TIE compiler. To invoke the TIE compiler, right-click in the TIE editor and select **Compile TIE** or alternately, click on the compile TIE icon in the **Configuration Overview**. The TIE compiler generates a TDK that extends the Xtensa software development tools to support TIE instructions. After the TIE is compiled, applications can be developed and debugged within the Xplorer environment to verify correct functional behavior of TIE and generate profile data.

Because Xplorer does not drive synthesis tools, TIE timing optimizations cannot be done completely within the Xtensa Xplorer framework. However, Xplorer is useful as the “front-end” environment for specifying TIE timing optimizations and creating the TDK (containing a TIE Verilog file and script for synthesis).

In the final phase of the TIE design methodology, the processor timing and area optimization phase, Xplorer provides the tools to build the RTL for the extended processor. After an extended processor is built, the TIE becomes completely integrated into the processor. This means that you cannot modify the processor configuration parameters or TIE without first going back to the original processor (without TIE). Thus, to build an ex-

tended processor with TIE, Tensilica recommends first cloning the original processor (it may be needed later if you decide to make any changes), then add the TIE to the cloned processor.

To build an extended processor, Xplorer uploads a binary TIE database (TDB) file and configuration parameters over the Internet to the Xtensa Processor Generator (XPG). A short time later, the extended processor build is complete and can be installed by clicking the Install button in Xplorer's Configuration Overview page. After the extended processor is installed, you can access the RTL for the extended processor and implementation scripts in the Xplorer build directories. Refer to the *Xtensa Hardware User's Guide* for information about the use of synthesis and place/route scripts.

When the processor is rebuilt with TIE, the software tools will contain built-in support for the TIE. When the extended processor software tools are installed, a TDK is no longer used to support the TIE. When a processor already contains TIE, the software tools will allow further extension using a TDK only if the additional TIE does not modify the underlying hardware. Note that not all TIE will result in changes to hardware. Some TIE constructs are software-only constructs that are used by the software tools to make it easier to use the extensions from a C/C++ application.

The process of modifying TIE after it has been integrated into a processor depends on whether the TIE modifies the underlying hardware.

If the additional TIE modifies the underlying hardware, or you want to change the TIE that has already been integrated in to the processor, follow these steps:

1. Recompile your modified TIE against the original processor to create a new TDB.
2. Rebuild the processor with this TDB.
3. Install the new processor with modified TIE.

If the new TIE does not modify the underlying hardware (a feature available in the RF-2014.0 release and onwards), follow these steps:

Note: This feature is not available for processor cores built prior to the RF-2014.0 release. Performing a software upgrade for such cores to release RF-2014.0 or later will make the feature available.

1. Add the new TIE to the processor via the "Attach Software-Only TIE" UI option in Xplorer.
2. Compile the new TIE against processor embedded with the original TIE to create a new TDB. This TDB now contains constructs from the new TIE. The software constructs in new TIE override the software constructs in the original TIE.
3. Perform a software upgrade and provide the new TIE.
4. Install the new processor.

3.2.1 Adding Software TIE Constructs to a Processor

When a processor with TIE is installed, Xtensa Xplorer will no longer permit further extension using a new TDK if the new TIE associated with the TDK modifies the underlying hardware. The interface to add TIE to a processor core is disabled as shown in Figure 3–8.

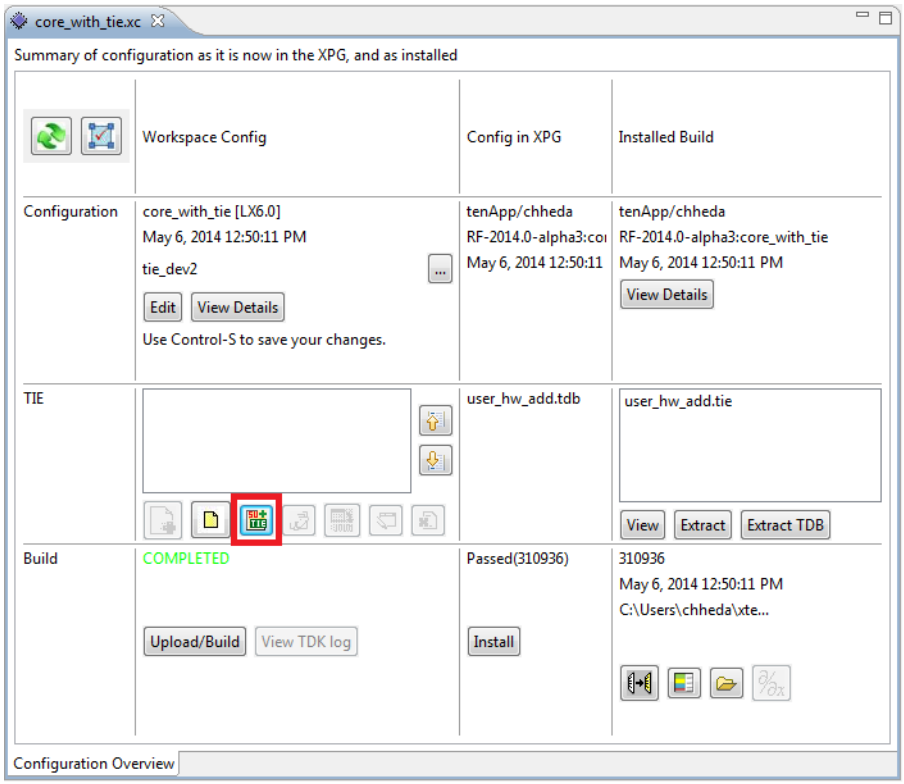

Figure 3–8. Disabled TIE Interface

At this point, the only extensions permitted are software-centric extensions that improve the programmability of the processor. TIE associated with these extensions can be added via the "Add Software-Only TIE" interface highlighted in Figure 3–8. The TIE Compiler will perform checks to ensure that the attached TIE does not modify the underlying hardware before creating a new TDK. The new TDK extends the software tools to support the new software constructs.

Submitting the binary TIE database (TDB) file and extended processor configuration parameters over the Internet to XPG will provide a new processor build that contains software tools with the new software constructs.

3.2.2 Application Profiling with Xplorer

The Xtensa Xplorer features include C/C++ code development tools and capabilities such as software project management, build management, profiling, and debugging. The Xplorer development environment streamlines the functional description phase of TIE design. In particular, you can quickly create a software project and obtain profile reports. Figure 3–9 shows the Xtensa Xplorer C/C++ Development view. This figure shows the software project `AMR_NB_Decoder` running on an Xtensa configuration `test3`. This view in Xplorer provides a listing of all your project source files and a syntax aware editor for viewing and editing C/C++ source code.

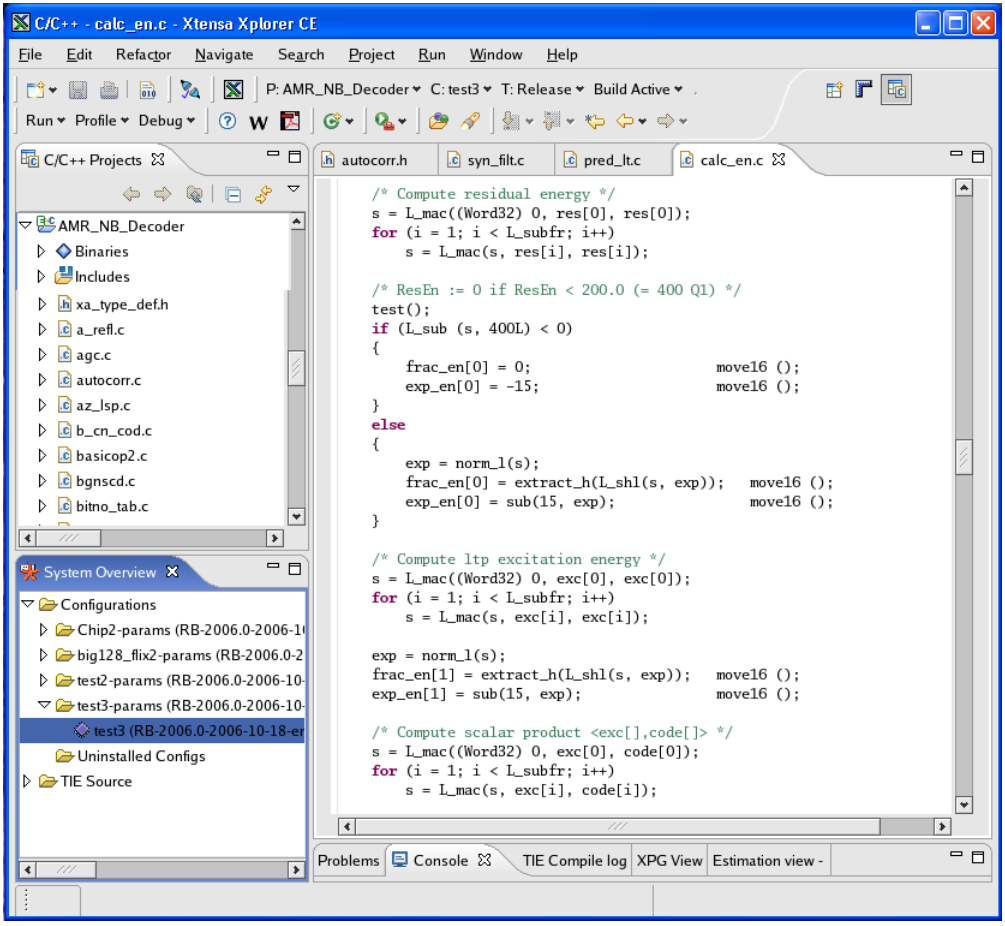

Figure 3–9. Xtensa Xplorer C/C++ Development View

When you start the process of profiling an application program to identify areas for acceleration using TIE, it is best to start with a feature rich Xtensa processor. For example, such a processor would include 128-bit wide data memory access width, dual load/store

units, FLIX support for wide instructions, support for Boolean register file, and so forth. Starting with a feature rich Xtensa processor allows you to explore a variety of TIE extensions to accelerate your application code. In the later phases of TIE design, unnecessary configuration features can be stripped out to reduce processor area.

After setting up your project, compiling the C/C++ code, setting the appropriate profile options and running a profile, the Xplorer Profile View displays as shown in Figure 3–10. Refer to Xtensa Xplorer online help for a step-by-step description of running an application profile in Xplorer.

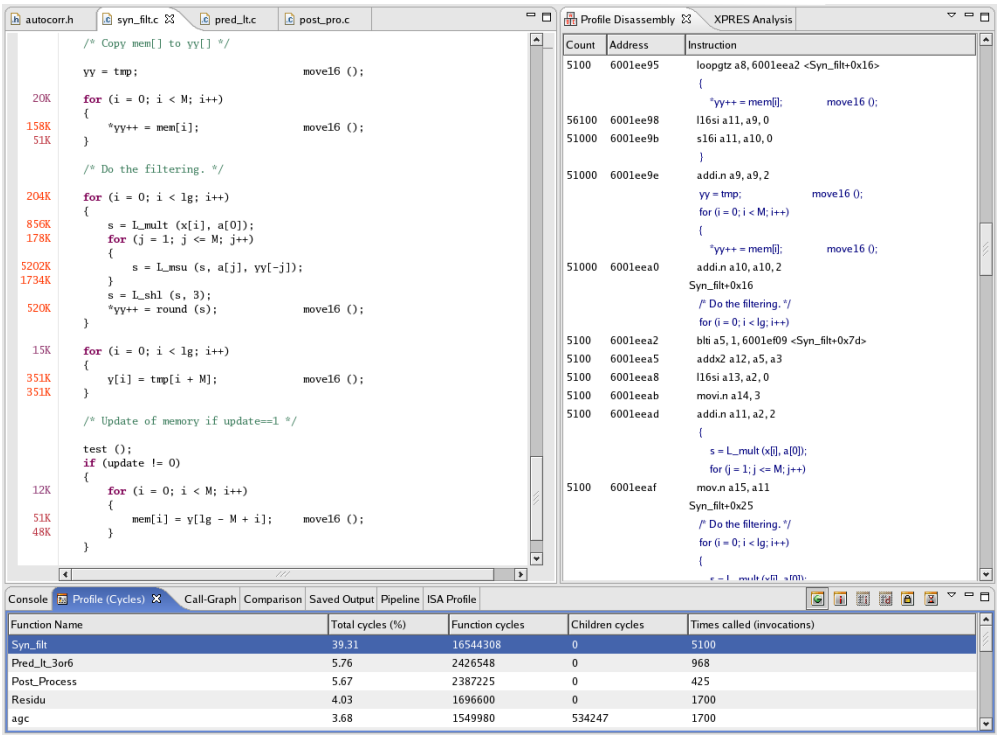

Figure 3–10. Xtensa Xplorer Profile View

The Profile view in Xplorer has three main sub-windows. The window at the bottom lists the various functions in your program that consume the greatest number of cycles. In the example of Figure 3–10, the function `Syn_filt` consumes almost 40% of the execution cycles, and is the first function to evaluate for acceleration using TIE. When you click on a function in this window the C code, as well as the compiled assembly code, for that function is shown in the top two windows.

In Figure 3–10, the `Syn_filt` function is shown in the C source view with a profile column on the left that gives the total number of cycles attributed to processing a C expression. The Profile Disassembly view on the right shows even greater profiling detail; it shows the assembly code for the `Syn_filt` function with the total number of cycles attributed to the execution of each instruction.

Chapter 5 describes how to design TIE instructions to improve the performance of your application code after you have identified the hot spots. First it is necessary to understand the use of basic TIE constructs, which is described in the next chapter.

4. TIE Language Basics

This chapter provides an introduction to the TIE language used to describe Xtensa processor extensions. The most basic TIE language constructs are described, along with examples of their use. In addition to the TIE constructs, fundamental concepts of the TIE language are introduced, followed by a more detailed discussion in the following chapter. Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a detailed description of all the TIE language constructs.

4.1 Operation Construct

The most fundamental TIE construct is `operation`. This construct specifies the name, arguments, and behavior of a TIE operation[1].

4.1.1 A Basic Operation: Byteswap

Consider the following C function byteswap, which performs a 32-bit endian conversion:

```
byteswap(unsigned swap_in)
{
     unsigned swap_out = (swap_in<<24)          |
                         ((swap_in<<8)&0xff0000) |
                         ((swap_in>>8)&0xff00)   |
                          (swap_in>>24);
     return swap_out;
}
```

This function requires many cycles to compute in software; however, it can be computed in a single cycle with a TIE operation. A useful technique for accelerating “hot spots” with TIE is to combine multiple operators into a fusion operation. A fusion operation combines a set of simple, connected operations to a single complex operation. This combination enables the Xtensa processor to perform more computations per operation, creates opportunities to share input operands, and eliminates the need to store and fetch intermediate operands.

1. For LX cores Note that there is a difference between a TIE operation and a TIE instruction. With the FLIX technology, several operations can be coded into a wide TIE instruction. Therefore, a TIE instruction may consist of one or more TIE operations.

All the operations used to perform the byteswap function are fused and optimized in the following TIE operation.

```
operation byteswap {out AR swap_out, in AR swap_in}{}
{
     assign swap_out =
          {swap_in[7:0],swap_in[15:8],swap_in[23:16],swap_in[31:24]};
}
```

These few lines are all that are necessary to extend the Xtensa processor with a custom operation that performs an endian conversion on a 32-bit word. The TIE compiler automatically extends all the software tools to enable development using the new operation.

Following is the syntax for the operation construct.[2]

```
operation <name> {<argument list>} {<state-interface list>}
{<operation description>}
```

The name of this example operation is `byteswap`. The TIE compiler makes the operation available to software by creating an assembly mnemonic and a C/C++ intrinsic function of the same name. Therefore, carefully choose operation names to best describe the operation's behavior.

After the operation name, there are three sets of braces that contain an argument list, a state-interface list, and a description of the operation's behavior. The argument list is a comma-delimited list of the register and immediate operands of an instruction. These operands are encoded in bits of the instruction word and listed in the assembly language syntax of the instruction. The argument list of the example byteswap operation is:

```
{out AR swap_out, in AR swap_in}
```

Each entry in the list of arguments has a direction, type, and name. A direction identifier prefaces each entry in the list of arguments: `in`, `out`, or `inout`, which respectively describes an operand to be an input, output, or both an input and an output of the operation. AR is a built-in operand type that references one of the sixteen visible registers from the windowed address register (AR) file as defined in the Xtensa Instruction Set Architecture (ISA). Note that user-defined register files can be defined and used as operand types (user-defined register files are covered in Section 5.6 on page 45). The names given to each operand in the argument list (`swap_out`, `swap_in`) are used in a similar manner as argument variable names in a C function. The operand names are referenced as the inputs and outputs of the operation within the description of the operation's behavior.

2. Notation for TIE syntax: <*variable*> indicates a user parameter that can be a string, list of strings, or a list of expressions. {*variable*} indicates a parameter within literal curly-braces. (*variable*) indicates a parameter within literal parentheses. [<variable>] indicates an optional parameter.

The operation construct's second set of curly braces contains a comma-delimited list of the operation's state and interface operands (states are described in Section 5.1 and interfaces are described in Section 5.7). Unlike operands declared in the argument list, these operands are implicit and not encoded in the instruction word. They are also not listed in the assembly language syntax of the instruction. When state and interface resources are not used by an operation, an empty list is required (as shown in the example byteswap operation).

The third set of braces describes the behavior of the operation. The operation's behavior is described as a list of one or more assignment statements that occur whenever the operation is executed. You can add comments throughout the TIE file in C (`/* this is a comment */`) or C++ (`//this is yet another comment`) syntax.

A TIE assignment statement can take the form of

```
assign <name> = <expression>;
```

where <*name*> is the name of an `out` or `inout` operand listed in the operation's argument or state-interface list, or a previously declared wire (wires are discussed in Section 4.1.4).

The expression in a TIE assignment statement uses the same syntax as that of combinational logic in the Verilog hardware description language. The TIE expression can be any expression using TIE operators and the names of source (`in` or `inout`) operands listed in the operation's argument or state-interface list. The TIE operators shown in Table 4–1 are essentially identical to the operators of combinational Verilog. Note that all TIE operators treat all values within a TIE expression as unsigned data. Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a detailed description of TIE operators.

Table 4–1. TIE Operators

Operator Type	Operator Symbol
Sub-range	a[n:m]
Arithmetic	+, -, *
Logical	, &&, ||
Relational	>, <, >=, <=, ==, =
Bit-wise	~, &, |, ^,^~,~^
Reduction	&, |, ^,^~,~^
Shift	>>, <<
Concatenation	{ }
Replication	{n{}}
Conditional	?:

Consider the behavioral description in the example byteswap TIE operation:

```
{
     assign swap_out =
     {swap_in[7:0],swap_in[15:8],swap_in[23:16],swap_in[31:24]};
}
```

In this example, `swap_in` and `swap_out` are declared in the argument list as operands that reference registers in the AR register file. A register is read from the AR register file and made available to the operation's behavioral description through the operand `swap_in`. Each byte of input is extracted using the sub-range operator and combined in reversed byte order with the concatenation operator. The result is assigned to the operand, `swap_out`. As `swap_out` references a register in the AR register file, the value of `swap_out` is written to the corresponding register in the AR register file.

4.1.2 Assembler and C/C++ Intrinsics

The TIE operation is immediately made available to software developers through assembly mnemonics and a C/C++ intrinsic.

The byteswap operation is mapped to an assembly mnemonic of the same name. In the assembly example

```
byteswap a4, a5
```

`a4` and `a5` refer to registers in the AR register file. The register-to-operand assignment is given by the order defined in the argument list. Register `a4` is mapped to `swap_out`, while `a5` is mapped to `swap_in`. In this example, register `a4` is written with the result of the byteswap operation on the contents of register `a5`.

An operation is also mapped to a C intrinsic of the same name (intrinsic names are case sensitive). Intrinsics simplify development of software using the custom TIE operations by automatically performing register allocation and scheduling optimizations that would otherwise be done manually if assembly was used.

C programs can use TIE intrinsics by including a header file generated by the TIE compiler. The header file (`.h`) is given the same name as the TIE file and is generated in the *<path_to_TDK>*`/include/xtensa/tie` directory. For example, if the TIE file is named `foo.tie`, the following include statement makes the TIE intrinsic available to the C program:

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

The header file contains C prototypes for the TIE intrinsics. The C prototype for the byteswap intrinsic is:

```
unsigned int byteswap (unsigned int arg1);
```

If an operation is named using a period (“.”), the name of the intrinsic is the same except that the period is replaced with an underscore (“_”). For example, an operation named “byte.swap” is given an intrinsic name of “byte_swap”.

By default, AR register file operands are cast to unsigned integer data types. Operands can be assigned to other data types using the TIE proto construct; see the *Tensilica Instruction Extension (TIE) Language Reference Manual* for details. All intrinsic arguments are mapped to operands in the order defined in the operation's argument list. However, if there is only a single out (not inout) operand specified, the out operand is always mapped to the return value of the intrinsic. In the byteswap example, there is only one out operand, `swap_out`, so it is mapped to the return value of the intrinsic. The remaining operands are mapped to arguments of the intrinsic in the same order specified in the operation’s argument list. If an operation contains an inout or more than one out operand then all operation operands are passed as arguments to the intrinsic and the intrinsic does not have a return value.

4.1.3 A More Complicated Example: Fused Dot Product Operation

Following is a slightly more complicated example that introduces a few more TIE constructs. Consider the following sample code:

```
for (i=0;i<N;i++)
        {
          acc += (Sample[i]*Coeff[i])>>8;
     }
```

This sample code calculates a dot product from two arrays. For this code example, assume that `Sample` and `Coeff` arrays hold unsigned 16-bit values. The product of the 16-bit values is scaled (right shifted by 8 bits) prior to accumulating in a 32-bit unsigned accumulator.

TIE designers can accelerate algorithms by creating fusion operations. Following is an example of a TIE operation that accelerates the dot product calculation by fusing multiplication, accumulation, and scaling into a single operation.

```
operation scaled_mac {inout AR oper0, in AR oper1, in AR oper2} {}
{
     assign oper0 = ((oper1[15:0] * oper2[15:0]) >> 8) + oper0;
}
```

The sample code is modified to support the fused operation.

```
for (i=0;i<N;i++)
     {
```

```
            //TIE accelerated version
            scaled_mac(acc,Sample[i],Coeff[i]);
        }
```

4.1.4 TIE Wires

Because TIE is used to describe more functionality with many fused operators, assigning output operands with a single TIE expression becomes impractical. Complex TIE expressions result in TIE operations that are often difficult to read and hard to debug. TIE wires allow complex TIE expressions to be described in simpler TIE expressions.

The original `scaled_mac` operation had subrange, multiplication, addition, and shift operators within the same TIE expression. This TIE expression can be described in simpler TIE expressions as follows:

```
operation scaled_mac {inout AR oper0, in AR oper1, in AR oper2}{}
{
     //declare wires
     wire [15:0] temp1, temp2;
     wire [31:0] product, scaled;

     //assignment of intermediate computations
     assign temp1 = oper1[15:0];
     assign temp2 = oper2[15:0];
     assign product = temp1 * temp2;
     assign scaled = product >> 8;

     assign oper0   = oper0 + scaled;
}
```

Wires are temporary “scratchpad” variables (similar to local variables in a C function), and are assigned intermediate computations of an operation. TIE wires are declared using the following syntax:

wire [<*width*>] <*wire names*>

By default, a wire is 1-bit wide, however, it can be defined as multiple bits when a width is specified (for example, [31:0] describes a width of 32-bits). A single wire, or a comma-delimited list of TIE wires, can be declared within the same TIE wire statement.

In the new version of the TIE `scaled_mac` operation, wires `temp1` and `temp2` are declared and assigned a 16-bit subrange of `oper1` and `oper2`, respectively. A 32-bit TIE wire, `product`, is declared and assigned the product of `temp1` and `temp2`. While debugging, the value of this wire can be examined and compared against expected intermediate results. TIE debugging is enhanced by the ability to examine the value of TIE wires to verify correct behavior of intermediate computations.[3]

The previous example shows TIE wires declared and then assigned in separate TIE statements. A single TIE wire can be declared and assigned in the same TIE statement as shown in the following example.

```
operation scaled_mac {inout AR oper0, in AR oper1, in AR oper2}{}
{
     //declaration and assignment of intermediate computations
     wire [15:0] temp1   = oper1[15:0];
     wire [15:0] temp2   = oper2[15:0];
     wire [31:0] product = temp1 * temp2;
     wire [31:0] scaled  = product >> 8;
     assign oper0        = oper0 + scaled;
}
```

When declaring and assigning a TIE wire in the same TIE expression, the following syntax is used:

```
wire [<width>] <name> = <expression>;
```

4.1.5 TIE Assignment Statement Ordering

Similar to Verilog, but unlike a C program, the ordering of assignment statements in an operation's behavioral description bears no relationship to the order in which the TIE assignments are carried out. All input arguments for all statements are read before any output arguments for any statement are written, regardless of the order of the statements. No variable can be written more than once. There is no functional difference in the following example and the example shown previously:

3. Debugging with TIE wires only applies to using the instruction set simulator. Debugging with TIE wires is not supported on hardware targets.

```
//Example 2
operation scaled_mac {inout AR oper0, in AR oper1, in AR oper2}{}
{
     //declare wires
     wire [15:0] temp1, temp2;
     wire [31:0] product, scaled;

     assign oper0   = oper0 + scaled;

     //assignment of intermediate computations
     assign scaled = product >> 8;
     assign product = temp1 * temp2;
     assign temp2   = oper2[15:0];
     assign temp1   = oper1[15:0];
}
```

Note the following concepts when developing TIE descriptions:

- An `inout` type operand is conceptually two operands. The `in` operand is read into the operation prior to performing any computations and the `out` operand is written after all computations have completed.
- All TIE assignment statements are evaluated for dependencies and computed in order of dependency. In the preceding example, wires `temp1` and `temp2` are computed first because they are only dependent on `in` operands. The wire `product` is computed after the wires `temp1` and `temp2` are assigned, and then wire *scaled* is computed. Finally, `oper0` is computed and written. Note that the TIE compiler checks for cyclical dependencies and flags these as errors.
- For better readability of the TIE behavioral descriptions, list TIE assignment statements in order of dependency.

4.2 TIE Modules

The TIE language contains built-in modules that can be used in operation expressions. TIE modules are used to specify computations that are cumbersome to describe through the use of standard TIE operators. The TIE modules provide a set of combinational functions that instantiate hardware that has been pre-optimized for improved timing and reduced area. The use of TIE modules generally results in TIE with more optimal timing and area compared to TIE described using standard TIE operators.

TIE modules, as defined in Table 4–2, are described in more detail in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. Table 4–2 is a list of available built-in modules.

Table 4–2. TIE Modules

Format	Description	Result Definition
`TIEadd(a, b, cin)`	Add with carry-in.	a + b + cin
`TIEaddn(`a_0`, `a_1`, ... `a_{n-1}`)`	N-number addition	$a_0 + a_1 + ... + a_{n-1}$
`TIEcmp(a, b, sign)`	Signed and unsigned comparison	{a < b, a <= b, a == b, a >= b, a > b}
`TIEcsa(a, b, c)`	Carry-save adder.	{a & b | a & c | b & c, a ^ b ^ c}
`TIElzc(a)`	Leading zero count	a[w-1] ? 0 : a[w-2] ? 1 : ... a[0] ? (w-1) : w where w is the computed width of a
`TIEmac(a, b, c, sign, negate)`	Multiply-accumulate.	negate ? c - a * b : c + a * b where `sign` specifies how `a` and `b` are extended in the same way as for `TIEmul`
`TIEmul(a, b, sign)`	Signed and unsigned multiplication.	{{m{a[n-1] & sign}}, a} * {{n{b[m-1] & sign}}, b} where n is size of a and m is size of b
`TIEmulpp(a, b, sign, negate)`	Partial-product multiply	negate ? - a*b : a*b
`TIEmux(s, `d_0`, `d_1`, ..., `d_{n-1}`)`	n-way multiplexor	s==0 ? d_0 : s==1? d_1 : ... : s ==n-2 ? d_{n-2} : d_{n-1}
`TIEpsel(`s_0`, `d_0`, `s_1`, `d_1`, ..., `s_{n-1}`, `d_{n-1}`)`	n-way priority selector	s_0 ? d_0 : s_1 ? d_1 : ... : s_{n-1} ? d_{n-1} : 0

`TIEsel(`s_0`, `d_0`, `s_1`, `d_1`, ..., `s_{n-1}`, `d_{n-1}`)`	n-way 1-hot selector	`(size{`S_0`} & `D_0`) |`
		`(size{`S_1`} & `D_1`) |...`
		`(size{`S_{n-1}`} & `D_{n-1}`)`
		where size is the maximum width of D_0 ... D_{n-1}

Previous sections described how to design a TIE operation (`scaled_mac`) to perform multiplication, scaling, and accumulation. However, this operation is only capable of handling unsigned operands (because the TIE multiply operator * is an unsigned operator). The operation must be rewritten for signed operands. The simplest way to support signed multiplication is to use the `TIEmul` module. The third argument to the `TIEmul` module is a single bit that instantiates a signed multiplier when set to 1. The following example operation, `sscaled_mac`, is a signed version of the previous `scaled_mac` example.

```
operation sscaled_mac {inout AR oper0, in AR oper1, in AR oper2}{}
{
     wire do_signed      = 1'b1; //signed arithmetic
     wire [31:0] product = TIEmul(oper1[15:0], oper2[15:0],
                                                  do_signed);
     wire [31:0] scaled = {{8{product[31]}}, product} >> 8;
     assign oper0   = oper0 + scaled;
}
```

4.3 TIE State

In the previous dot product examples, scaling was used so that the summation of the 32-bit product and 32-bit accumulator can be stored in a 32-bit AR register operand without an overflow. Consider a situation where greater accumulator precision is required as a result of the multiplier and multiplicand being 24-bits wide instead of being 16-bits wide. Because the AR register file is limited to 32 bits, AR operands cannot be used for the accumulator. Instead, wide TIE state can be defined for the accumulator.

TIE states are special purpose registers for exclusive use by TIE operations. TIE states are directly read and written by TIE operations. TIE state is also useful for maintaining commonly used data inside the processor. Performance is improved when the processor has direct access to commonly used data instead of accessing the data through memory accesses.

The syntax for the state construct is:

```
state <name> <width> [<reset_value>] [<add_read_write>] [<shared_or>] [<export>]
```

Note that states, as well as other TIE resources, must be defined before use. Therefore, states and other resources are defined at the beginning of a TIE file, and operations that use them are defined towards the end of a TIE file. Consider the following definition of a TIE state:

```
state foo 32 32'b0 add_read_write
```

This defines a state named `foo` that is 32-bits wide. The `add_read_write` keyword directs the TIE compiler to automatically create WUR (Write User Register) and RUR (Read User Register) operations to read from and write to the state respectively. For example, a new operation, `rur.foo`, is automatically defined to read data from the `foo` state to a register in the AR register file. Another operation, `wur.foo`, is automatically defined to write data to the `foo` state from a register in the AR register file. By adding the `add_read_write` keyword, the `foo` state is made directly accessible to software. Xtensa operating systems automatically save and restore accessible states upon

context switch. Also, accessible states are visible when debugging hardware targets. For these reasons, the use of the `add_read_write` keyword is recommended for all TIE states.

By default, a state is not initialized at reset. However, a reset value can be defined using the optional `reset_value` parameter. In the state example, state `foo` is initialized to 0 at reset. Multiple operations can write to the same state in the same cycle if shared_or parameter is specified. The end result is applying the OR operator on the assigned values (as an example, this could be used to detect any overflow from multiple MAC operations executing in parallel). Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for details. Finally, state values can be exported to logic outside of the processor using the optional `export` keyword (this keyword is described in more detail in Section 5.10.1).

As with all TIE operations, intrinsic functions are also automatically generated for RUR and WUR operations. The intrinsic function generally has the same name as the TIE operation. However, a period in the operation name is replaced with an underscore in the intrinsic function name (this is necessary because the period is a structure member operator in ANSI C). Thus, the intrinsic functions are named using the original state name with RUR_ (for read) or WUR_ (for write) preceding the name. The syntax for intrinsic functions used to access state defined with the `add_read_write` keyword is:

```
void WUR_<state_name>(unsigned int data_to_write_into_state)
```

and

```
unsigned int RUR_<state_name>(void);
```

When TIE states are wider than 32 bits, multiple read/write operations are created to access the state in 32-bit words. A word number (0 for least significant word) is appended to the name of the RUR or WUR operation, separated with an underscore. Consider the following TIE state definition:

```
state bar 40 add_read_write
```

The following intrinsic functions are automatically generated to access a 40-bit state, `bar`:

```
WUR_bar_0(unsigned int value_to_write): Writes [31:0] of state bar
WUR_bar_1(unsigned int value_to_write): Writes [39:32] of state bar
unsigned int RUR_bar_0(void):           Reads [31:0] of state bar
unsigned int RUR_bar_1(void):           Reads [39:32] of state bar
```

After states are declared in the state-interface list, they can be used as operands in the TIE assignment statements. Like the other operation arguments, a direction designator (`in`, `out`, or `inout`) precedes the name of the state. This is shown in the following TIE example.

```
state acc 56 add_read_write

operation state_mac {in AR oper1, in AR oper2}{inout acc}
{
     wire do_signed = 1'b1; //signed arithmetic
     wire negate    = 1'b0; // do not negate
     assign acc     = TIEmac (oper1[23:0], oper2[23:0], acc,
                                          negate, do_signed);
}
```

This TIE example accelerates the dot product function (with 24-bit multiplier and multiplicand) using a wide TIE state based accumulator with fused multiplication and accumulation. The 56-bit accumulator provides extra guard bits to accumulate the 48-bit product without an overflow. Following is the C source code for the dot product function using the TIE:

```
WUR_acc_0(0); WUR_acc_1(0); //Initialize wide state accumulator

for (i=0;i<N;i++)
     {
     //TIE accelerated version
     state_mac(Sample24[i],Coeff24[i]);
     }

accumulator_high = RUR_acc_1();
accumulator_low  = RUR_acc_0();
```

Note that the state `acc` is not listed as an argument to the `state_mac` intrinsic. This is because state operands are passed implicitly to the operation. Also note that `Sample24` and `Coeff24` are assumed to be arrays of 24-bit values.

5. TIE Examples

This chapter explores several approaches to optimize a matrix transform computation, and in the process introduces several new TIE concepts and features. Note that the examples provided in this chapter are primarily chosen to illustrate various TIE acceleration possibilities, rather than to demonstrate the process of arriving at optimal TIE solutions. Tradeoffs between various TIE acceleration approaches are explored in later chapters.

Before developing TIE, it is important to understand the algorithm you wish to accelerate. Consider a basic 4x4 matrix transform used in video systems for color space conversion. The 4x4 matrix transformation is represented as:

```
[ A0 ]     [ M0  M1  M2  M3 ]   [ B0 ]
[ A1 ]     [ M4  M5  M6  M7 ]   [ B1 ]
[ A2 ]  =  [ M8  M9  M10 M11] * [ B2 ]
[ A3 ]     [ M12 M13 M14 M15]   [ B3 ]
```

where:

- `[B0..B3]` is the input vector, each element is a 16-bit signed value
- `[M0..M15]` is the 4x4 transformation matrix, each element is a 16-bit signed value
- `[A0..A3]` is the output vector, each element is a 32-bit signed value

The matrix transformation is expanded as follows:

```
A0 = (M0 * B0) + (M1 * B1) + (M2 * B2) + (M3 * B3)
A1 = (M4 * B0) + (M5 * B1) + (M6 * B2) + (M7 * B3)
A2 = (M8 * B0) + (M9 * B1) + (M10 * B2) + (M11 * B3)
A3 = (M12 * B0) + (M13 * B1) + (M14 * B2) + (M15 * B3)
```

Following is the C function for Matrix Transformation:

```
void MatrixTransform
(short InputVector[], int OutputVector[], short TransformMatrix[])
{
     int i;
     //partial accumulators
     int a0,a1,a2,a3;

     a0=0; a1=0; a2=0; a3=0;
     for (i=0;i<4;i++)
        {
          a0 += InputVector[i]*TransformMatrix[i];
          a1 += InputVector[i]*TransformMatrix[i+4];
```

```
        a2 += InputVector[i]*TransformMatrix[i+8];
        a3 += InputVector[i]*TransformMatrix[i+12];
        }
    OutputVector[0]=a0;
    OutputVector[1]=a1;
    OutputVector[2]=a2;
    OutputVector[3]=a3;
    }
```

Most of the processor cycles are consumed within the `for` loop. In each iteration of the loop, there are several memory loads, multiplications, and additions. This function requires 550 cycles on a base Xtensa processor (without an integer multiply instruction). This section presents TIE examples that accelerate this loop, starting with basic TIE acceleration, progressing to more advanced examples.

5.1 Basic TIE Acceleration

Before new TIE features are introduced, consider accelerating the `MatrixTransform` function using the basic optimization concepts and TIE features covered in the previous section. Note that the basic example is not optimal in terms of area or performance and is only provided as a starting point to help illustrate the benefits of the TIE features and concepts presented in this chapter. In later sections, new concepts and features will be introduced to reduce processor area and provide even greater performance across the `MatrixTransform` function.

One approach to accelerating the `MatrixTransform` function with TIE is to use fusion operations with TIE modules and states. The `MatrixTransform` function uses four accumulators, a0 through a3. Four TIE states named `acc0` through `acc3` are used to hold the 32-bit accumulator values for a0 through a3 respectively. Fusion operations, `macc_0-macc_3`, use the `TIEmac` module with accumulators `acc0`-`acc3` respectively.

```
state acc0 32 add_read_write
state acc1 32 add_read_write
state acc2 32 add_read_write
state acc3 32 add_read_write

operation macc_0 {in AR oper0, in AR oper1}{inout acc0}
{
     wire do_signed = 1'b1; //signed arithmetic
     wire negate    = 1'b0; //do not negate product
     assign acc0    = TIEmac(oper0[15:0], oper1[15:0], acc0,
                             do_signed, negate);
}
operation macc_1 {in AR oper0, in AR oper1}{inout acc1}
{
     wire do_signed = 1'b1; //signed arithmetic
```

```
     wire negate    = 1'b0; //do not negate product
     assign acc1    = TIEmac(oper0[15:0], oper1[15:0], acc1,
                             do_signed, negate);
}
operation macc_2 {in AR oper0, in AR oper1}{inout acc2}
{
     wire do_signed = 1'b1; //signed arithmetic
     wire negate    = 1'b0; //do not negate product
     assign acc2    = TIEmac(oper0[15:0], oper1[15:0], acc2,
                             do_signed, negate);
}
operation macc_3 {in AR oper0, in AR oper1}{inout acc3}
{
     wire do_signed = 1'b1; //signed arithmetic
     wire negate    = 1'b0; //do not negate product
     assign acc3    = TIEmac(oper0[15:0], oper1[15:0], acc3,
                             do_signed, negate);
}
```

The TIE description is saved in a file named `matrixtransform.tie` and compiled with the TIE compiler. Following is the C source code for the `MatrixTransform` function using the TIE:

```
#include <xtensa/tie/matrixtransform.h>
void MatrixTransform (short InputVector[], int OutputVector[], short
TransformMatrix[])
{
     int i;

     //Initialize accumulators
     WUR_acc0(0); WUR_acc1(0); WUR_acc2(0); WUR_acc3(0);

     //Perform Transformation
     for (i=0;i<4;i++)
        {
          macc_0(InputVector[i],TransformMatrix[i]);
          macc_1(InputVector[i],TransformMatrix[i+4]);
          macc_2(InputVector[i],TransformMatrix[i+8]);
          macc_3(InputVector[i],TransformMatrix[i+12]);
          }

     //Store accumulator results
     OutputVector[0]=(int)RUR_acc0();
     OutputVector[1]=(int)RUR_acc1();
     OutputVector[2]=(int)RUR_acc2();
     OutputVector[3]=(int)RUR_acc3();
}
```

The basic TIE example significantly improves processor performance for Matrix Transformation. The base Xtensa processor required 550 cycles for Matrix Transformation, whereas the TIE accelerated Xtensa processor requires only 60 cycles (a speedup of nine times).

In some cases, TIE state is a major contributor to processor area. It is important to use the TIE area estimator to examine how state affects area. The TIE area estimate for the earlier state and fusion instructions is shown in Figure 5–11.

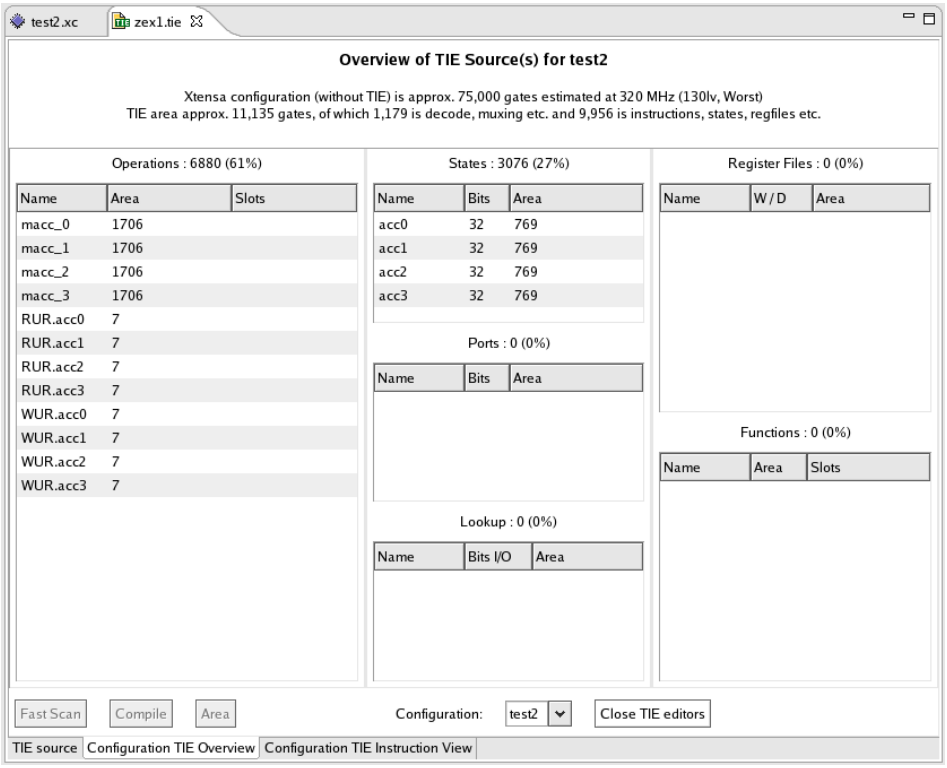

Figure 5–11. TIE Area Estimate for State and Fusion Operations

The new TIE states require additional area; however, multipliers and adders used in each of the `macc` operations use the majority of area. Because the multiplier/adder hardware of each `macc` operation performs the exact same function, you can share this hardware across all the `macc` operations. Section 5.2 describes the use of TIE "Functions" to share hardware across operations.

5.2 TIE Functions

In the previous example in Section 5.1, each of the four operation statements uses a separate hardware multiplier, even though the four operations can never be executed in parallel. The use of shared functions allows you to share one multiplier among four operations.

A TIE function creates a block of continuous assignment statements with explicitly declared input arguments and return size. For example, a function such as a multiplier and adder may be used by several different processor operations. The easiest way to implement this structure is by using a shared function. The processor generator identifies all instructions using this function and builds appropriate routing and multiplexing so that one function hardware block can operate on different operands at different times.

Following is the syntax of a TIE function:

```
function [<width>] <name> (input argument list) [<sharing attribute>]
{function description}
```

TIE functions take input from any number of arguments and generate a single output. The function name is used as the output wire of the function. The width parameter that precedes the function name defines the bit width of the output wire and is optional if the width of the output is 1. The input argument list contains a comma-delimited list of named arguments. A width parameter that defines the bit width of each input precedes these arguments. You can define a sharing attribute to allow hardware sharing of the function's hardware across multiple operations. Finally, a description of the function's behavior is provided. The description of a function is written using the same syntax as an operation's behavioral description.

Consider the following `mac16` function:

```
function [31:0] mac16 ([31:0] accumulator, [15:0] multiplier,
                       [15:0] multiplicand)
{
     wire do_signed = 1'b1; //signed arithmetic
     wire negate = 1'b0; //do not negate product
     assign mac16= TIEmac(multiplier, multiplicand, accumulator,
                              do_signed, negate);
}
```

The `mac16` function takes two 16-bit input arguments (multiplier and multiplicand), performs a multiplication on these operands, adds it with the 32-bit accumulator, and then returns the sum via the `mac16` output wire.

By default, a function is instantiated once for each use. Each instantiation creates a new hardware block to implement the function. Multiple instances of function hardware makes sense when the individual blocks are computing in parallel. In cases where the

functions are not computing in parallel, you can reduce the hardware area by sharing the function across operations. When a function is given an attribute of “shared”, a single instance of hardware will be shared for every use.

Be aware that shared functions have some limitations. For example, they cannot be nested (a shared function cannot use another function). In addition, shared functions must execute within a single cycle (execution of these functions cannot span across multiple cycles; however, a multi-cycle operation can use a shared function). Using the TIE schedule construct to define multi-cycle operations is discussed in Section 5.8).

The following TIE example shows a `mac16` function shared across the four `macc` operations.

```
state acc0 32 add_read_write
state acc1 32 add_read_write
state acc2 32 add_read_write
state acc3 32 add_read_write

function [31:0] mac16 ([31:0] accumulator, [15:0] multiplier,
                       [15:0] multiplicand) shared
{
wire do_signed = 1'b1; //signed arithmetic
wire negate    = 1'b0; //do not negate product
assign mac16   = TIEmac(multiplier, multiplicand, accumulator,
                        do_signed, negate);

}

operation macc_0 {in AR oper0, in AR oper1}{inout acc0}
{
assign acc0 = mac16(acc0, oper0[15:0], oper1[15:0]);
}

operation macc_1 {in AR oper0, in AR oper1}{inout acc1}
{
assign acc1 = mac16(acc1, oper0[15:0], oper1[15:0]);
}

operation macc_2 {in AR oper0, in AR oper1}{inout acc2}
{
assign acc2 = mac16(acc2, oper0[15:0], oper1[15:0]);
}

operation macc_3 {in AR oper0, in AR oper1}{inout acc3}
{
assign acc3 = mac16(acc3, oper0[15:0], oper1[15:0]);
}
```

The TIE area estimate for the `macc` instructions is shown in Figure 5–12. Compare this estimate with the example provided in the previous section to see that the gate count has been significantly reduced (from 1706 to 7) due to function sharing.

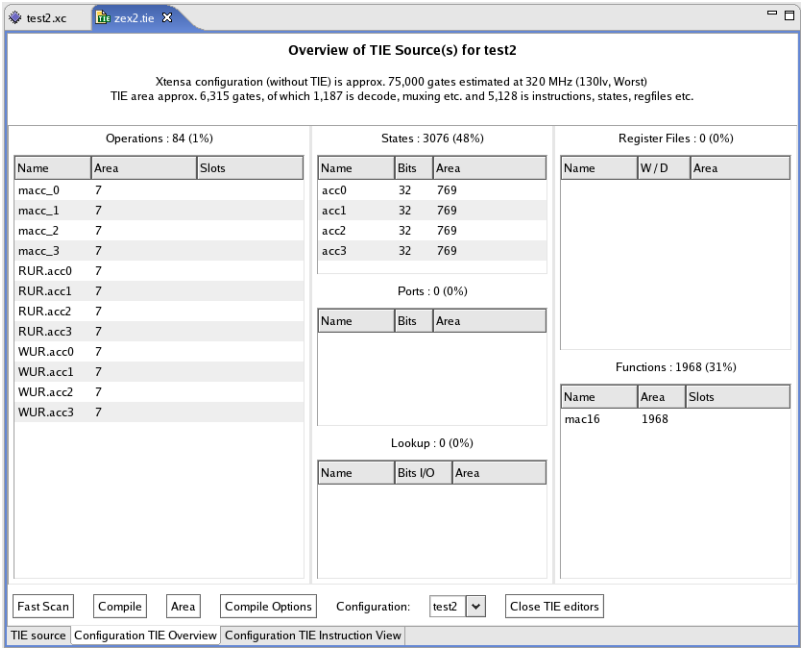

Figure 5–12. TIE Area Estimate: Hardware Sharing with TIE Function

Note that sharing a multiplier through a shared TIE function results in significantly better area; however, it may limit the maximum operating frequency of the processor. This is because the shared function, `mac16`, must execute in a single cycle. For high frequency designs, it may be necessary to schedule operations across multiple cycles (multi-cycle TIE is covered in greater detail in Section 5.8). Potentially, the `mac16` function can be split into two shared functions (each executing within a single cycle), one using the TI-Emulpp module and the other using a TIEadd module. Another option is to use the TIE semantic construct. This provides another method of hardware sharing that does not limit execution of shared hardware to a single cycle.

5.3 TIE Semantics

In the Functional Description phase of the TIE design methodology, processor extensions are described using the TIE operation construct. The TIE operation construct is used to describe the behavior of a single *operation*, with emphasis on readability and simplicity. In the optimization phase of the TIE design methodology, it may be necessary to generate more efficient hardware implementations of individual TIE operations or

share common hardware blocks across different TIE operations. The TIE semantic construct provides a convenient way to describe hardware-optimized versions of one or more TIE operations. By having both a simple operation description and an optimized semantic description of an operation, formal verification can be used to assure that there are no functional bugs introduced in the optimized TIE semantic description (formal verification of optimized TIE operations is described in Section 9.6.1). Note that a TIE semantic can only be used on TIE operations that have been described previously in the TIE file. When a TIE operation is also described by a semantic, the semantic description will be modeled by the instruction set simulator and implemented in hardware while the operation description is used as the golden reference during equivalency checking.

The syntax for the semantic construct is:

```
semantic <name> {operation-list}{semantic description}
```

A comma-delimited list describes the set of operations that are being optimized by the semantic. The operands for the semantic are automatically determined from the TIE operation definitions. Also, each operation name is used as a 1-bit operand that is logic 1 when the operation is being executed; otherwise it is logic 0.

The `macc` operations presented in Section 5.1 can be optimized for area by sharing hardware. This is done by adding the following semantic description to the previous TIE example:

```
semantic macc_ops {macc_0, macc_1, macc_2, macc_3}
{
//select the accumulator
wire [31:0] accumulator = TIEsel(macc_0, acc0, macc_1, acc1,
                                 macc_2, acc2, macc_3, acc3);

//perform mac
wire do_signed = 1'b1; //signed arithmetic
wire do_negate = 1'b0; //do not negate product
wire [31:0] mac16 = TIEmac(oper0[15:0], oper1[15:0], accumulator,
                           do_signed, do_negate);

//assign results back to accumulator
assign acc0 = mac16;
assign acc1 = mac16;
assign acc2 = mac16;
assign acc3 = mac16;
}
```

In this semantic definition, the `macc_0` - `macc_3` operations are executed with a single hardware module named `macc_ops`. Depending on which of the four operations are being executed, the TIEsel module selects one of four accumulators (`acc0`-`acc3`) as the accumulator passed to the TIEmac module. Finally, all accumulator states are

assigned with the `mac16` result. However, only the state operand defined by the operation that is being executed will be written by the semantic (that is, the `macc_0` operation is defined with `acc0` as the only output operand. If `macc_0` is being executed by this semantic, only the `acc0` state will be written). An advantage of sharing hardware using the semantic construct is that shared hardware can be scheduled to execute in multiple cycles (as opposed to TIE functions which must execute within a single clock cycle).

The TIE area estimate for the `macc` operations defined by a semantic is shown in Figure 5–13. Compared with the example provided in the previous section, the total gate count is similar. However, note that the gate count attributed to operations is described as the total semantic gate count divided by the number of operations of the semantic.

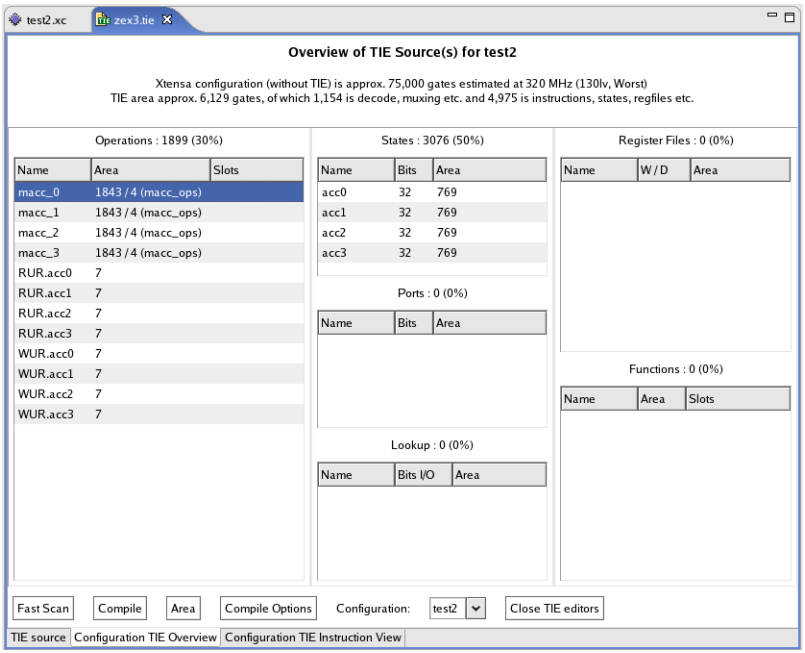

Figure 5–13. TIE Area Estimate: Hardware Sharing with TIE Semantic

5.4 TIE Tables

The `MatrixTransform` function uses an array, `TransformMatrix`, to hold the 4x4 transform matrix. Consider the following array:

```
short
TransformMatrix[16]={0,323,543,453,432,956,2047,72,843,98,1000,113,
1232,132,1450,-2048};
```

The performance of the MatrixTransform function is limited by the memory access cycles required to access this array.

A TIE table is a table of constants that is built into the processor hardware. By defining the array as a TIE table, an operation has direct access to the transform matrix without the overhead associated with memory access cycles. The following example shows the same array mapped to a TIE table:

```
table matrix 32 16
{0,323,543,453,432,956,2047,72,843,98,1000,113,1232,132,1450,-2048}
```

Following is the syntax for a table construct:

```
table <name> <width> <number of elements> {element-list}
```

where *element-list* is a comma-delimited list of table entries where each entry is either an integer or Verilog number representation (for example, 8'h00). If negative integers are listed in the value-list of a table, the width of the table must be set to 32 bits. Note that the above table example is defined to be 32 bits, although the TransformMatrix array contains only 16-bit signed values.

TIE operations can access tables through direct reference. This can be useful for TIE operations that always access the same table values. The following TIE assignment statements show direct access to matrix table values.

```
wire [31:0] matrix_5, matrix_10;
wire [3:0] offset = 10
assign matrix_5 = matrix[5];  /* 956 */
assign matrix_10 = matrix[offset] /* 1000 */
```

TIE operands can also access tables through explicit operand reference by using the table name as an operand type. The following TIE operation example, macc_0, uses the matrix table operand, oper1, to reference a value in the matrix table. The fusion operation performs a signed mac of the table and operand value, accumulating the result in state acc0. The same modification is made for operations macc_1 to macc_3.

```
table matrix 32 16
{0,323,543,453,432,956,2047,72,843,98,1000,113,1232,132,1450,-2048}


state acc0 32 add_read_write
state acc1 32 add_read_write
state acc2 32 add_read_write
state acc3 32 add_read_write

function [31:0] mac16 ([31:0] accumulator, [15:0] multiplier,
                       [15:0] multiplicand) shared
{
```

```
        wire do_signed = 1'b1;//signed arithmetic
        wire negate = 1'b0; //do not negate product
        assign mac16 = TIEmac(multiplier, multiplicand, accumulator,
                              do_signed, negate);
}

operation macc_0 {in AR oper0, in matrix oper1}{inout acc0}
{
        assign acc0 = mac16(acc0, oper0[15:0], oper1[15:0]);
}

operation macc_1 {in AR oper0, in matrix oper1}{inout acc1}
{
        assign acc1 = mac16(acc1, oper0[15:0], oper1[15:0]);
}

        operation macc_2 {in AR oper0, in matrix oper1}{inout acc2}
{
        assign acc2 = mac16(acc2, oper0[15:0], oper1[15:0]);
}

operation macc_3 {in AR oper0, in matrix oper1}{inout acc3}
{
        assign acc3 = mac16(acc3, oper0[15:0], oper1[15:0]);
}
```

When a TIE table is defined in the operand list, an immediate operand field is automatically created in the instruction word. As a convenience for the programmer, the assembly and C syntax passes table immediate operands by values in the table, rather than by the table index.

Following is an example of the C/assembly usage:

```
macc_0 a15, -2048             //assembly usage example
macc_0 (InputVector[2],-2048);//C usage example
```

In the preceding examples, the binary code representation of the table value (-2048) is not embedded in the instruction word. Instead, the assembler encodes the value to the 4-bit table offset and embeds this into the instruction word. The processor automatically performs a table look-up using the value embedded in the instruction word. Note that if an argument is provided that does not exist in the table, an error will be flagged at compile time.

To take advantage of the `macc` instructions with table access, the main loop of the `MatrixTransform` function is unrolled and table values are specified as the second argument to the intrinsic functions. Using the following modified `MatrixTransform`, the function performance improves to 37 cycles per function invocation.

```
void MatrixTransform (short InputVector[], int OutputVector[])
{
     int i;

     //Initialize accumulators
     WUR_acc0(0); WUR_acc1(0); WUR_acc2(0); WUR_acc3(0);

     //loop unrolled
     macc_0 (InputVector[0],0);
     macc_1 (InputVector[0],432);
     macc_2 (InputVector[0],843);
     macc_3 (InputVector[0],1232);

     macc_0 (InputVector[1],323);
     macc_1 (InputVector[1],956);
     macc_2 (InputVector[1],98);
     macc_3 (InputVector[1],132);

     macc_0 (InputVector[2],543);
     macc_1 (InputVector[2],2047);
     macc_2 (InputVector[2],1000);
     macc_3 (InputVector[2],1450);

     macc_0 (InputVector[3],453);
     macc_1 (InputVector[3],72);
     macc_2 (InputVector[3],113);
     macc_3 (InputVector[3],-2048);

     //Store accumulator results
     OutputVector[0]=RUR_acc0();
     OutputVector[1]=RUR_acc1();
     OutputVector[2]=RUR_acc2();
     OutputVector[3]=RUR_acc3();
}
```

In this example, the Matrix Transformation performance is improved because the overhead of memory access cycles for table access is eliminated. The processor accesses the matrix transform values directly from a table instantiated inside the processor, as opposed to memory. Using TIE tables can significantly increase performance of any algorithm that contains sets of fixed constants; however, the performance is at the expense of increased processor area (especially if the number of constants in the table is large). Also, because TIE tables are “hard-wired” and cannot be changed once an Xtensa processor is instantiated, TIE tables are not appropriate for holding values that are likely to change. For some designs, committing algorithm constants to a TIE table may be too costly in terms of processor area or too restrictive.

5.5 Immediate Operands

In the previous example we accelerated `MatrixTransform` performance by extending the processor with direct access to algorithm constants (of the Transform Matrix array) without the overhead associated with loading constants from memory. The TIE table operands are ideal for passing a limited number of random or unrelated values to TIE operations. However, if the possible values passed to an operation are from a range of linearly incrementing values, then immediate operands should be used.

An immediate operand is an operand that is embedded in the instruction word. The TIE `immediate_range` construct defines a range of linearly incrementing values that are encoded in an immediate operand. The TIE `immediate_range` construct gives the name of the immediate range, the low and high value in the range, and the step size. The step size must be a number that is a power of two. In addition, the low and high range values must be a multiple of the step size.

The TIE compiler automatically determines the minimum number of bits required to support the immediate type and allocates these bits to unused fields in the opcode space.

Following is the syntax for the immediate_range construct:

```
immediate_range <type_name> <low_range> <high_range> <step_size>
```

For example, the following `immediate_range` specifies the immediate values -16, -14, -12, ..., 10, 12, and 14:

```
immediate_range my_ir -16 14 2
```

After defining `immediate_range`, `my_ir`, you can use it as an operand type in an operation's argument list.

The `MatrixTransform` function uses an array, `TransformMatrix`, to hold the 4x4 transform matrix. Each element of the array is a 16-bit signed value. These values can be embedded as an immediate operand by first defining the `immediate_range`:

```
immediate_range my_ir -32768 32767 1
```

The `my_ir` immediate range will require 16 bits of the instruction word. The standard 24-bit TIE instruction word does not provide enough unused bits to support this immediate range. If the TIE compiler cannot allocate enough bits in the TIE instruction word to support a specified immediate range, the TIE compiler will report an error.

For LX cores Greater immediate range operands can be defined when the Xtensa processor is configured with wide TIE instruction words because additional opcode bits are available to support wider operand encoding.

The following TIE example demonstrates the use of the `my_ir` immediate operand type.

```
state acc0 32 add_read_write
state acc1 32 add_read_write
state acc2 32 add_read_write
state acc3 32 add_read_write

function [31:0] mac16 ([31:0] accumulator, [15:0] multiplier,
                       [15:0] multiplicand) shared
{
     wire do_signed = 1'b1; //signed arithmetic
     wire negate    = 1'b0; //do not negate product
     assign mac16   = TIEmac(multiplier, multiplicand, accumulator,
               do_signed, negate);
}

immediate_range my_ir -32768 32767 1

operation macc_0 {in AR oper0, in my_ir oper1}{inout acc0}
{
     assign acc0    = mac16(acc0, oper0[15:0], oper1);
}

operation macc_1 {in AR oper0, in my_ir oper1}{inout acc1}
{
     assign acc1    = mac16(acc1, oper0[15:0], oper1);
}

     operation macc_2 {in AR oper0, in my_ir oper1}{inout acc2}
{
     assign acc2    = mac16(acc2, oper0[15:0], oper1);
}

operation macc_3 {in AR oper0, in my_ir oper1}{inout acc3}
{
     assign acc3    = mac16(acc3, oper0[15:0], oper1);
}
```

Because the assembly and C syntax passes immediate operands by value, just as is done with TIE table operands, the same C code presented in the TIE table section can be used with the new TIE. Note that if an immediate operand is provided that is beyond the specified immediate range or not within the specified step size, an error is reported at C/C++ compile time.

5.6 Register Files

In the previous sections, TIE states were used to hold the accumulators of the `MatrixTransform` function. TIE state is perfect for maintaining key variables for a limited set of algorithms. However, TIE state is not well suited for the design of general acceleration processing platforms that may be used across a large number of algorithms. From a software standpoint, a programmer must manually control the allocation of variables to TIE state. In general, a TIE state is mapped to a single variable. When attempting to reuse TIE state to hold a number of key variables across different algorithms, manual state allocation (writing the right variable to the state at the right time) becomes quite difficult. TIE register files provide a much more flexible method of maintaining key algorithm variables inside the processor.

A TIE register file is a custom set of registers integrated into the Xtensa processor for dedicated use by TIE operations. The width and depth of the register file can be tuned to meet the requirements of the algorithms being accelerated. Following is the TIE construct used to define a register file:

```
regfile <name> <width> <depth> <short_name>
```

Following is a TIE register file definition example:

```
regfile myreg 80 8 mr
```

This example defines an 80-bit register file, `myreg`, with eight registers. The assembler and debugger use the short name, `mr`, to reference the individual registers in the register file. For example, `mr0` accesses the 0th register in the vector register file, `mr1` accesses the 1st register, and so on.

When a TIE register file width is defined to be less than or equal to the data memory interface width of the processor, the TIE compiler automatically generates register-to-register move, load and store operations for the register file. The automatically created operations are given the name `mv/ld/st`, followed by the register file's name (`mv.myreg`, `ld.myreg`, and `st.myreg`). The memory access width of the load/store operations is determined by the width of the register file. The memory access width can be 8, 16, 32, 64, 128, 256, or 512 bits wide. The smallest access width that is greater than the register file width will be used. In the case of the 80-bit wide `myreg` register, 128-bit load/store operations are automatically created. The most significant 48-bits of the memory accesses are ignored during a load and set as logic 0 during a store operation because the register file is only 80-bits wide.

In addition to automatically creating supporting operations for the TIE register file, the TIE compiler also extends the Xtensa C/C++ Compiler (XCC) with a new data type of the same name as the register file. A new data type named `myreg` is created for the TIE example above. Variable instances of the myreg data type are automatically allocated to the myreg register file in the same manner variables of standard data types (`int`,

short, char, and so forth) are automatically allocated to the AR register file. This enables software to use an unlimited number of myreg variables, even though there are only eight physical registers defined in the myreg register file.

Register files wider than the data memory access width can be defined; however you must create your own supporting operations and define additional supporting sections for the register file. Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for more details on defining register files wider than data memory interface width.

5.6.1 Register File Operands

When a register file is defined, a new operand type is created and given the same name as the register file. The new operand type is used in an operation's argument list to define operands that reference the new register file. The use of the register file myreg operand type is shown in the following example operation of an xor.myreg operation that performs a bitwise exclusive-or on the contents of two operands from the myreg register file.

```
regfile myreg 80 8 mr

operation xor.myreg {out myreg output, in myreg input_0, in myreg input_1}{}
{
     assign output = input_0 ^ input_1;
}
```

An operation can access multiple TIE register file entries via different register operands, up to the number of entries of the TIE register file. This includes both in and out operand accesses. Note that increasing the number of register files in operands results in additional gate count and more complex timing paths.

5.6.2 Using TIE Register Files in C/C++

After the regfile example is compiled with the TIE compiler, the C/C++ compiler is extended to support a new data type named myreg. Initialization of register data types is done differently than variables of standard C types because C has no facility for declaring wide constant values. The following code provides one example of how a variable of the myreg data type is declared and initialized.

```
//declaration of myreg variable
myreg foo;

//initialization of foo
int *p_foo = (int*)&foo;
p_foo[0]=0; p_foo[1]=0; p_foo[2]=0; p_foo[3]=0;
```

Wide data types may be initialized by assignment to a pointer that aliases the instance of the wide data type. Individual 32-bit words of the wide data variable can be initialized and accessed through the array subscript of the pointer.

If initializing to zero is all that is required, a TIE initialization operation can be created.

```
operation zero.myreg {out myreg output}{}
{
     assign output = 80'b0;
}
```

and used as follows:

```
//declaration of myreg variable
myreg foo;
//initialize foo to zero
foo = zero_myreg();
```

In some cases, it is convenient to initialize a wide variable with data from a pre-initialized array as shown in the following example:

```
#define aligned_by_16 __attribute__ ((aligned(16)))
unsigned int init_data[4] aligned_by_16;
myreg *p_init, foo;

//initialize array with initialization values
init_data[3]=0x00000000; //Most significant word (Little Endian order)
init_data[2]=0x0000ffff;
init_data[1]=0xffffffff;
init_data[0]=0xffffffff; //Least significant word

//set p_init to initialization array
p_init=(myreg*)init_data;

//and set foo to contents of p_init pointer
foo=*p_init;
```

In this example, a wide variable is initialized by pointer assignment to a pre-initialized memory location. Because 128-bit wide load/store operations are created for the `myreg` register file, the pre-initialized memory location, `init_data`, is an array that also contains 128-bits (four unsigned integer words). Note that the arrangement of words in the array is endian-dependent (ordering for a little endian processor is shown in the example).

Also note that the array, `init_data`, is declared with an alignment attribute such that the array is stored with a 16-byte (128-bit) aligned address. This is required because the automatically generated load/store operations require that memory accesses are aligned to the memory access width.

The following pseudo-code provides an example of how a variable of `myreg` data type is assigned to an unsigned integer array. This allows each word of the foo variable to be individually processed or analyzed.

```
#define aligned_by_16 __attribute__ ((aligned(16)))
unsigned int out_data[4] aligned_by_16;
myreg *p_out, foo;

//set p_out pointer to output array
p_out=(myreg*)out_data;

//assuming foo has some data
//and store foo to array
*p_out=foo;
```

5.6.3 Basic Acceleration Using TIE Register Files

In previous sections, accumulators for the `MatrixTransform` function are maintained in TIE state. Another possible TIE acceleration approach uses TIE register files instead of TIE state. The following TIE example maintains the accumulators in a TIE register file `accum`. This example does not have significant area or performance benefits compared to previous examples, but serves as a basis for a more advanced acceleration approach introduced in the following section.

```
regfile accum 32 4 ac

operation mac.accum {in AR oper0, in AR oper1, inout accum
accumulator}{}
{
     wire do_signed      = 1'b1; //signed arithmetic
     wire negate         = 1'b0; //do not negate product
     assign accumulator = TIEmac(oper0[15:0], oper1[15:0],
                    accumulator, do_signed, negate);
}
```

The `mac.accum` operation shown is similar to previous `macc` operations shown in this chapter. The main difference is that this operation example uses a user-defined TIE register file for the accumulator, rather than TIE states. This changes the way that TIE is used within the context of a C program. As stated earlier, the C compiler automatically performs register allocation of variables to the TIE register file. Variables of the `accum`

data type are declared in the same manner that other variables are declared. Instances of the `accum` data type are automatically maintained in memory and “cached” into the register file on an as-needed basis by the Xtensa optimizing compiler.

The `MatrixTransform` function is modified to take advantage of the `accum` register file and the new `mac.accum` operation. Following is the modified `MatrixTransform` function:

```
void MatrixTransform (short InputVector[], int OutputVector[],
short TransformMatrix[])
{

int i;
//declare accum variables
accum acc0, acc1, acc2, acc3;
int *p_acc0 = (int*)&acc0;
int *p_acc1 = (int*)&acc1;
int *p_acc2 = (int*)&acc2;
int *p_acc3 = (int*)&acc3;

//initialize acc
p_acc0[0]=0;
p_acc1[0]=0;
p_acc2[0]=0;
p_acc3[0]=0;

     //Perform Transformation
     for (i=0;i<4;i++)
        {
          mac_accum(InputVector[i],TransformMatrix[i],acc0);
          mac_accum(InputVector[i],TransformMatrix[i+4],acc1);
          mac_accum(InputVector[i],TransformMatrix[i+8],acc2);
          mac_accum(InputVector[i],TransformMatrix[i+12],acc3);
          }


     //write out accumulator results
     OutputVector[0]=p_acc0[0];
     OutputVector[1]=p_acc1[0];
     OutputVector[2]=p_acc2[0];
     OutputVector[3]=p_acc3[0];

}
```

5.6.4 Single Instruction/Multiple Data (SIMD) Example

Many of the most important and performance-demanding applications share a common characteristic—the same sequence of operations is performed repetitively on a stream of data operands. Many error-correction, video-compression, audio processing, and digital-signal-recovery tasks fit this model. This trait enables a set of operations where a single computational sequence can be applied many times across a group of operands. Moreover, computational logic blocks are duplicated such that the computational sequence is applied to many operands in parallel. Single Instruction/Multiple Data (SIMD) operations describe a class of operations where an identical computation is performed across a vector, or a series, of operands in parallel. TIE designed with the SIMD approach provides large performance benefits for data-intensive applications.

The SIMD approach requires vectorizing the function first. That is, the function is first restructured such that data is grouped into vectors and operations are performed on vectors rather than individual data elements. `MatrixTransform` function performance can be improved by creating TIE operations using the SIMD approach.

The main loop of the `MatrixTransform` function can be represented as:

```
OutputVector[0] = InputVector[0]*TransformMatrix[0] +
                  InputVector[1]*TransformMatrix[1] +
                  InputVector[2]*TransformMatrix[2] +
                  InputVector[3]*TransformMatrix[3];
OutputVector[1] = InputVector[0]*TransformMatrix[4] +
                  InputVector[1]*TransformMatrix[5] +
                  InputVector[2]*TransformMatrix[6] +
                  InputVector[3]*TransformMatrix[7];
OutputVector[2] = InputVector[0]*TransformMatrix[8] +
                  InputVector[1]*TransformMatrix[9] +
                  InputVector[2]*TransformMatrix[10] +
                  InputVector[3]*TransformMatrix[11];
OutputVector[3] = InputVector[0]*TransformMatrix[12] +
                  InputVector[1]*TransformMatrix[13] +
                  InputVector[2]*TransformMatrix[14] +
                  InputVector[3]*TransformMatrix[15];
```

Conceptually, this can be vectorized as follows:

```
OutputVector[0] = InputVector[0:3] . TransformMatrix[0:3];
OutputVector[1] = InputVector[0:3] . TransformMatrix[4:7];
OutputVector[2] = InputVector[0:3] . TransformMatrix[8:11];
OutputVector[3] = InputVector[0:3] . TransformMatrix[12:15];
```

where

- `InputVector[x:y]` refers to a vector consisting of InputVector elements `x` to `y`.
- `TransformMatrix[x:y]` refers to a vector consisting of `TransformMatrix` elements `x` to `y`.
- "." refers to the vector dot product operation (`[w,x,y,z]. [a,b,c,d] = w*a + x*b + y*c + z*d`).

To realize the vectorized version of the `MatrixTransform` function, the following operations and resources are defined in TIE:

- A 64-bit wide register file, `vec16x4`, to hold a vector of four 16-bit elements of the InputVector or TransformMatrix.

```
regfile vec16x4 64 8 vec
```

- A TIE operation that performs a dot product calculation on vectors in the vec16x4 register file. The dot product consists of a SIMD multiplication, followed by a summation of vector elements. These computations are fused into a single operation, `dotprod`, as follows:

```
operation dotprod {out AR acc, in vec16x4 vec, in vec16x4 mat}{}
{
//4-way SIMD multiplication
wire do_signed = 1'b1;
wire [31:0] prod0 = TIEmul(vec[15:0],  mat[15:0], do_signed);
wire [31:0] prod1 = TIEmul(vec[31:16], mat[31:16], do_signed);
wire [31:0] prod2 = TIEmul(vec[47:32], mat[47:32], do_signed);
wire [31:0] prod3 = TIEmul(vec[63:48], mat[63:48], do_signed);
//fused accumulation
assign acc = TIEaddn(prod0, prod1, prod2, prod3);
}
```

Following is the corresponding `MatrixTransform` function:

```
void MatrixTransform (vec16x4 InputVector[], int OutputVector[],
vec16x4 TransformMatrix[])
{
//Perform Transformation
OutputVector[0]=dotprod(InputVector[0],TransformMatrix[0]);
OutputVector[1]=dotprod(InputVector[0],TransformMatrix[1]);
OutputVector[2]=dotprod(InputVector[0],TransformMatrix[2]);
OutputVector[3]=dotprod(InputVector[0],TransformMatrix[3]);
}
```

Because four-way SIMD is used in this example, the performance is improved by nearly four times compared to the non-SIMD example.

5.7 Load/Store Operations

Designers can create TIE load/store operations to implement the following:

- Wide (64-, 128-, 256-, or 512-bit) memory transfers between memory and TIE resources. The TIE language enables designers to define wide states and register files, along with operations that include wide processing units. Yet, without a wide memory channel, overall performance will be gated by a 32-bit memory bottleneck. To mitigate this issue, configure the Xtensa processor with a wide (64-, 128-, 256-, or 512-bit) data bus and extend it with wide load/store operations.
- Memory access with special addressing modes. Some algorithms access memory in a specialized manner. For example, FIR filters often use circular addressing, while an FFT uses bit-reversed addressing. TIE expressions can be used to define the effective load/store address, as well as update address pointers.
- Optimized conditional memory access. Often load/store instructions are preceded by branch instructions such that the memory access occurs only based on some conditions. The TIE language allows designers to fuse a condition test with a load/store operation. When a TIE load/store operation is executed, but operands do not meet a specified condition, the operation will not issue a memory access cycle.

TIE load/store operations access the Xtensa processor's load/store unit signals. These signals are called TIE Load/Store Interfaces and are described in Table 5–3. When TIE operations perform memory access, the appropriate interfaces must be declared in the state-interface list of the operation definition.

Table 5–3. Predefined Load/Store Interface Signals

Name	Width	Direction[1]	Purpose
`VAddr`	32	out	Addresses for load and store instructions
`MemDataIn512`[2]	512	in	The 512-bit load data
`MemDataIn256`[2]	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`[2]	512	out	The 512-bit store data
`MemDataOut256`[2]	256	out	The 256-bit store data
`MemDataOut128`	128	out	The 128-bit store data
`MemDataOut64`	64	out	The 64-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.

Table 5–3. Predefined Load/Store Interface Signals (continued)

Name	Width	Direction[1]	Purpose
`MemDataOut32`	32	out	The 32-bit store data
`MemDataOut16`	16	out	The 16-bit store data
`MemDataOut8`	8	out	The 8-bit store data
`LoadByteDisable`[2]	16	out	The byte disable signal for conditional load
`StoreByteDisable`[2]	16	out	The byte disable signal for conditional stores

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.

5.7.1 Load/Store Operation Examples

When a TIE register file is defined, load/store operations are automatically created when the register file width is equal to or less than the Xtensa processor’s configured data bus width. Consider the following register file definition:

```
regfile vec16x4 64 8 vec
```

Assuming that the processor’s configured data bus width is at least 64-bits wide, the TIE compiler automatically creates load/store operations similar to the following examples:

```
immediate_range imm4x8 0 120 8
//Automatically generated load and store operation
operation ld.vec16x4 {out vec16x4 regval, in vec16x4 *addr,
                          in imm4x8 offset}{out VAddr, in MemDataIn64}
{
     assign VAddr   = addr + offset;
     assign regval  = MemDataIn64;
}
operation st.vec16x4 {in vec16x4 regval, in vec16x4 *addr,
                      in imm4x8 offset}{out VAddr, out MemDataOut64}
{
     assign VAddr        = addr + offset;
     assign MemDataOut64 = regval;
}
```

The `ld.vec16x4` operation performs a 64-bit load from memory with the effective address described by the `*addr` operand, plus an offset defined by the immediate operand `offset`. Because this 64-bit load operation requires that the effective address is aligned to a 64-bit (8 byte) boundary, the `offset` operand has a step size of 8.

Note that the "*" precedes the name of the vec16x4 operand addr. This informs the C/C++ compiler to expect a vec16x4 pointer argument, rather than a `vec16x4` argument. The MemDataIn64 interface presents the loaded data to the operation, where it is assigned to the `regval` operand. The `st.vec16x4` operation is similar, except that it outputs the value of the `regval` operand to the `MemDataOut64` interface.

Consider the following function that copies an array of `vec16x4` data. The XCC compiler will use the load and store operations generated automatically by the TIE compiler to perform the copy:

```
void vec16x4_copy (vec16x4 *dest, vec16x4 *src, int len) {
    int i;
    for (i=0; i<len; i++)
      dest[i] = src[i];
}
```

Each iteration of the loop in the copy function requires the source and destination address pointers to be updated. Thus the performance of this function can be improved by providing updating load and store operations. Updating load/store operations perform a load/store and update the address pointer to point to the next element in the array as part of the same instruction. Although updating load/store operations must be defined explicitly, the XCC compiler can infer their use in loops such as the one in the copy function. The TIE `proto` construct must be used in the TIE description to get the XCC compiler to infer these updating load/store instructions. Refer to the Proto section in the *Tensilica Instruction Extension (TIE) Language Reference Manual* for details on the `proto` construct and its usage.

Conditional load/store operations can be defined using TIE LoadByteDisable and StoreByteDisable interfaces. Asserting (setting to logic 1) these interface bits selectively disables byte accesses during a load or store operation. If all bits of the interface are asserted, memory access is completely disabled. Refer to the Interface section in the *Tensilica Instruction Extension (TIE) Language Reference Manual* for more details about designing conditional load/store operations.

5.8 Schedules

The TIE language provides the schedule construct to allow the designer to control the pipeline stage in which input operands of an instruction are read, and output operands of an instruction are written. One of the most common uses of the schedule construct is to implement multi-cycle TIE instructions, where more than one cycle is required. Without scheduling, a TIE operation is expected to execute in a single cycle. Depending on the nature of the computation, even a complex-looking series of dependent operations can still fit into one processor cycle. However, for a faster implementation (increased operating frequency), you may want to allow an operation to be pipelined across more than one clock cycle. Scheduled TIE operations are always pipelined, with the exception of

operations that use TIE functions iteratively (see Section 5.8.3). Therefore, increased operation latency does not result in reduced performance if operation results are not immediately required by subsequent operations. XCC automatically performs scheduling optimizations to “hide the latency” of scheduled operations.

Following is the syntax of the TIE schedule construct:

```
schedule <schedule_name> {operation-list} {stage_assignments}
```

The TIE schedule construct is given a unique name, a comma-delimited list of operations to be scheduled, and a set of stage assignment statements. The stage assignment statements consist of a semicolon-delimited list of `use` or `def` schedule statements. The `use` schedule defines the execution cycle at which the input operands are read. The `def` schedule defines the execution cycle at which operation results are written to output operands. `inout` operands of an instruction may have both a `use` stage and a `def` stage defined in the schedule. The pipeline stage may be specified using either integer numbers or symbolic references to the pipeline stages. Symbolic references for Xtensa pipeline stages are `Estage`, `Mstage`, and `Wstage`. In a 5-stage pipeline configuration, these references refer to integer 1, 2, and 3 respectively. In a 7-stage pipeline configuration, these references refer to integers 1, 3, and 4 respectively. Also, the pipeline stage can be specified using an integer added to a symbolic reference (i.e., Wstage + 2).

The default operand schedules for register files and state operands are as follows: `in` operands are scheduled to be `use Estage` or `use 1` (used at the beginning of the first execution cycle), and `out` operands are generally scheduled to be `def Estage` or `def 1` (defined at the end of the first execution cycle). The TIE interface use/def schedules are fixed and cannot be changed. Refer to Section 5.8.4 for fixed interface schedules. If an `out` operand is a direct assignment from an input interface, the default schedule of the out operand is the same as the use schedule of the interface.

The schedule expressions drive pipelining and retiming of the operation and automatically implement appropriate pipeline interlocks and operand-forwarding paths for any operation that might depend on these operands. The *Tensilica Instruction Extension (TIE) Language Reference Manual* describes the syntax and behavior of the schedule construct in more detail.

5.8.1 Automatic Behavioral Retiming

With the schedule statement, you can specify that an operation requires multiple cycles without having to specify how computation is divided among those cycles. TIE hardware can be retimed using a synthesis tool’s retiming feature, which pipelines logic across a defined number of clock cycles, and balances the logic for optimal timing. For additional information on using the retiming feature, see the *Xtensa Hardware User’s Guide* chapter on *Synthesis Flow*.

The dotprod operation that was previously designed contains multipliers that operate on two vectors, and an adder that calculates the sum of products. Without scheduling, this operation will be synthesized to execute in a single cycle. Due to the long timing path of the multiplier and adder network, the maximum operating frequency of the processor will be significantly reduced. To increase the operating frequency, pipeline the operation's logic across multiple clock stages using the TIE schedule construct as follows:

```
schedule dotprod_sched {dotprod}
{
     //schedule out operand
     def acc 3;
}
```

The `dotprod_sched` schedule applies to the dotprod operation. The schedule specifies that the `acc out` operand must be assigned by the end of the third execution cycle. The above schedule provides three cycles for the dotprod operation to complete. Note that the above schedule implies that the latency of the operation is three cycles. But, because the implementation is fully pipelined, the processor can continue to issue these instructions on every cycle. Due to schedule optimizations made by the Xtensa C/C++ compiler, the latency of the operation is often completely hidden. In the `MatrixTransform` example, there is no performance degradation due to adding the `dotprod_sched` schedule.

From a hardware perspective, the schedule statement creates a “template” for a three-cycle operation. The TIE compiler packs all logic required to generate the `acc` result into one execution cycle. A synthesis tool automatically balances the logic across three cycles during synthesis. Note that additional logic is added to the Xtensa processor to support operation pipelining and ensure correctness of the operation and resulting operands. It is important to check the area estimator after scheduling operations to determine the gate count increase associated with scheduling.

5.8.2 TIE Wire Scheduling

TIE Wire scheduling provides an alternative means to design multi-cycle instructions without retiming.

When an operation is scheduled, the operation’s logic is spread across multiple clock stages and pipeline flip-flops are placed between clock stages to hold intermediate calculation results. The TIE language allows the designer to specify the position of these flip-flops by specifying `def` stages for intermediate wires within an operation. Consider the following schedule example:

```
operation dotprod {out AR acc, in vec16x4 vec, in vec16x4 mat}{} {

//4-way SIMD multiplication
wire do_signed = 1'b1;
```

```
wire [31:0] prod0 = TIEmul(vec[15:0],  mat[15:0], do_signed);
wire [31:0] prod1 = TIEmul(vec[31:16], mat[31:16], do_signed);
wire [31:0] prod2 = TIEmul(vec[47:32], mat[47:32], do_signed);
wire [31:0] prod3 = TIEmul(vec[63:48], mat[63:48], do_signed);

//fused accumulation
assign acc = TIEaddn(prod0, prod1, prod2, prod3);

}

schedule dotprod_sched {dotprod}
{
     //end of cycle 1
     def prod0 Estage;
     def prod1 Estage;
     def prod2 Estage;
     def prod3 Estage;
     //end of cycle 2 - assuming 5-stage pipeline
     def acc Mstage;
}
```

The preceding schedule contains `def Estage` statements for wires `prod0-3`. This forces the logic of the `TIEmul` modules into the first execution cycle and creates flops to hold the multiplication results. There is also a `def Mstage` statement on `acc`. This forces the `TIEaddn` module into the second execution cycle.

To schedule the dotprod operation beyond two cycles (without retiming), decompose the `TIEmul` module into smaller width multiplications or into equivalent expressions using the partial product (TIEmulpp) module. Then, intermediate results can be assigned to wires that can be scheduled. Refer to the Module section of the *Tensilica Instruction Extension (TIE) Language Reference Manual* for more details on the TIEmulpp module.

It is not necessary to provide schedules for all the wires of a computation (for example, `do_signed`), because the TIE compiler automatically determines the schedule for wires without a schedule statement. Note that it is not possible to assign use stages to wires or def statements to wires inside a TIE function used by an operation.

By default, the TIE compiler marks every multi-cycle TIE operation as a candidate for retiming. Even if this operation 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. For example, the timing of the dotprod operation can be improved through pipelining the multiplies across both the first and second execution cycles (rather than just the first cycle). 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 may be because of license issues, or because of formal verification tools' limitations in comparing RTL designs to gate level netlists in which the flip-flops have moved as a result of retiming. For such flows, modifying the synthesis scripts (as described in the *Xtensa Hardware User's Guide*) can disable the retiming step. Before disabling retiming, ensure that you have appropriately scheduled the operation operands and wires to meet the timing requirements of the design.

5.8.3 Scheduling Iterative Use of Functions

Often a complex function is computed by repeated execution of a simpler function. In such scenarios, there are two choices; you can instantiate multiple copies of the function, or you can share the function across cycles using TIE iterative functions. By sharing hardware across cycles, iterative functions may require less hardware than replicated, fully pipelined hardware. However, because they share hardware across cycles, instructions that use the same iterative function cannot be issued every cycle. Whether this impacts an application's performance depends on whether XCC can find different instructions to fill in the gaps between instructions.

Consider the following example:

```
function [31:0] func_mul ([15:0] multiplier, [15:0] multiplicand,
                    do_signed) shared
{assign func_mul = TIEmul(multiplier, multiplicand, do_signed);}

operation dotprod {out AR acc, in vec16x4 vec, in vec16x4 mat}{} {

//4-way SIMD multiplication
wire do_signed = 1'b1;
wire [31:0] prod0 = func_mul(vec[15:0],  mat[15:0], do_signed);
wire [31:0] prod1 = func_mul(vec[31:16], mat[31:16], do_signed);
wire [31:0] prod2 = func_mul(vec[47:32], mat[47:32], do_signed);
wire [31:0] prod3 = func_mul(vec[63:48], mat[63:48], do_signed);

//fused accumulation
assign acc = TIEaddn(prod0, prod1, prod2, prod3);
}
```

Because function `func_mul` is defined to be shared, there is only one copy of it in the hardware. However, the operation `dotprod` uses this function four times with 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 operation, as follows:

```
schedule dotprod_sched {dotprod}
{
```

```
        //wires:  iterative function use
        def prod0 1;
        def prod1 2;
        def prod2 3;
        def prod3 4;

        //out operand
        def acc 5;
    }
```

The first iteration of the `func_mul` function happens in cycle 1. Additional iterations of the `func_mul` function occur until the fourth iteration in cycle 4. Note that shared functions must be computed within a single cycle. Thus, there is no opportunity to schedule the multiplication in the `func_mul` function across multiple cycles.

Iterative instructions take less hardware as compared to fully pipelined, multi-cycle instructions. On the other hand, iterative instructions result in processor pipeline stalls if a second operation requiring the shared function is issued while the first one is still executing. Thus, the choice between an iterative operation and a fully pipelined multi-cycle operation is based on the trade-off between hardware area and execution performance.

5.8.4 Implicit Schedule for TIE Load/Store Interface Operands

TIE load/store operations access memory through TIE interfaces. 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. Therefore, TIE interface operands have fixed use and def schedules and are implicitly scheduled as follows:

- `VAddr` is written at the end of the first execution cycle (`def 1` or `def Estage`).
- `LoadByteDisable` is written at the end of the first execution cycle (`def 1` or `def Estage`).
- `StoreByteDisable` is written at the end of the second execution cycle (`def 2`) for a 5-stage pipeline processor. For LX cores For a 7-stage pipeline processor, the interface is written at the end of the third execution cycle (`def 3`). For both a 5- and 7-stage pipeline, the schedule can be written as `def Mstage`.
- `MemDataOut{8/16/32/64/128/256/512}` is written at the end of the second execution cycle (`def 2`) for a 5-stage pipeline processor. For LX cores For a 7-stage pipeline processor, the interface is written at the end of the third execution cycle (`def 3`). For both a 5- and 7-stage pipeline, the schedule can be written as `def Mstage`.
- `MemDataIn{8/16/32/64/128/256/512}` is read at the beginning of the second execution cycle (`use 2`) for a 5-stage pipeline processor. For LX cores For a 7-stage pipeline processor, the `MemDataIn` signals are read at the beginning of the third execution cycle (`use 3`). For both a 5- and 7-stage pipeline, the schedule can be written as `use Mstage.`

Because the TIE interface scheduling shown above is implicit, it is not necessary to create a schedule for load/store operations. However, the fixed scheduling of TIE interface operands implies several limitations on how TIE load/store operations are defined. For example, designers should be careful that all assignments to `def 1` interfaces, such as the `VAddr` and `LoadByteDisable` interfaces, consist of a simple expression that is easily computed within a single cycle. It is not practical to use highly complex operators or functions in assignment statements to these interfaces.

Any destination operand that is assigned with an expression containing the `MemDataIn` interface is implicitly scheduled to `def 2` for a 5-stage pipeline. For LX cores For a 7-stage pipeline processor, the interface is implicitly scheduled to `def 3`. This implicit scheduling takes into account that data is loaded in the M stage of the Xtensa pipeline. Within the M stage of the pipeline, much of the clock cycle time is allocated to memory access. The remainder of the clock cycle is used towards logic to support assignment statements using the `MemDataIn` interface. As a result, TIE assignment statements that use the `MemDataIn` interface should be composed of simple expressions. Complex expressions using the `MemDataIn` interface may require that the destination operand/wire be manually scheduled to a later stage to prevent degradation of operating frequency. Consider the example of an operation that fuses a load and a multiplication.

```
operation LD_MUL {out AR product, in AR *addr, in AR multiplier }
                                {out VAddr, in MemDataIn32}
{
     assign VAddr = addr;
     assign product = MemDataIn32 * multiplier;
}
```

This example significantly decreases the maximum operating frequency of the processor because the multiplication logic is constrained to a small fraction of a clock cycle. The following schedule improves the maximum operating frequency by providing an additional cycle for the multiplication.

```
schedule sched {LD_MUL}
{
     //implicit schedules
     //def VAddr 1;
     //use MemDataIn32 2;

     def product 3;
}
```

5.9 FLIX For LX cores

In the previous sections, TIE operations were designed with both fusion and SIMD approaches to accelerate algorithms. A third approach to accelerate algorithms takes advantage of the Flexible Length Instruction eXtensions (FLIX) technology of the Xtensa LX processor. FLIX technology allows modeless blending of designer-defined, VLIW-style instructions in to code that contains 16- or 24-bit instructions. Each VLIW-style wide instruction combines multiple operations in to one instruction word of flexible width, allowing the Xtensa processor to execute several instructions simultaneously. The width of these wide instructions can be any multiple of 8-bit value between 32 and 128.

FLIX broadens the flexibility of solutions developed with TIE. The FLIX approach begins with the definition of general-purpose operations that may be used across many algorithms. Rather than combining operations into complex fused operations, general-purpose operations are automatically pipelined and packed into instruction packets at compile time. The Xtensa C/C++ compiler (XCC) automatically creates different instructions from available operations depending upon operation usage in the application.

Along with the standard 24-bit TIE instructions, the Xtensa LX processor allows FLIX instructions of wide widths between 32 and 128 bits, in multiples of 8. Instructions of different widths can be freely intermixed without setting any “mode bit" The Xtensa processor has no mode setting for instruction size. It identifies, decodes, and executes any mix of 16-bit, 24-bit, and FLIX instructions from the incoming instruction stream. The FLIX instruction can be divided into slots, with independent operations placed in one, all, or some of the slots. The slots need not be equally sized; hence, the name FLIX (flexible length instruction extensions). Any combination of the operations allowed in each slot can occupy a single FLIX instruction word.

5.9.1 Defining FLIX Instructions

The TIE constructs used to enable packing of operations into FLIX instructions are format and slot_opcodes. The format construct defines FLIX instruction coding formats that consist of one or more operation slots. The syntax of the `format` construct is:

```
format <format_name> <width_between_32_and_128_in_multiple_of_8>
{<operation_slot_list>}
```

Multiple formats can be defined in a single TIE file, all of which are given a unique format name. Any number of instruction widths can be defined in the TIE file. The maximum instruction width is specified in the Xtensa processor configuration. A width that is smaller than the maximum instruction width can also be specified. The instruction format contains one or more operation slots. These slots are given unique slot names, and are provided in a comma-delimited list in the operation_slot_ list.

The slot_opcodes construct defines all possible operations that can be assigned to a slot. Following is the syntax of the `slot_opcodes` construct:

```
slot_opcodes <operation_slot> {<operation_list>}
```

Each slot listed in format definitions requires a slot_opcodes definition. The `slot_opcodes` construct is followed by the name of the slot, and a comma-delimited list of operations that can be assigned to a slot. The TIE compiler determines the bit width requirements of each slot based upon the operations assigned to it, and automatically allocates opcode bits to each of the slots.

Consider the following simple example:

```
format flix_3 64 {slot_0, slot_1, slot_2}
slot_opcodes slot_0 {LD_INST, ST_INST}
slot_opcodes slot_1 {FOO}
slot_opcodes slot_2 {BAR}
```

In this example, the `flix_3` format defines an 8-byte FLIX instruction with three slots: `slot_0`, `slot_1`, and `slot_2`. `NOP`s are always implicitly included in each slot. Therefore, `slot_0` can hold a `LD_INST`, `ST_INST`, or a `NOP` operation. Likewise, `slot_1` can hold a `FOO` or a `NOP`, and `slot_2` can hold a `BAR` or a `NOP` operation. This allows the following FLIX instructions to be issued (where each instruction is contained within the {} braces and each slot within separated by a comma):

```
{LD_INST, FOO, BAR}   ,
{ST_INST, FOO, BAR}   ,
{LD_INST, NOP, BAR}   ,
{LD_INST, FOO, NOP}   ,
{ST_INST, NOP, BAR}   ,
{ST_INST, FOO, NOP}   ,
{NOP, FOO, BAR}       ,
{NOP, NOP, NOP}       ,
{ST_INST, NOP, NOP}   ,
{LD_INST, NOP, NOP}   ,
{NOP, NOP, BAR}       ,
{NOP, FOO, NOP}
```

Note that the last set of instructions contains only a single operation (non-NOP) in each 64-bit instruction. Depending upon how often these instructions are issued, code size may be reduced by mapping each operation to a 24-bit instruction. There is a built-in format for standard 24-bit instructions with a single slot named `Inst`. By mapping single

operations to the Inst slot, a 24-bit instruction can be used instead of a 64-bit instruction. Note that all operations are mapped to the Inst slot by default, unless explicitly mapped to another slot.

```
slot_opcodes Inst {LD_INST, ST_INST}
```

The preceding example maps the operation LD_INST and ST_INST to the Inst slot, thus these instructions can be executed as standard 24-bit instructions. Because the FOO and BAR operations are not mapped to the Inst slot, they can only be executed as part of a 64-bit FLIX instruction.

5.9.2 Accelerating Performance with FLIX

Prior to creating FLIX instructions, it is often useful to analyze performance of a function that has already been accelerated with TIE. Consider the following example of TIE used to accelerate the MatrixTransform function using the SIMD approach.

```
regfile vector 64 8 v
regfile widevector 128 8 w

function [127:0] func_mul4 ([63:0] multiplier, [63:0] multiplicand,
do_signed)
{
     //4 way SIMD multiplier
     wire [31:0] prod0 = TIEmul(multiplier[15:0], multiplicand[15:0],
                    do_signed);
     wire [31:0] prod1 = TIEmul(multiplier[31:16], multiplicand[31:16],
                    do_signed);
     wire [31:0] prod2 = TIEmul(multiplier[47:32], multiplicand[47:32],
                    do_signed);
     wire [31:0] prod3 = TIEmul(multiplier[63:48], multiplicand[63:48],
                    do_signed);
     assign func_mul4 = {prod0, prod1, prod2, prod3};
}

operation mul4 {out widevector out_vector, in vector vec, in vector mat}{}
{
     wire do_signed  = 1'b1;
     assign out_vector = func_mul4(vec, mat, do_signed);
}
//schedule multiplies to 2 cycles
schedule my_sched {mul4} {def out_vector 2;}

//Accumulation of vector values
operation accum4 {out widevector out_vector, in widevector vec0, in
widevector vec1,
     in widevector vec2, in widevector vec3}{}
```

```
{

     wire [31:0] acc0 = TIEaddn(vec0[31:0], vec0[63:32], vec0[95:64],
vec0[127:96]);
     wire [31:0] acc1 = TIEaddn(vec1[31:0], vec1[63:32], vec1[95:64],
vec1[127:96]);
     wire [31:0] acc2 = TIEaddn(vec2[31:0], vec2[63:32], vec2[95:64],
vec2[127:96]);
     wire [31:0] acc3 = TIEaddn(vec3[31:0], vec3[63:32], vec3[95:64],
vec3[127:96]);

     assign out_vector = {acc3, acc2, acc1, acc0};
}
```

In this example, there are two register files: `vector` and `widevector`. The dotprod operation that was described in previous sections has been decomposed to two general-purpose vector operations, `mul4` and `accum4`. The `mul4` operation takes two 16-bit x 4 vectors from the `vector` register, performs a SIMD multiplication on these vectors, and stores the resulting 32-bit x 4 vector to the `widevector` register. A second operation (`accum4`) takes four 32-bit x 4 vectors from the `widevector` register, performs a SIMD reduction by accumulation of each vector, and stores the sums as a 32-bit x 4 vector to a `widevector` register. A major benefit of decomposing a fused operation such as dotprod into general-purpose vector operations is that the operations are more likely to be reused across other similar applications.

The following `SIMDMatrixTransform` function performs 16 4x4 matrix transformations on 16 vectors using the defined TIE operations.

```
void SIMDMatrixTransform (short InputVector[][], int OutputVector[][],
short TransformMatrix[])
{
     //declare vector variables
     vector *ptr_VectorIn;
     vector *ptr_Matrix;
     widevector w0, w1, w2, w3;
     widevector *ptr_VectorOut;
     int i;

     //initialize vector variables through pointer de-reference
     ptr_VectorIn=(vector*)InputVector;
     ptr_VectorOut=(widevector*)OutputVector;
     ptr_Matrix=(vector*)TransformMatrix;

     for (i=0; i<16; i++)
     {
          //perform dotprod operation on vectors and write out
          w0=mul4(ptr_VectorIn[i],ptr_Matrix[0]);
          w1=mul4(ptr_VectorIn[i],ptr_Matrix[1]);
```

```
            w2=mul4(ptr_VectorIn[i],ptr_Matrix[2]);
            w3=mul4(ptr_VectorIn[i],ptr_Matrix[3]);
            ptr_VectorOut[i]=accum4(w3,w2,w1,w0);
        }
    }
```

The FLIX approach accelerates performance by executing multiple operations simultaneously. Through detailed analysis of the algorithm, the TIE designer determines which set of operations provides the greatest performance when executed simultaneously. These operations are then assigned to individual operation slots using the slot_opcodes construct such that they can be grouped together into FLIX instructions. One approach of mapping the operations of the `SIMDMatrixTransform` example to FLIX instructions follows.

```
//FLIX Instruction definition
format flix64 64 {slot_0, slot_1, slot_2}
slot_opcodes slot_0 {st.vector, st.widevector, ld.vector,
ld.widevector}
slot_opcodes slot_1 {mul4, accum4}
slot_opcodes slot_2 {mul4}
```

This FLIX example assigns the automatically generated TIE load/store operations into the first slot, assigns all remaining TIE operations into the second slot, and then duplicates the highest utilized operations (such as `mul4`) in the third slot. The FLIX definition allows TIE load/store operations to be issued concurrently with both performance critical operations: `mul4` and `accum4`. The FLIX optimization is demonstrated in the assembly profile shown in Figure 5–14. The FLIX approach significantly improves the performance of the `SIMDMatrixTransform` function compared to the SIMD approach alone.

Note that no change to the C/C++ code is required to take advantage of FLIX. The XCC compiler automatically packs TIE operations into FLIX instructions and uses software pipelining on the main loop of the `SIMDMatrixTransform` function to optimize performance.

When using FLIX, it is important to determine hardware impacts by viewing the TIE estimator results. In the previous example, the MUL4 operation was assigned to two slots. To support this requirement, the TIE compiler must instantiate two copies of the hardware required to execute the MUL4 operation. Because each MUL4 operation requires four 16x16 multipliers, this results in a substantial increase in gate count.

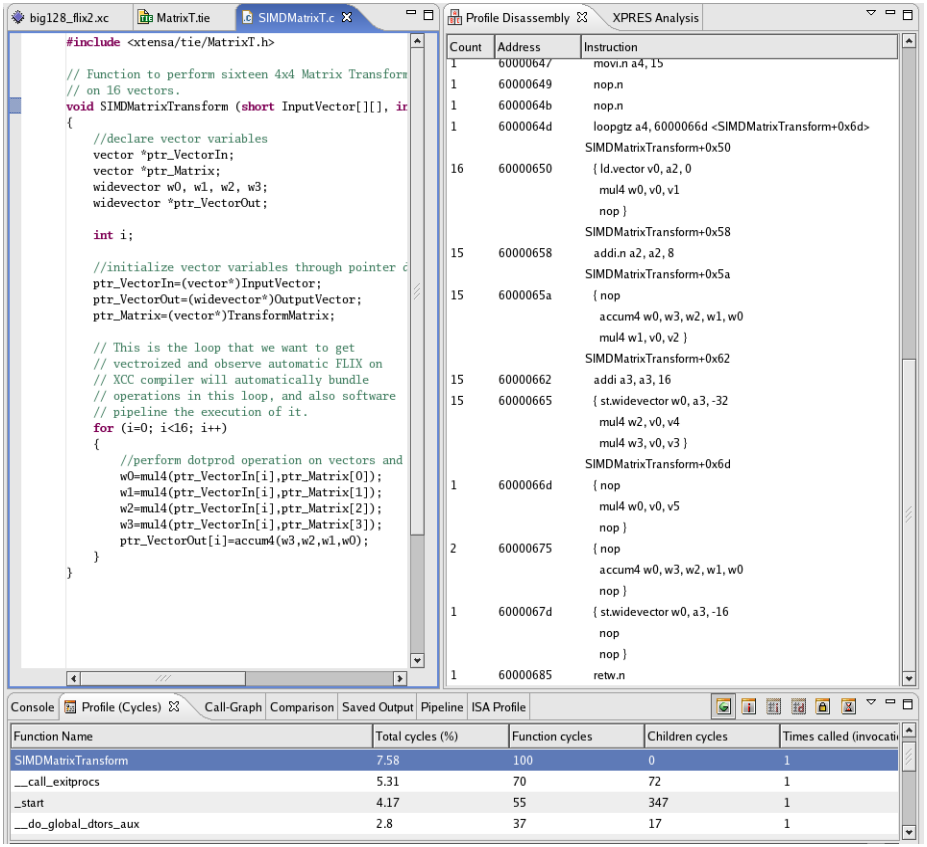

Figure 5–14. Assembly Profile of MatrixTransform using FLIX

5.9.3 FLIX Hardware Sharing

The TIE compiler groups operation hardware by slot index. A slot index refers to the position of a slot in an instruction format's slot list. In the TIE example below, `slot_a0`, `slot_b0`, and `Inst` occupy slot index 0. `slot_a1` and `slot_b1` occupies slot index 1.

```
format flix_a 64 {slot_a0, slot_a1}
format flix_b 64 {slot_b0, slot_b1}
//built-in format {Inst}

//SLOT INDEX 0
slot_opcodes slot_a0 {mul4, ADD}
slot_opcodes slot_b0 {mul4}
slot_opcodes Inst {mul4}

//SLOT INDEX 1
```

```
slot_opcodes slot_a1 {mul4}
slot_opcodes slot_b1 {ADD}
```

Hardware requirements for the preceding example are determined using the following rules:

- When the slots are within the same slot index, the same operation in multiple slots share operation hardware. In the example, `slot_a0`, `slot_b0`, and `Inst` share the MUL4 operation hardware. On the other hand, the MUL4 operation in `slot_a1` is in a different slot index and requires its own copy of MUL4 hardware. If operations contain non-shared functions (as the MUL4 operation does), the functions are also replicated along with the operation hardware. Because MUL4 operation appears in both `slot_a0` and `slot_a1`, two MUL4 operations can be issued in the same cycle. It is not possible to share hardware between them.
- Register file read/write ports can be shared across slot indexes if they are never accessed in parallel. Register file ports can also be shared within a slot index if they belong to different formats. Furthermore, if the register file ports are within a format, but not accessed in parallel, they can still be shared. In this example, the MUL4 operation appears in slots `slot_a0` and `slot_a1` of format `flix_a`, so two instances of the operation may be issued in the same cycle. As a result, four read ports are generated for the vector register file and two write ports to the widevector register file. The ADD operation appears in slots `Inst`, `slot_a0`, and `slot_b1`. As each slot belongs to a different format, they cannot be issued in the same cycle, and so only two read ports and one write port are generated[1].
- The `Inst` slot implicitly contains all Xtensa ISA operations. Designers can add Xtensa ISA operations to a slot (refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a complete list of Xtensa ISA operations that can be used). Assigning the Xtensa ISA operations to slot index 0 allows reuse of existing Xtensa ISA hardware. Assigning the Xtensa ISA operations to other slot indices will infer new instances of hardware to support the operation. For example, the ADD operation used in `slot_a0` will reuse the existing Xtensa arithmetic logic unit (ALU), because both `slot_a0` and `Inst` are in slot index 0. In contrast, new hardware will be required by default to support the ADD operation used in `slot_a1`, which is in slot index 1. Because the ADD operations in slot index 0 and index 1 can not be executed in parallel, one instance is sufficient. If the `shared_semantic` property is specified in the TIE file, one instance is created. Detailed description of the `shared_semantic` property can be found at the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

1. Before release RC-2009.0, register file read/write ports are not shared across slot indices (four read ports and two write ports would be implemented to support the ADD in this example.

- Because most Xtensa ISA operations are implicitly allocated to the `Inst` slot, they can be made available to all slots of slot index 0, without additional hardware replication. This provides the XCC compiler with more opportunities to bundle operations and accelerate code performance. The following example shows `slot_a0` with many Xtensa ISA operations added to the `slot_opcode` list.

```
slot_opcodes slot_a0 {
//TIE OPS
MUL4,
//XTENSA ISA OPS
L16SI, L16UI, L32I, L32I.N, L32R, L8UI, S16I, S32I, S32I.N, S8I,ABS,
ADD, ADD.N, ADDI, ADDI.N, ADDMI, ADDX2, ADDX4, ADDX8, ALL4,ALL8, AND,
ANDB, ANDBC, ANY4, ANY8, NEG, OR, ORB, ORBC, SUB, SUBX2, SUBX4, SUBX8,
XOR, XORB,SLL, SLLI, SRA, SRAI, SRC, SRL, SRLI, SSA8B, SSA8L,SSAI,SSL,
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, J, JX,MOV.N, MOVEQZ, MOVF, MOVGEZ, MOVI, MOVI.N,
MOVLTZ, MOVNEZ, MOVT,MUL16S, MUL16U}
```

5.9.4 FLIX Function Sharing

Consider the case where operations containing shared functions are assigned to different slot indices, as follows:

```
function [127:0] func_mul4 ([63:0] multiplier, [63:0] multiplicand,
                                do_signed) shared
{
     wire [31:0] prod0= TIEmul(multiplier[15:0],multiplicand[15:0],
          do_signed);
     wire [63:32] prod1 = TIEmul(multiplier[31:16], multiplicand[31:16],
          do_signed);
     wire [95:64] prod2 = TIEmul(multiplier[47:32], multiplicand[47:32],
          do_signed);
     wire [127:96] prod3= TIEmul(multiplier[63:48], multiplicand[63:48],
          do_signed);
     assign func_mul4 = {prod0, prod1, prod2, prod3};
}

operation mul4 {out widevector out_vector, in vector vec, in vector mat}{}
{
     wire do_signed = 1'b1;
     assign out_vector = func_mul4(vec, mat, do_signed);
}

format flix_a 64 {slot_0, slot_1}

//SLOT INDEX 0
```

```
slot_opcodes slot_0 {mul4,....}

//SLOT INDEX 1
slot_opcodes slot_1 {mul4,....}
```

Hardware for a “shared” function is instantiated only once. If a set of operations contain a “shared” function and the operations are assigned to different slot indices, the operations cannot be packed into the same FLIX instruction. In the preceding example, the FLIX instruction `{mul4, mul4}` is illegal. The XCC compiler will never create a `{mul4, mul4}` instruction. Attempting to use `{mul4, mul4}` in assembly code will cause an assembler error.

Functions can also be defined with the slot_shared attribute. Hardware for a slot-shared function is instantiated once per each slot index that contains operations using the function. The function hardware is shared among all operations assigned to a slot. Consider the case where operations containing slot-shared functions are assigned to different slot indices.

```
function [127:0] func_mul4 ([63:0] multiplier, [63:0] multiplicand,
                     do_signed) slot_shared
{
     wire [31:0] prod0= TIEmul(multiplier[15:0], multiplicand[15:0],
                                    do_signed);
     wire [63:32] prod1 = TIEmul(multiplier[31:16],
                                    multiplicand[31:16], do_signed);
     wire [95:64] prod2 = TIEmul(multiplier[47:32],
                                    multiplicand[47:32], do_signed);
     wire [127:96] prod3 = TIEmul(multiplier[63:48],
                                    multiplicand[63:48], do_signed);
     assign func_mul4 = {prod0, prod1, prod2, prod3};
}

//signed SIMD multiplication
operation mul4 {out widevector out_vector, in vector vec,
                          in vector mat}{}
{
     wire do_signed = 1'b1;
     assign out_vector = func_mul4(vec, mat, do_signed);
}

//unsigned SIMD multiplication
operation u.mul4 {out widevector out_vector, in vector vec,
                     in vector mat}{}
{
     wire do_signed = 1'b0;
     assign out_vector = func_mul4(vec, mat, do_signed);
}
```

```
format flix_a 64 {slot_0, slot_1}

//SLOT INDEX 0
slot_opcodes slot_0 {mul4, u.mul4,....}

//SLOT INDEX 1
slot_opcodes slot_1 {mul4, u.mul4,....}
```

The TIE compiler determines that the slot-shared function is used across two different slot indices and creates an instance of function hardware for each slot index. In the preceding example, the `func_mul4` function is instantiated twice, once for slot index 0 and another time for slot index 1. The `mul4` and `u.mul4` operations in `slot_0` share the same `func_mul4` function hardware, while the `mul4` and `u.mul4` operations in `slot_1` share the other `func_mul4` hardware.

5.9.5 Dual Load/Store Instructions

Depending on the algorithms being accelerated, a significant performance boost can be attained through the use of two load/store units. Xtensa LX processors can be configured with two load/store units. If two load/store elements are configured, all of the memory-address, data-input, data-output, and byte-lane-enable signals for all data memory interfaces (DataRam, DataRom, and XLMI) are duplicated.

Each TIE operation is limited to controlling a single load/store unit. To benefit from the dual load/store unit, FLIX is used to bundle two load/store operations into a FLIX instruction. The following FLIX example creates a four-slot instruction format. All load/store operations are assigned to both `slot_a` and `slot_d`.

```
//FLIX Instruction definition
format flix64 64 {slot_a, slot_b, slot_c, slot_d}
slot_opcodes slot_a {st.vector, st.widevector, ld.vector,
ld.widevector}
slot_opcodes slot_b {mul4, accum4}
slot_opcodes slot_c {mul4}
slot_opcodes slot_d {st.vector, st.widevector, ld.vector,
ld.widevector}
```

If the underlying Xtensa LX processor is configured with one load/store element, load/store operations can only be assigned to slot index 0 (slot_a belongs to slot index 0). If the processor is configured with two load/store elements, load/store operations can be assigned to one other slot index (besides slot index 0). In this example, the second load/store element is assigned to slot index 3 (slot_d belongs to slot index 3).

5.10 TIE Ports, Queues and Lookups For LX cores

TIE ports, queues and lookups provide a flexible mechanism for defining new external interfaces of the Xtensa LX processor. These interfaces allow direct coupling between processors and surrounding hardware functionality, to provide higher-bandwidth communications among processors and system devices. The TIE ports features include a mechanism to import signals into the processor via import wires and generate status output signals via state export wires—providing a General Purpose I/O (GPIO) capability. The TIE queues feature provides a mechanism to operate directly on input and output data through a queue interface—providing a "FIFO-like" interface with all necessary controls. The TIE lookup feature allows you to generate a request for data from an external device (for example, memory) and receive that data a fixed number of clock cycles later as part of one single TIE instruction.

The Xtensa Instruction Set Simulator can perform software simulation of processors with TIE ports and queues. The Instruction Set Simulator is capable of being driven by input ports/queues and capturing the status of output ports/queues. For more advanced simulation, the Xtensa Modeling Protocol (XTMP) environment is used to model connection to system devices or other processors cores. Refer to the *Xtensa Instruction Set Simulator (ISS) User's Guide* for more information regarding the use of ISS/XTMP for simulation of Xtensa LX processors that have TIE ports/queues.

5.10.1 TIE Output Port: State Export

TIE states can be defined to export a TIE state's data to signals at the boundary of the Xtensa processor. Use this feature to have the status of the processor visible to an external device or another processor. State export signals output the committed value of a TIE state register on wires specifically generated for this task by the Xtensa processor generator. The value of the state output signals change at or after the completion of the W stage. Therefore, speculative writes to state operands are never presented to the output.

Exported states are specified with the keyword `export` in the state declaration. Exporting a state adds an output port to the Xtensa processor boundary with the name `TIE_<state_name>` and is the same width as the state. Exported states must also have a reset value, the initial value presented to the external interface when the processor is reset.

Consider the definition of a status wire driven by the Xtensa LX processor that is used to inform external logic when the processor is busy transforming vectors.

```
state busy 1 1'b0 export
```

The preceding state definition creates a 1-bit state named `busy` and an output signal at the processor boundary, named `TIE_busy`. Upon reset, the `TIE_busy` output is driven to logic 0. The following example operations are executed to set, reset, and check the state of the TIE_busy signal.

```
operation set_busy {}{out busy}{assign busy = 1'b1;}
operation reset_busy {}{out busy}{assign busy = 1'b0;}
operation rd_busy {out AR busy_state}{in busy}
{assign busy_state = busy;}
```

5.10.2 TIE Input Port: Import Wires

TIE operations can be defined with implicit operands from input signals at the boundary of the processor. A TIE import wire is a port whose value is taken directly from a set of automatically generated wires at the interface of the processor. TIE operations can use import wires like other operands in TIE expressions. Note that there is no automatic signal that informs external logic when the import wire is read. Moreover, import wires may be read speculatively. Therefore, import wires are not suited for connection to devices with read-side-effects (for example, FIFOs). The import wires are registered before the first execution cycle of the TIE operation that uses it (import wires are always scheduled as "use 1"). Following is the import wire syntax:

```
import_wire <name> <width>
```

The TIE `import_wire` construct creates an input port at the boundary of the processor, named `TIE_`*<name>*, with a specified bit-width. The input port is made available to operations through an input operand of the same name.

Consider the definition of an input status signal that informs the processor when a resource contains a set of vectors to transform:

```
import_wire ready 1
operation devrdy {out AR rdy} {in ready}
{assign rdy = ready;}
```

An input port is created with the name, `TIE_ready`. The operation, `devrdy`, is issued to return the status of the `TIE_ready` input port where it can be processed by software.

5.10.3 TIE Queues

The TIE queue construct defines an interface for designer-defined instructions to read or write data from an external queue (for example, a FIFO). This provides a mechanism for high bandwidth streaming of data between Xtensa LX processors and other system devices. The queue construct does not instantiate a queue inside the Xtensa processor, but provides an interface to a queue that is external to the processor. Designers must

connect the Xtensa LX processor to an output queue that contains a Push request and Full control signal, or an input queue that contains a Pop request and Empty control signal. Following is the syntax for a TIE queue:

```
queue <queue_name> <width> <in/out>
```

The TIE queue construct creates an input or output queue interface to the processor. Queues have a name, a width, and a direction (in or out). This allows queues to serve as implicit operands to TIE operations: an input queue is an `in` operand and an output queue serves as an `out` operand. A simple send and receive mechanism used to transfer messages between Xtensa processors might be implemented as follows:

```
queue QO 32 out
queue QI 32 in
operation RECV {out AR mes} {in QI} {assign mes = QI;}
operation SEND {in AR mes} {out QO} {assign QO = mes;}
```

This TIE example creates two queue interfaces: `QO` is a 32-bit output queue interface and `QI` is a 32-bit input queue interface. The `RECV` operation transfers data from `QI` to the `AR` register file operand, while the send operation transfers data from the `AR` register file operand to `QO`.

The declaration of an input queue interface named `QI` creates three signals in the processor-interface hardware implementation:

```
input [31:0] TIE_QI
input TIE_QI_Empty /* Empty causes stall on attempted use */
output TIE_QI_PopReq /* Request pop from input queue*/
```

When the Xtensa processor wants to read data from this queue, it asserts the `TIE_QI_PopReq` signal. If the queue is not empty at this time (that is, `TIE_QI_Empty` is deasserted), the Xtensa processor samples the data on the `TIE_QI` interface. This data is available to an operation in the M (Memory Access) stage of the processor pipeline.

The Xtensa processor may buffer an input queue read arbitrarily ahead of the operation performing the input queue pop. For example, if an operation that pops an input queue gets killed (perhaps by an exception), the data read from the queue is saved in an internal buffer within the processor. The next operation that reads the queue will use the saved data from this buffer instead of popping new data from the queue. This buffering hides the speculative nature of the processor pipeline.

When the internal buffer corresponding to an input queue is empty, and the external queue is also empty as indicated by `TIE_QI_Empty` being asserted, an operation attempting to read the queue will stall the processor. The processor will be stalled until the external queue has data, that is, until `TIE_QI_Empty` becomes deasserted.

The declaration of an output queue interface named QO creates the following processor ports:

```
output [31:0] TIE_QO
input TIE_QO_Full   /* Causes a stall on attempted use if full */
output TIE_QO_PushReq /* request push to output queue*/
```

When the Xtensa processor wants to write data to this queue, it asserts the TIE_QO_PushReq signal and provides the write data on the TIE_QO interface. If the queue is not full at this time (that is, TIE_QO_Full is deasserted), the queue push is successful and the external queue is assumed to have sampled the data on TIE_QO. The operation that writes to the output queue is required to provide the write data in the M stage of the processor pipeline.

The output queue interface also has a buffer internal to the Xtensa processor. Its purpose is to buffer speculative writes and to present them to the external queue only after the instruction writing the data has committed.

When the internal buffer corresponding to an output queue is full, and the external queue is also full as indicated by TIE_QO_Full being asserted, an operation attempting to write the queue will stall the processor. The processor will be stalled until the external queue is no longer full, that is, until TIE_QO_Full becomes deasserted.

The following TIE is used to perform a Matrix Transformation on a 16x4 vector operand. The input vector is read from the input queue, and stored in a TIE state when the READ_Q operation is executed. The contents of the state are transformed into a 32x4 vector that is output to the output queue when the WRITE_Q operation is executed.

```
queue INPQ 64 in
queue OUTQ 128 out

regfile qbuffer 64 2 qb

table matrix 32 16
{0,323,543,453,432,956,2047,72,843,98,1000,113,1232,132,1450,-2048}

operation READ_Q {out qbuffer buffer}{in INPQ}{assign buffer = INPQ;}

operation WRITE_Q {in qbuffer buffer}{out OUTQ}
{
     wire [15:0] in0 = buffer[15:0];
     wire [15:0] in1 = buffer[31:16];
     wire [15:0] in2 = buffer[47:32];
     wire [15:0] in3 = buffer[63:48];

     wire do_signed = 1'b1;
```

```
        wire [31:0] prod0_a = TIEmul(in0, matrix[0], do_signed);
        wire [31:0] prod0_b = TIEmul(in1, matrix[1], do_signed);
        wire [31:0] prod0_c = TIEmul(in2, matrix[2], do_signed);
        wire [31:0] prod0_d = TIEmul(in3, matrix[3], do_signed);
        wire [31:0] out0 = TIEaddn(prod0_d, prod0_c, prod0_b, prod0_a);

        wire [31:0] prod1_a = TIEmul(in0, matrix[4], do_signed);
        wire [31:0] prod1_b = TIEmul(in1, matrix[5], do_signed);
        wire [31:0] prod1_c = TIEmul(in2, matrix[6], do_signed);
        wire [31:0] prod1_d = TIEmul(in3, matrix[7], do_signed);
        wire [31:0] out1 = TIEaddn(prod1_d, prod1_c, prod1_b, prod1_a);

        wire [31:0] prod2_a = TIEmul(in0, matrix[8], do_signed);
        wire [31:0] prod2_b = TIEmul(in1, matrix[9], do_signed);
        wire [31:0] prod2_c = TIEmul(in2, matrix[10], do_signed);
        wire [31:0] prod2_d = TIEmul(in3, matrix[11], do_signed);
        wire [31:0] out2 = TIEaddn(prod2_d, prod2_c, prod2_b, prod2_a);

        wire [31:0] prod3_a = TIEmul(in0, matrix[12], do_signed);
        wire [31:0] prod3_b = TIEmul(in1, matrix[13], do_signed);
        wire [31:0] prod3_c = TIEmul(in2, matrix[14], do_signed);
        wire [31:0] prod3_d = TIEmul(in3, matrix[15], do_signed);
        wire [31:0] out3 = TIEaddn(prod3_d, prod3_c, prod3_b, prod3_a);

        assign OUTQ = {out3, out2, out1, out0};
    }

    //implicit queue op schedules for five-stage pipeline
    //schedule read_q_sched {READ_Q} {use INPQ 2; def buffer 2;}
    //schedule write_q_sched {WRITE_Q} {use buffer 1; def OUTQ 2;}

    format flix64 64 {slot_0, slot_1}
    slot_opcodes slot_0 {READ_Q}
    slot_opcodes slot_1 {WRITE_Q}
```

The following example C program shows how the new TIE port examples are used in a task engine to transform 1024 vectors:

```
    void MatrixTransformTask (void)
    {
    qbuffer temp;
    int i;
    //Check TIE_ready input to make sure that
    //vectors are available to transform
    if (devrdy()==1) {
         set_busy();    // assert TIE_busy output upon starting transform
         #pragma flush    // Assure that TIE_busy is asserted prior to
    transform
              for (i=0; i<1024; i++)
```

```
                {
                temp=READ_Q();
                WRITE_Q(temp);
                }
        #pragma flush // Assure last queue write prior to deasserting busy
        reset_busy();// deassert TIE_busy output upon completion
        }
    return;
    }
```

A scheduler can periodically call the example function, MatrixTransformTask. Whenever the function is called, it transforms 1024 vectors if the `TIE_ready` input is asserted, otherwise the function will return. The TIE solution demonstrates the high processing bandwidth the Xtensa LX processor can attain when several Xtensa LX features and acceleration approaches are combined. With FLIX and XCC software pipeline optimizations, vectors are transformed at a rate of one vector every cycle.

Although the transform performance approaches one vector every cycle, the actual performance is dependent on how fast external devices can fill `INPQ`, and empty `OUTQ`. If the `INPQ` queue ever becomes empty (`TIE_INPQ_Empty` is asserted), the processor will be stalled and cannot perform any work until new vectors are put into INPQ queue. Likewise, if the `OUTQ` queue ever becomes full (`TIE_OUTQ_Full` is asserted), the processor will be stalled until vectors are read out of the `OUTQ` queue.

In the C example, a special pragma, `#pragma flush`, is used. This pragma directs XCC to insert an EXTW instruction at the point of the pragma. This assures that all memory or TIE port references issued before the point of the pragma are guaranteed to complete before executing any memory or TIE port references issued after the pragma. Without this pragma, the READ_Q operation may execute before the `TIE_busy` signal is asserted, and the busy signal may be deasserted prior to the final WRITE_Q operation. It is important to place the `#pragma flush` pragma in code to avoid system deadlock problems when TIE I/O operations are reordered.

When designing systems with TIE ports and queues, designers must take into account potential synchronization problems that may arise due to reordering caused by the Xtensa compiler. For more information regarding synchronization and ordering issues when using this feature, refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* and *Xtensa C and C++ Compiler User's Guide*. Also refer to the *Xtensa LX Microprocessor Data Book* for information about TIE port/queue timing and interface issues.

Conditional Access of TIE Queues

The TIE queue feature of stalling the processor on an empty/full queue can be useful in synchronizing the processor with the producer/consumer of the data. This is especially true in situations where the processor has nothing else to do while waiting. On the other

hand, there may be situations in which you wish to avoid stalling the processor when the external queue is not ready. Perhaps the processor can perform some other task while the queue is not ready for an access, or the instruction may want to conditionally access the queue based on a run-time condition. This functionality is achieved by using the _KILL interface of TIE queues as illustrated in the following example.

```
queue IPQ 8 in    // Input queue to read 8-bit data from
state BITREG 8    // 8-bit shift register
state COUNT 3     // Count of bits in BITREG

operation BITSHFT {inout AR val} {inout BITREG, inout COUNT, in IPQ,
                                  out IPQ_KILL}
{
   assign val = {val[31:1], BITREG[7]};
   assign BITREG = (COUNT == 0) ? IPQ :{BITREG[6:0], 1'b0};
   assign COUNT = COUNT - 1;
   assign IPQ_KILL = (COUNT != 0);
}
```

In the preceding example, IPQ is an 8-bit wide input queue from which you want to extract one bit at a time. BITREG is an 8-bit shift register used to store the popped queue data. Every time the BITSHFT instruction is executed, the most significant bit of BITREG is extracted and shifted into the output AR register. When the BITREG register is empty, it pops another data value from the input queue.

The input queue only needs to be read when the count of bits in BITREG is zero. For any other count value, the interface IPQ_KILL is assigned a value of logic 1, which kills the input queue access. This means that the data from the input queue is not consumed, and if the queue were to be empty, the instruction would not stall.

The _KILL interface is also available for output queues. Please refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for further information on the _KILL interface, on the _NOTRDY interface of queues and the ability to peek at input queue data without popping it.

5.10.4 TIE Lookups

Each TIE port or queue is a unidirectional data interface in the sense that it either sends data from the Xtensa processor to an external device, or receives data from an external device. The TIE lookup construct can send a request or address out to an external device and receive a response or data back from the external device as part of the same TIE instruction.

The TIE lookup construct is ideally suited for connecting an external lookup table to the Xtensa processor. Lookup tables are typically implemented using a ROM, a CAM, or even synthesized logic. The following example illustrates the use of this construct:

```
lookup TBL {10, Estage} {44, Mstage}
state DATA 44 add_read_write

operation myROM {in AR addr} {out DATA, out TBL_Out, in TBL_In} {
     assign TBL_Out = addr[9:0];
     assign DATA = TBL_In;
}
```

The myROM instruction of the preceding example uses the lookup construct to read data from an external table. It generates a 10-bit address, which is derived from an AR register file operand, and this address is driven out of the Xtensa processor in the E stage of the processor pipeline. The 44-bit return data is expected back from the external device in the M stage of the processor pipeline, and is assigned to the state DATA. The declaration of the lookup named TBL as in the previous example creates the following new interfaces to the Xtensa processor:

```
output TIE_TBL_Req;        // One bit request signal
output [9:0] TIE_TBL_Out;  // Output address to external device
input [43:0] TIE_TBL_In;   // Input data from external device
```

The one bit request signal, TIE_TBL_Req, is asserted by the Xtensa processor when it puts out a valid address on TIE_TBL_Out. Thus the request signal can be used to enable the external device, which could be in a low power mode at other times. Note that the request signal is not assigned to in the instruction semantic; it is automatically asserted for one clock cycle by the Xtensa processor when a valid lookup request is presented to the external device.

The lookup interface can also be used to interface to a variety of different external devices, other than lookup tables. For example, it can be used to interface to some legacy RTL block that performs a certain computation. Assume that you have an RTL block that performs a certain computation on two 32-bit input values to generate a 30-bit result. Furthermore, the block takes 3 clock cycles to perform this computation. You want to interface this block to the Xtensa processor such that the values from two AR register file operands can provide the two inputs to this computation, and the result of the computation is put into a third AR register file operand. This can be achieved using the lookup construct as illustrated in the following example:

```
lookup RTL {64, 1} {30, 4} rdy

operation myCOMP {out AR result, in AR in0, in AR in1}
                 {out RTL_Out, in RTL_In}
{
     assign RTL_Out = {in0, in1};
     assign result = {2'b0, RTL_In};
}
```

In this example, the 64-bit address output is actually the concatenation of the two input operands for the computation. This value is sent to the external logic in stage 1, or in the E stage of the processor pipeline. Because the external logic takes 3 cycles to compute the result, the result is shown to come back in stage 4.

The lookup construct also allows the external logic to stall the Xtensa processor if it is not ready to accept the request. This could happen if the external RTL module of the previous example cannot accept a second request while processing the first request. Another scenario would be that the external device is a single port ROM that is shared between two Xtensa processors, in which case there needs to be arbitration for access to the ROM. In either case, the external device conveys its status (to accept or not accept a lookup request) to the Xtensa processor using the optional `rdy` interface. In the previous example, the lookup named `RTL` is declared with the optional keyword `rdy`. This creates an additional 1-bit input interface to the Xtensa processor named `TIE_RTL_Rdy`. The external logic will deassert this signal when it is not ready to accept a lookup request from the Xtensa processor. The Xtensa processor will stall when a lookup instruction is to be executed, and the `Rdy` signal is deasserted. It will keep the `Req` signal asserted, to indicate to the external device that it would like access to it.

Note that the external device must accept a lookup request if it has asserted the `Rdy` signal and the Xtensa processor asserts the `Req` signal. The lookup request cannot be stalled by deasserting the `Rdy` signal in response to the `Req` signal.

The lookup construct and its features are described in more detail in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. This includes a listing of a few restrictions regarding the pipeline stages in which the lookup output can be written and input can be sampled, conditions under which the `rdy` interface can be used, and so forth. For timing diagrams and related details corresponding to the lookup construct, refer to the *Xtensa LX Microprocessor Data Book*.

6. TIE Design Considerations

There are several trade-offs made during any design process, including optimizing your application with TIE instructions. This chapter describes some of these trade-offs. The color conversion algorithm example that is introduced and optimized in this and following chapters is described in Appendix A.

6.1 Algorithm Implementation Considerations

It is important to understand the distinction between algorithms and the implementation of algorithms. Examples of algorithms include:

- Fast Fourier Transform (FFT)
- Finite Impulse Response (FIR) Filter
- Data Encryption Standard (DES)

The specification of an algorithm is not its implementation. For each algorithm there are several possible implementations. Different implementations are suitable for different performance requirements (for example, the implementation of a FIR filter in the C programming language, Xtensa assembly instructions, or in hardware RTL). Therefore, the selection of an implementation can influence the maximum performance that an Xtensa processor can obtain. This section concentrates on algorithm implementations for a programmable processor.

The first step is to select a suitable implementation candidate, then implement an algorithm with TIE instructions. There are several issues to consider when selecting an implementation candidate, which are discussed in the following sections.

6.1.1 Optimized Versus Unoptimized Implementation

Some algorithm implementations are optimized for a certain processor. These optimizations result in a low number of cycles, but they are based on the features of that processor. Using code optimized for a specific processor as a basis for optimization with TIE can lead to worse results than starting from a clean, less optimized implementation.

Consider an algorithm that multiplies two 8-bit arrays. If the algorithms have been optimized for a processor that does not have multipliers, the algorithm might have been written using a series of lookup tables instead of multipliers. While perhaps more efficient for the original processor, the implementation obscures the algorithm and makes it

harder to optimize with TIE. Starting with a straightforward implementation, it is immediately apparent that adding SIMD (single-instruction/multiple-data) 8-bit multipliers in TIE can speed up the implementation.

There are also some DES implementations that make good examples. However, some of these implementations are highly optimized for an x86 architecture, which makes it difficult to understand the DES algorithm from this implementation. Using this description as the basis of TIE optimization leads to sub-optimal results because the x86 architecture does not contain the basic operators that the DES standard uses.

Therefore, to achieve the best results use a simple and direct implementation that contains all high level operators the algorithm uses.

6.1.2 Data Versus Computation Requirements

Algorithm implementations can also differ in the number of memory accesses and computations they require. In general, reducing the amount of memory accesses improves the maximum performance available using TIE instructions.

The FFT algorithm has two commonly used implementations, Radix-2 and Radix-4. Radix-2 uses more memory transfers but less temporary storage than Radix-4. TIE instructions provide the temporary storage, therefore requiring less memory transfers and providing a higher maximum performance for Radix-4.

6.1.3 Data Layout Considerations

Data layout is another important differentiation between algorithm implementations. Each implementation processes data, but two implementations that require the same number of data values might not produce the same optimized result.

It is important to look at the relationship between data values, such as the spacing between two loads (consecutive or non-consecutive) and the reuse of data.

The base Xtensa processor can load/store up to 64 bytes of data with one instruction. If two load/store units are used, then 128 bytes of data can be loaded with one instruction. This means that algorithm implementations that use consecutive values allow for higher performance because one load can load multiple data values, enabling SIMD optimization, for example.

Because the color conversion example in Appendix A reads and writes all data consecutively, SIMD is an option. The color conversion example also provides an example of data reuse in the reuse of the Cb and Cr values. Four Luminance values (Y) use only one Cb and Cr pair, as described in Appendix A. This is important because all multiplica-

tions are on the Cb and Cr values, and not on the Y values. The number of Cb and Cr values therefore control the number of multiplications, if we take advantage of the data reuse.

Another aspect of data layout is the placement of data in different memories and the resulting data throughput, which is discussed in the next section.

6.2 Data Throughput Considerations

It is important to consider where the data is coming from, and where it is going when algorithms are processing data. This section focuses on how to load the input data and output the results instead of the operations themselves. This section also discusses the different options that Xtensa provides for transferring data using TIE instructions.

Following are the mechanisms to transfer data to and from the core:

- Core load/store instructions, which can transfer up to 32 bits per cycle to and from local memory.
- TIE load/store instructions, which can transfer up to 512 bits per cycle to and from local memory.
- Inbound PIF, which can move up to 128 bits every 1 to 4 cycles[1] to and from the core local memories.
- For LX cores Two load store units, which doubles the load/store capacity of TIE load/store instructions.
- For LX cores TIE Queues/Ports, which allow high bandwidth for unaddressed data.

The following sections explain the trade-offs in using these options.

6.2.1 External Memory Versus Local Memory

The Xtensa processor contains general load/store instructions, which are capable of transferring 8, 16, or 32 bits to or from the core per instruction. Creating TIE load/store instructions extends this to 64, 128, 256, and 512 bits per TIE instruction.

The data must be in local memory or cache to achieve one load/store per cycle. Load/stores to or from external memory cause a replay of the load/store instruction and a minimum delay of five cycles for uncached and seven cycles for a cache miss.

1. The bandwidth number mentioned is for one inbound transaction. It is not possible to have back-to-back inbound transactions because of re-arbitration on the PIF interface.

For LX cores To add more bandwidth to local memory, extend the Xtensa LX processor with a second load/store unit. Note that enabling this option can complicate the system design because both load/stores can go to the same memory in the same cycle. Also, using two load/stores per cycle requires the use of FLIX instructions. Using two load/store units and local memory provides the theoretical memory bandwidth of two 512-bit load/stores per cycle. At 350MHz, this results in 44.8GB/s of load/store bandwidth.

6.2.2 Using Inbound PIF to Improve Data Throughput

As previously mentioned, the location of data is very important for maximum data throughput. The maximum bandwidth can be one or two load/stores per cycle if the data is in local memory. For some systems this requirement cannot be met when the data the processor must process is coming from an external processor or device, or when the data structures are too large for local memory.

The Xtensa inbound PIF feature transfers data to and from the core's local memories without using load/store instructions. Instead, it allows an external agent to the Xtensa processor to move data in and out of the core. Inbound PIF uses the local memory interface to move data in and out of local memory when the processor is not accessing that memory. One approach to maximizing inbound PIF transaction bandwidth is to configure the Xtensa processor with two local data memories and set the inbound PIF transfers to have the highest priority. Bandwidth is maximized when the load/store instructions read and write from and to one memory, while the other memory is used for Inbound PIF.

For LX cores Another approach to achieving maximum inbound data throughput is to configure the processor with two load/store units. The first load/store unit handles the load/store instructions while the second load/store unit is dedicated for inbound PIF. In this case, inbound PIF only uses the second load/store unit for its transactions.

Using inbound PIF requires the design of a device external to the Xtensa processor, which can transfer data to and from the core using the PIF protocol as described in the *Xtensa LX Microprocessor Data Book*.

6.2.3 Transferring Data with TIE Queues, Lookups and Ports For LX cores

TIE queues, lookups and ports create new primary input and output interfaces of the Xtensa processor, and provide a method to get data in and out of the core. TIE ports allow for data communication in and out of the core where the data changes very slowly compared to instruction execution. An example of using TIE ports is to read/write status information to and from the core. A TIE queue is a communication channel that runs synchronous with the processor. TIE lookups can be used to interface to external lookup tables, or to other computational units implemented in RTL.

TIE queues allow TIE instructions to access external data. A TIE queue does not implement a queue; instead it provides a synchronous access to the external data and the means to stall the processor if no data is available or no data can be written.

An Xtensa processor configuration can access up to 1024 queues/lookups/ports. Each of these can be from 1 to 1024 bits wide. Therefore, a staggering amount of data can be made available thorough the use of these interfaces.

TIE queues can be used to interface FIFOs to the core and to provide inter-processor communication. TIE queues can also provide a memory-like transfer (for example, a rotational buffer that can contain multiplication coefficients for use in a FIR filter). It is important to note that TIE queues do not provide the same features as a local memory, such as debug support and context switching.

6.3 TIE Optimization Strategies

After the implementation to optimize has been selected and investigated, the next step is to decide on the high-level optimization strategy and the types of instructions to add to the Xtensa processor.

6.3.1 General Versus Custom TIE Instructions

One decision for a high-level optimization strategy is whether to create general-purpose TIE instructions or more dedicated (custom) instructions. This decision is dependent on the target performance you want to achieve. General instructions are easier to reuse in other parts of your application and can therefore simplify your implementation. Custom instructions can achieve higher performance using more hardware.

If the performance target is very aggressive due to the number of operations that have to be performed per TIE instruction, or due to memory limitations, then the TIE instructions need to be more customized.

The color conversion that implements the matrix multiply shown in Figure 6–15 is an example of this.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 91881 \\ 1 & -22554 & -46802 \\ 1 & 116130 & 0 \end{bmatrix} x \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

Figure 6–15. Color Conversion Example Matrix Multiply

The profile for this application shows that the function using the most cycles was an integer multiplier.

To optimize the integer multiply, you can add a TIE instruction as a multiply between a register with a table value, as a multiply between a register and an immediate, or as a multiply between two registers.

Putting the multiplier coefficients in a TIE table is a valid implementation for the color conversion, but makes it hard to reuse this multiplier for multiplications that use different multiplication values.

Using an immediate for the multiplier value makes the instruction a lot more general. But if you look at the constant values above, an 18-bit immediate value is needed. However, using 18 instruction bits to hold the immediate is not practical in most cases.

Instead, make this TIE multiply instruction more general by using a register to hold the coefficient. The coefficient could reside in a core register, although putting the coefficients in state or a custom coefficient register file is preferable. A custom state/register file is better because the coefficients only need to be loaded once for the whole algorithm.

In this color conversion example, four multiplication values are needed, which would take four core registers. If there are enough core registers available, then this is not a problem. However, if there are not enough registers, extra instructions to reload the multiplication values are needed. To prevent this problem, use TIE state or a custom TIE register file for the multiplication values.

It is usually better to generate more general-purpose TIE instructions, and customize them later if necessary, because general TIE instructions can be easier to write and are easier to debug. Also, the chance of reusing a general TIE instruction is much higher.

6.3.2 Using TIE State Versus TIE Register Files

A problem very closely related to the discussion of general versus custom TIE instructions is whether to use TIE state or TIE register files for your temporary storage needs. This is a complex question and there is not a simple answer because both have their strengths and weaknesses.

Because the XCC compiler will perform register set allocation for you, TIE register files provide the best software support. This makes writing software that uses TIE register files easy. Using regular files is more likely to allow you to reuse your TIE operations because the choice of which register to use in a given operation is not fixed in hardware. However, using register file operands require the use of instruction bits.

TIE states allow you to write multiple states in one instruction; yet writing too many states can cause place and route problems. Another feature is that TIE states do not use any instruction bits. TIE states can easily be used when their use is hidden by instructions. An example of this is using a TIE state as an accumulator in the color conversion example. As long as only one accumulator is required, the programming model is acceptable.

Problems start to occur when you try to model a register file with TIE states. The state might require significantly more hardware than the register file. Also, the register allocation needs to be done by hand, thus creating the possibility that advanced compiler optimizations might not be feasible. It is important to note that modeling a register file with TIE states can cause timing problems because the selection of which state to use is performed in the execute stage while a TIE register file performs this selection in the decode stage.

As a general guideline, use register files if you need multiple and general storage. If you need single task, dedicated storage, use a state.

6.3.3 Optimization Techniques: Fusion, SIMD, FLIX

The next strategy decision is to decide which optimization techniques to use: Fusion, SIMD (single-instruction/multiple-data) and/or FLIX. Think of all possible performance points as a three-dimensional space with Fusion, SIMD and FLIX as its axis, as shown in Figure 6–16.

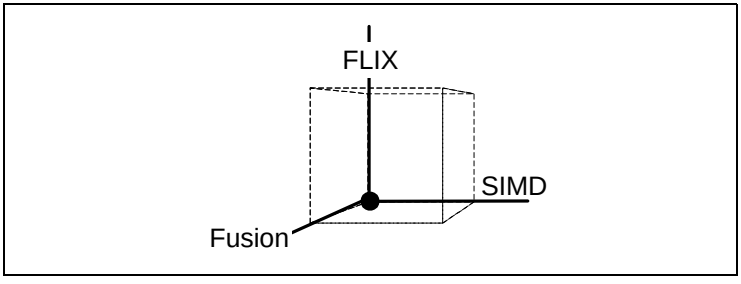

Figure 6–16. Possible Optimization Techniques

This section explains how to optimize the color conversion example using each of these three techniques and shows the differences between them.

First, examine the C code for the color conversion example. The following operations are performed on each YCbCr triplet:

```
void ConvertYCbCrToRGB_Orig (unsigned char Yin,
                             unsigned char Cbin,
                             unsigned char Crin,
                             unsigned char * Rout,
```

```
                                unsigned char * Gout,
                                unsigned char * Bout)
{
  int Y, Cr, Cb;
  int R, G, B;

  /* U and V must be adjusted to lie in the range -128 .. 127 */
  Cb = (int) Cbin - 128;
  Cr = (int) Crin - 128;

  /* Convert YUV to RGB */
  Y = (int)Yin<<SCALEBITS;
  R = Y              + c_RCr * Cr + ONE_HALF;
  G = Y - c_GCb * Cb - c_GCr * Cr + ONE_HALF;
  B = Y + c_BCb * Cb              + ONE_HALF;

  /* Scale down the values and clamp values to the range 0 .. 255 */
  if (R < (1<<SCALEBITS)) *Rout = 0; else if (R > (0xff<<SCALEBITS))
*Rout = 255; else *Rout = R >> SCALEBITS;
  if (G < (1<<SCALEBITS)) *Gout = 0; else if (G > (0xff<<SCALEBITS))
*Gout = 255; else *Gout = G >> SCALEBITS;
  if (B < (1<<SCALEBITS)) *Bout = 0; else if (B > (0xff<<SCALEBITS))
*Bout = 255; else *Bout = B >> SCALEBITS;


} /* ConvertYCbCrToRGB */
```

The C code contains an addition of a half, which is not shown in the matrix multiply of the equation shown in Figure 6–15. This addition is necessary to lessen errors when reducing the integer multiplier results back to the range of [0:255].

The following sections explain how to optimize this example using first fusion, then SIMD.

Optimizing the Color Conversion Example with Fusion

Fusion means merging multiple basic operations into one instruction. A fusion example is a multiply accumulate instruction because it combines a multiply and an addition. Another example is a load with address update instruction because it combines a load with the addition of the address with the update value.

Fusion is a very effective and important technique for optimizing your algorithm and can always be used.

The preceding examples show how fusion can accelerate the color conversion example. If the optimization goal is to get maximum performance at all costs using fusion, then use one fusion instruction that contains all the C code of the `ConvertYCbCrToRGB_Orig` function shown above. This super-fusion TIE instruction will have to become a multi-cycle instruction because of the number of required operations (see Chapter 7).

The inputs for this color conversion instruction are three states, which contain four Y, Cb, and Cr components. The outputs are the corresponding four R, G and B values, as shown in Figure 6–17. Note that this super-fusion instruction only converts one YCbCr value into one RGB value. The diagram in Figure 6–17 does not contain the selection of which of the four YCbCr values to use as input, nor the combination of the current calculated RGB value with the R, G, and B state registers.

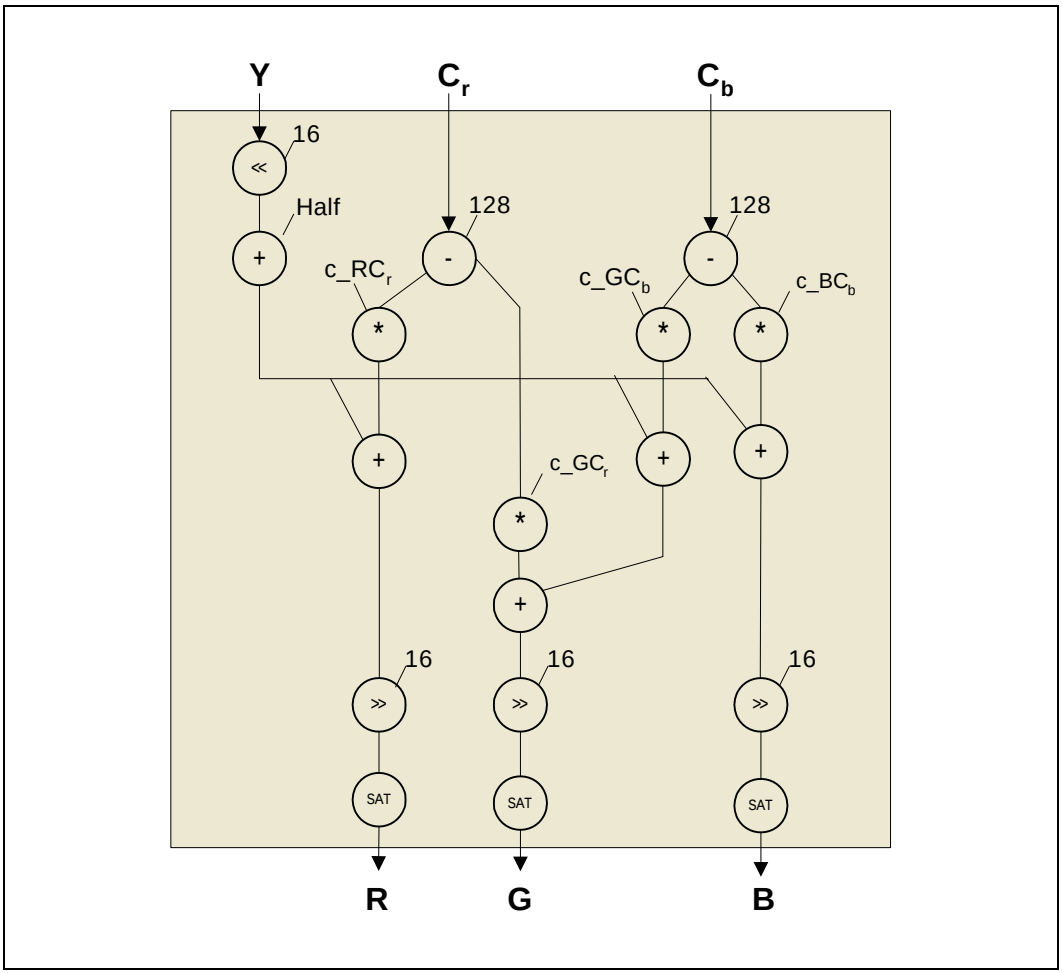

Figure 6–17. Results of Fusion: the Color Conversion TIE Instruction

The example super-fusion color conversion TIE instruction reduces the cycle count for 8192 color conversions by a factor of 9, from 262,830 to 26,953 cycles.

Fusion is a powerful technique to achieve performance improvements; however, it generates customized instructions. If too many operations are merged together, such as in the preceding color conversion example, the possibility of reusing the new TIE instructions for other algorithms becomes small.

Optimizing the Color Conversion Example with SIMD

Another way to implement the `ConvertYCbCrToRGB_Orig` function is to perform operations on multiple-data elements (SIMD). This optimization technique can be used in the color conversion example because the same color conversion function is applied on sequential data elements.

It is relatively easy to generate general instructions using SIMD by using register files for the instruction input and outputs. Now, the question is on how many data values will these instructions operate? Again the goal is to provide high performance, as in the fusion case.

In the example, color conversion performs multiplies on only the Cb and Cr values. These arrays are a quarter of the size of the Y array. Therefore, by focusing on the Cb and Cr arrays, the total number of loads can be reduced.

For this example, use eight parallel multipliers and a main register file of 128-bits wide. The register file can then hold 16 Y values and eight Cb/Cr values. The next task is to define the necessary SIMD instructions.

The following instructions are created to accelerate 16 color conversions:

- Load 64 bits (8 Cb or Cr value) into a 128-bit register. The Cb/Cr values are zero extended
- Subtract 128 from Cb/Cr in a register
- Multiply accumulate using round state
- Multiply accumulate
- Saturate and add the instruction, which adds the results of the multiplications to Y and saturates the result

For the multiplications, use a coefficient register file of 18-bits wide and add an accumulator register file of two entries.

Using SIMD TIE instructions to perform color conversion for 16 YCbCr results in the following implementation for the `ConvertYCbCrToRGB` function:

```
load64_extend(Cb, AddrCb, 8);
```

```
Cb = vec_SUB_SCALAR(Cb, 128);

load64_extend(Cr, AddrCr, 8);
Cr = vec_SUB_SCALAR(Cr, 128);

Y= *(++pY);

Res0 = vec_MAC_R(Cr, c_RCr);
R          = vec_ADDSAT(Y, Res0);
*(++pR)    = R;

Res1 = vec_MAC_R(Cb, c_GCb);
vec_MAC(Res1, Cr, c_GCr);
G          = vec_ADDSAT(Y, Res1);
*(++pG)    = G;

Res2       = vec_MAC_R(Cb, c_BCb);
B          = vec_ADDSAT(Y, Res2);
*(++pB)          = B;
```

The full source code and the complete TIE implementation are shown in Appendix B. The result of applying SIMD is that the color conversion example now runs in 8192 cycles.

These TIE instructions make a good basis for optimizing more algorithms using SIMD. Some instructions, such as the `load64_extend` and `vec_ADDSAT`, will probably remain specific to the color conversion example.

Optimizing the Color Conversion Example using FLIX For LX cores

FLIX allows multiple operations to execute in parallel where each operation is an Xtensa core or TIE operation. Section 5.9 describes FLIX instruction basics; Section 7.5 explains how to optimize the color conversion example using FLIX. A much higher improvement can be obtained by using fusion and/or SIMD optimizations along with FLIX.

6.3.4 Fusion Versus FLIX Instructions For LX cores

Fusion combines multiple computations into one TIE operation, whereas FLIX allows for the execution of multiple operations in one instruction. These two techniques are similar in that both allow for multiple computations to be performed in one instruction, but there are also important differences.

Fusion instructions tend to become specific to a certain application when more computations are combined. Conversely, FLIX stays general because each slot can contain multiple operations and the compiler tries to schedule these operations into a FLIX word. Also, frequently you need multiple flavors of similar fusions with slightly different combi-

nations of the underlying computation. Each flavor needs to be described as a different operation, resulting in many operations. With FLIX, you need an operation to describe each underlying computation, and though proper bundling and ordering of these operations, you can perform the desired computation.

Fusion has some other advantages. It typically takes fewer register ports, so it is frequently cheaper. You do not need to pipeline as much for fusion, which leads to cheaper hardware and better performance for small loops.

Both techniques can provide a similar performance with a slight advantage for FLIX because it can use multiple load/store units. Therefore, the main reason to use FLIX is for general instructions; the main reason for using fusion is because it requires less area and provides better code density.

7. Using TIE Instructions to Optimize Software

This chapter consists of two parts: The first part describes how to use TIE instructions in software and how to debug your software containing TIE instructions. The second part of this chapter discusses ways to improve your program's performance. This chapter introduces the following methods to optimize software performance:

- Controlling pipeline hazards
- Removing address increment instructions
- Using FLIX instructions

7.1 Using TIE Instructions in Software

The TIE Compiler (tc) generates a header file when it compiles the TIE file. This header file must be included in every C/C++ file that uses a TIE instruction. For example, for a TIE file named, `Test.tie`, add the following line to each C/C++ file:

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

Note that the header file name follows the same case format as the TIE file name.

After including the TIE header file, C intrinsics are available for all TIE instructions. The TIE instruction intrinsic has the same name as the corresponding `operation` construct.

Only the register and immediate operands of the operation are relevant for the intrinsics. The intrinsic arguments use the following rules:

- All arguments are in the same order as the argument-list of the TIE operation declaration.
- If the TIE operation has only one argument of type `out` in the argument-list (and no `inout` arguments), it becomes the return value of the intrinsic.
- If there is more than one `out` argument or any `inout` arguments, they are implicitly passed by reference,[1] and the intrinsic has no return value.
- Any argument that is used as a load address needs to be a pointer to the ctype of the target register file or a void pointer for loads into state.[2]
- Any other argument needs to be of the corresponding ctype.

Following are example TIE instructions and their use as intrinsics in C code:

1. . Passing by reference in C programs is a Tensilica extension to C.

2. To cause the function prototype to use a pointer in the original TIE operation construct, use an asterisk (*) before the argument that contains the address. To make the function prototype, use a pointer in the original TIE operation construct.

```
TIE:
     operation MyTIE {in AR A}{inout COUNT}{ assign COUNT=COUNT+A;}
C code:
     int value=7;
     MyTIE(value);

TIE:
     operation MyTIE{inout AR COUNT, in AR A}{}{assign COUNT=COUNT+A;}
C code:
     int count=0;
     int value=7;
     MyTIE(count, value);


TIE:
     operation MyTIE{out AR COUNT, in AR A}{}{assign COUNT=A;}
C code:
     int count;
     int value=7;
     count=MyTIE(value);

TIE:
     operation MyTIE{out AR COUNT, out BR flag, in AR A}{}
     {
          assign COUNT=A;
          assign flag= (A==1);
     }
C code:
     int count=0;
     int value=7;
     xtbool flag;
     MyTIE(count, flag, value);
```

If multiple types use the same TIE register file, a custom proto allows you to specify which types are used by which intrinsic. The *Tensilica Instruction Extension (TIE) Language Reference Manual* contains more examples.

In addition to calling the C intrinsics, TIE operations can also be represented as C/C++ operator symbols (such as "+", "-"). TIE designers need to write an `operator` construct to connect C/C++ operator symbols to protos they are translated to. Details of the `operator` construct are in the Operator chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. The following example illustrates how to overload the operators.

```
TIE code:
     regfile XR 32 16 xr
     operation XR_ADD {out XR a, in XR b, in XR c} {} {
```

```
            assign a = b + c;
        }
        operator "+" XR_ADD
C code:
        XR a, b, c;
        /* assign values to b and c */
        a = b + c;
```

Use of the `operator` construct in your TIE design is one way to specify operator overloading. It is also possible to specify operator overloading by writing C functions with stylized names that are recognized by the XCC compiler. This approach may be particularly useful when the overloaded operation cannot be easily described using a TIE proto. It is also useful for adding operator overloading after the TIE design is finalized. See the *Xtensa C and C++ Compiler User's Guide* for details. In the above example, the following C function can be used to overload the "+" operator.

```
inline XR __xt_operator_PLUS(XR a, XR b) {
  return XR_ADD(a, b);
}
```

7.2 Debugging TIE Instructions

After specifying your TIE instructions and incorporating them into your software, you can simulate and debug your TIE instructions. Debugging TIE instructions is possible using either Xtensa Xplorer or xt-gdb. When debugging your application code using Xplorer, the Debug perspective provides a TIE Wires view as shown in Figure 7–18. This view shows all your TIE instructions, and by double clicking on any instruction, you can see the value of all the operands of that instruction.

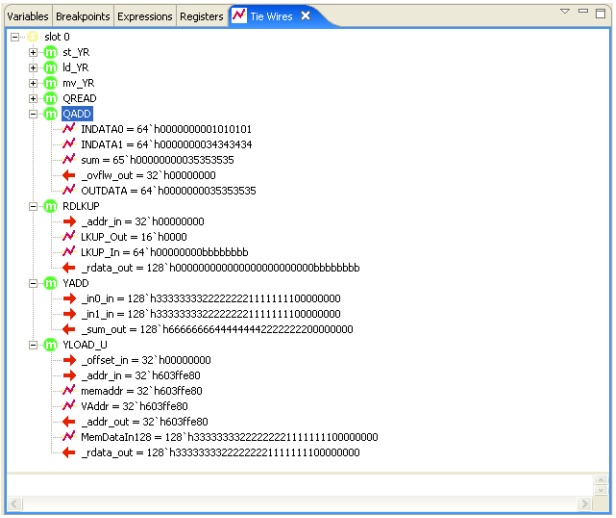

Figure 7–18. Xtensa Xplorer TIE Wires View

To view the instructions in the Tie Wires view, you must compile your TIE code with debugging enabled. This is done by selecting the appropriate option in Xtensa Xplorer's TIE Compile Options dialog box before compiling the TDK. When using the command line interface, you must use the `-libdebug` flag of the TIE compiler when compiling the TDK. Note that the TIE wires view is only available when performing a cycle-accurate ISS simulation; it is not available when using the TurboXim engine.

The following values corresponding to each TIE instruction are visible in the Tie Wires view of Xtensa Xplorer:

- TIE Instruction input/outputs
  - Register operands
  - States
  - Interfaces
- Internal wire declarations

All register or state input/output operands of the instruction are prefixed with an underscore character ("_") and followed by the direction `_in` or `_out`. For inout operands, you will see two entries; one with the `_in` suffix (the input value of that operand), and one with the `_out` suffix (the output value of that operand). You can also see each wire that you have defined within your TIE operation or semantic construct.

The TIE language allows you to specify an `operation` (or `reference`) description of an instruction and a `semantic` description of the same instruction. By default the ISS simulates the semantic description of the instruction, thus the TIE Wires view will display

the wires as defined in the semantic description. You can simulate the operation description of the instructions by specifying the flag `--reference` in the Additional Arguments section of the Simulator tab in the Run menu of Xplorer. The same flag can also be used with the command line interface of the ISS. When simulating with the `--reference` flag, the TIE Wires view displays the wires as defined in the operation description of the instructions.

When debugging TIE load/store instructions, the following interface signals are visible in the TIE Wires view:

- `VAddr`: Virtual address of the load or store
- `MemDataInX`: For load instructions, this is the load data coming from memory. `X` is the load size in bits.
- `LoadByteDisable`: For load instructions, this is the interface that controls the byte enables. Displayed only if the instruction uses this interface.
- `MemDataOutX`: For store instructions, this is the data going to memory. `X` is the store size in bits.
- `StoreByteDisable`: For store instructions, this is the interface that controls the byte enables. Displayed only if the instruction uses this interface.

For LX cores The Xtensa LX processor can be extended with TIE input ports, exported state, queues and lookups. Special debug signals are available to support debugging with these TIE features.

Using a TIE input port with `import_wire` adds the following debug signal under the TIE wires view:

- `<import wire name>`

Using a TIE exported state does not add a new debug signal under the TIE wires view. It uses the normal state debug signal:

- `_<state name>_out`

Using a TIE queue adds the following debug signal under the TIE wires view:

- `<queue name>`: Input or output data value from/to the queue
- `<queue name>_NOTRDY`: Input interface that indicates whether the queue is ready to accept the next pop/push without stalling. Displayed only if the instruction uses this interface.
- `<queue name>_KILL`: Output interface indicating that the queue access is to be killed. Displayed only if the instruction uses this interface.

Using a TIE lookup adds the following debug signals under the TIE wires view:

- `<lookup name>_Out`: Output address or value to the external device
- `<lookup name>_In`: Input data or response from the external device

7.2.1 Simulating and Debugging TIE Ports, Queues and Lookups

Tensilica recommends that you use the Xtensa SystemC (XTSC) or Xtensa Modeling Protocol (XTMP) environment for debugging and profiling application code that uses TIE ports, queues or lookups. In the XTSC/XTMP environment you can provide models of the external devices that connect to these interfaces and perform a realistic simulation using these interfaces. Please refer to the *Xtensa SystemC (XTSC) User's Guide* for more information on this subject.

It is also possible to perform a quick and simple simulation of these interfaces using the ISS with a file that contains values for all input ports, queues and lookups. In this example the following TIE file is simulated and debugged:

```
import_wire ready 1
queue Data0 40 in
queue Data1 40 in
queue Data2 40 out
state done 1 1'b0 add_read_write export

operation QueueAdd{}{in ready, in Data0, in Data1, out Data2,
                     out done}{
     assign Data2 = Data0 + Data1;
     assign done_kill = ~ready;
     assign done = ready;
}
```

The C code that is used to debug the TIE file shown above is:

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

int main (){
     int i;

     Wdone(0);

     QueueAdd();
     Wdone(0);
     QueueAdd();
     Wdone(0);
     QueueAdd();
     Wdone(0);
     QueueAdd();
     Wdone(0);

}
```

A control file is needed to simulate and debug this TIE file on the ISS. The control file tells the ISS what the input values are for the input ports and queues. It is also possible to let the ISS write the output values to a file. Following is an example of a control file.

```
ready
0
1
Data0
1
2
3
4
5
Data0_Empty
0
Data1
10
20
30
40
50
Data1_Empty
0
Data2_Full
0
Data2 testqueue.out
done testqueue2.out
```

TIE port or queue inputs can be specified in a file or by listing the values below the name using only one value per line. The ISS cycles through all the inputs. When it has used the last value it provides the first value the next time. Note that the values written to the TIE output Queue `Data2` are written to a `testqueue.out` file. The values for exported state `done` are written to the output file `testqueue2.out`.

To run the ISS for this example it is necessary to add the option `--tieports=<`*`filename`*`>`. This option tells the ISS to read `<`*`filename`*`>` as the control file for all TIE ports and queues. The option can be added when running xt-run on the command line or by adding it to the simulator tab of the run launch. This is shown in Figure 7–19.

If the lookup device can be modeled as a ROM or RAM, ISS can also be used to simulate them. Refer to the *Instruction Set Simulator (ISS) User's Guide* for details.

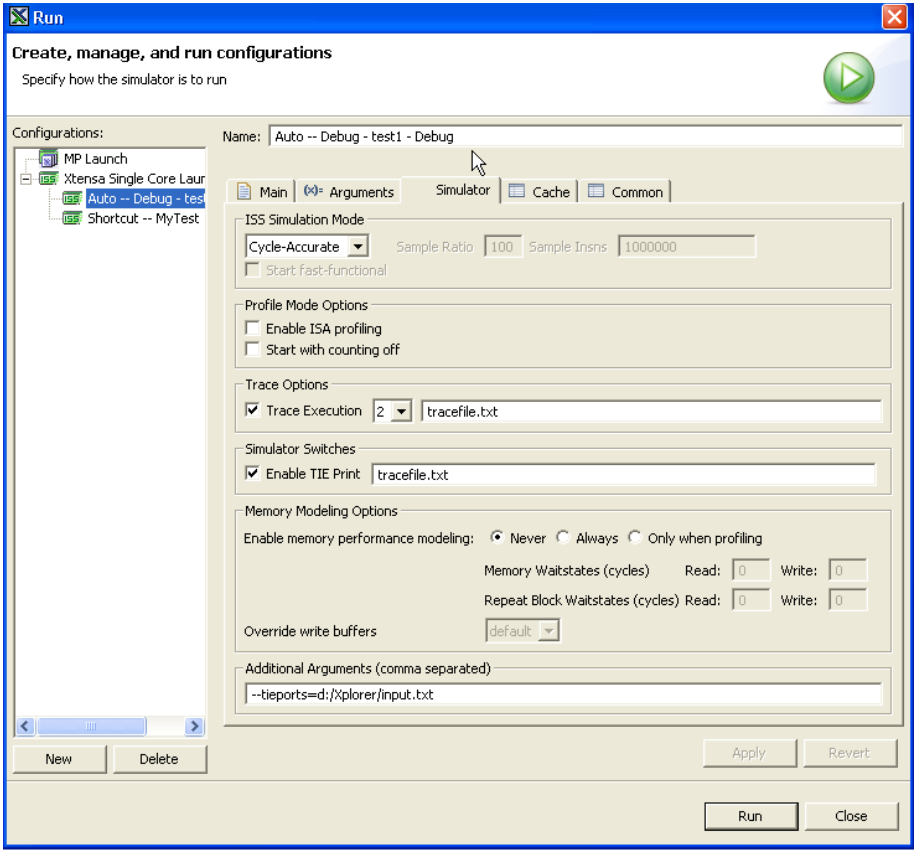

Figure 7–19. Specifying an Input File for an ISS Run Using TIE Ports/Queues

7.2.2 Using TIEprint for Debugging

The previous section described how you can view the values of various TIE variables when debugging in the Xtensa Xplorer debug perspective. In some situations, it may be easier to debug using C (programming language) style *printf* statements in your TIE code. Such print statements can be embedded in your TIE code using the `TIEprint` construct. The output of these print statements is directed to the trace log file when the corresponding instruction is executed during an ISS simulation of your application code. This debugging method is particularly useful when you want to run a long simulation and observe how the value of some variables change over the course of the simulation. To activate TIEprint, you must select the Enable TIE Print option in the Simulator tab of Xtensa Xplorer's Run Options menu. If running the simulation using the command line interface, you must use the `--tieprint` command line argument to the ISS.

The following TIE code illustrates the use of the TIEprint construct:

```
state COUNT 10 add_read_write
regfile XR 64 16 x

operation XADD {out XR res, in XR in0, in XR in1} {inout COUNT} {
     wire [63:0] tmp_res = in0 + in1;
     assign res = tmp_res;
     assign COUNT = COUNT + 1;
     TIEprint((COUNT[0] == 0), "XADD: Count = %d, res = %08x_%08x\n",
                    COUNT, tmp_res[63:32], tmp_res[31:0]);
}
```

The first element of the TIEprint construct is a Boolean qualifier, which controls whether the TIE print will be executed or not. If this qualifier evaluates to true, the TIEprint is executed, otherwise it is not. Thus in the previous example, the TIEprint is executed only when the `COUNT` value is an even number. This qualifier is optional, and if not included, the TIE print is always executed.

The rest of the TIEprint construct syntax is similar to the *printf* construct of the C programming language. It includes a formatting string followed by a list of arguments to be printed. The formatting string allows the use of format qualifiers such as `%d` to print a decimal number and `%08x` to print a hex number using a minimum of 8 digits with leading zeros.

One restriction of the TIEprint construct is that you cannot print output arguments of an instruction. To work around this restriction, the result of the addition in the `XADD` instruction is assigned to a temporary wire, and this wire is then assigned to the output operand of the instruction. You can now print the value of the temporary wire as is done in the TIEprint of the `XADD` instruction. Because of this restriction, when you print the value of an `inout` operand of an instruction, it always prints the input value. Thus the value of the state `COUNT` printed by the `XADD` instruction is always the input value, and never the incremented (output) value. Another restriction of the TIEprint construct is that the printed values can be a maximum of 32-bits wide. Variables that are wider than 32-bits have to be broken up into 32-bit segments before being printed, as is done with the `tmp_res` wire.

Consider executing the following piece of C code that uses the `XADD` instruction:

```
#include <xtensa/tie/tieprint.h>
main() {
     // Code to initialize input operands of XADD instruction
     XR *p_in0, *p_in1, in0, in1, res;
     int i;
     unsigned int in0_data[8] = {0x00000000, 0x00001111,
                                 0x00002222, 0x00003333,
                                 0x00004444, 0x00005555,
```

```
                                0x00006666, 0x00007777};

    unsigned int in1_data[8] = {0x88880000, 0x99990000,
                                0xaaaa0000, 0xbbbb0000,
                                0xcccc0000, 0xdddd0000,
                                0xeeee0000, 0xffff0000};

    p_in0 = (XR *)in0_data;
    p_in1 = (XR *)in1_data;

    // Initialize state COUNT to 0
    WUR_COUNT(0);

    // Loop to execute XADD instruction 4 times.
    for (i = 0; i < 4; i++) {
      in0 = *p_in0++;
      in1 = *p_in1++;
      res = XADD(in0, in1);
    }
}
```

The first half of the program is to initialize the 64-bit input operands of the XR register file as explained in Section 5.6.2 “Using TIE Register Files in C/C++” on page 46. The program then initializes the state COUNT to 0 and performs four XADD operations. When this program is executed, the ISS trace output will show:

```
XADD: Count = 0, res = 99991111_88880000
XADD: Count = 2, res = dddd5555_cccc4444
```

TIEprint is only activated for even values of COUNT, and there is no output for the odd values. Furthermore, the COUNT value that is printed is the input value, and not the incremented output value.

The TIEprint construct is only used for generating print statements during software simulations. It has no impact on the hardware that is generated for your TIE instructions. For the formal syntax definition of the TIEprint construct and additional examples of its use, refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

7.3 Optimizing Performance by Controlling Pipeline Hazards

Performance of any program using Xtensa core instructions or TIE instructions depends on the instruction ordering and inter-instruction dependencies. These inter-instruction dependencies may cause the delay of an instruction (for example, when the input register of an instruction has not been updated by the previous instruction). The processor pipeline causes this type of problem (called a pipeline hazard). The delays pipeline hazards cause are pipeline bubbles, which lower the performance of your code.

Pipeline bubbles can be caused by source interlocks (described above), load delays, and branch delays.

If all instructions were to execute in a single cycle, there would never be a pipeline hazard or pipeline bubbles caused by source interlocks, because it is multi-cycle instructions that cause source interlocks that lead to pipeline bubbles.

Branch delays cause pipeline bubbles because the decision to take a branch or not is performed in the E stage. By this time the next two instructions are already in the pipeline, assuming that the branch will not be taken. When the branch is evaluated and the branch is taken, then these two instructions must be flushed from the pipeline, which causes 2 to 4 cycles delay, known as a branch delay.

Figure 7–20 shows a basic example of a load delay pipeline hazard. Load data is available at the end of the Xtensa pipeline's M stage. An instruction that wants to use the load data at the beginning of the execute stage is delayed for one cycle in a 5-stage pipeline. This delay increases to two cycles for the 7-stage pipeline.

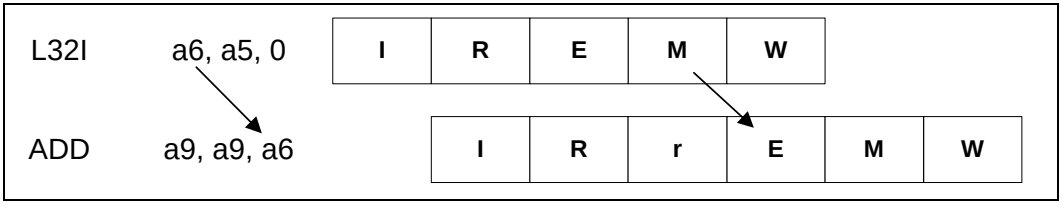

Figure 7–20. Example Pipeline Hazard

Multi-cycle TIE instructions are a generalization of the load delay slots. These instructions do not cause problems when there is no dependency between a multi-cycle TIE instruction and the next few instructions. However, if the next instruction uses the result of a multi-cycle instruction, then this next instruction is delayed for *N* cycles (where *N* is the number of execution cycles of the multi-cycle instruction, minus one). If the first instruction after the multi-cycle instruction is not dependent on the multi-cycle instruction, but the next one is, then it will be delayed for N-1 cycles.

TIE may create pipeline dependencies that can lower the performance gain of your software's performance. The performance is lowered when the compiler cannot fill the pipeline bubbles with useful instructions. This tends to happen more often when optimizing using TIE instructions. Optimization reduces the number of instructions and therefore the chance that the compiler can fill a pipeline bubble with a useful instruction.

7.3.1 Detecting Pipeline Hazards

This section discusses how to detect pipeline hazards and how to prevent them from causing pipeline bubbles, thus making the real TIE performance improvement visible. Consider the following TIE code that defines a load instruction and another that performs an addition.

```
state st_A     32   add_read_write
state st_Acc   32   add_read_write

immediate_range imm4 0 60 4

operation LoadA {inout AR *address, in imm4 offset}
                    {out st_A, out VAddr, in MemDataIn32} {
     wire [31:0]TmpAddr  = address + offset;
     assign VAddr        = TmpAddr;
     assign st_A         = MemDataIn32;
     assign address      = TmpAddr;
}

operation AddA {}{inout st_Acc, in st_A}{
     assign st_Acc  = st_Acc + st_A;
}
```

The following piece of C code adds ten numbers together using the previously defined TIE instructions. The loop contains a load delay pipeline hazard because the `LoadA` instruction loads data into state `st_A` and the `AddA` instruction adds state `st_A` to `st_Acc`.

```
     for(i=0;i<10;i++){
          LoadA(p, 4);
          AddA();
     }
```

After profiling the C code on the ISS, the **Pipeline View** in the **Benchmark Perspective** of Xtensa Xplorer can be used to view such pipeline hazards. This is illustrated in Figure 7–21, which shows that the `AddA` instruction is always held in the R-stage for an extra cycle because of this data hazard.

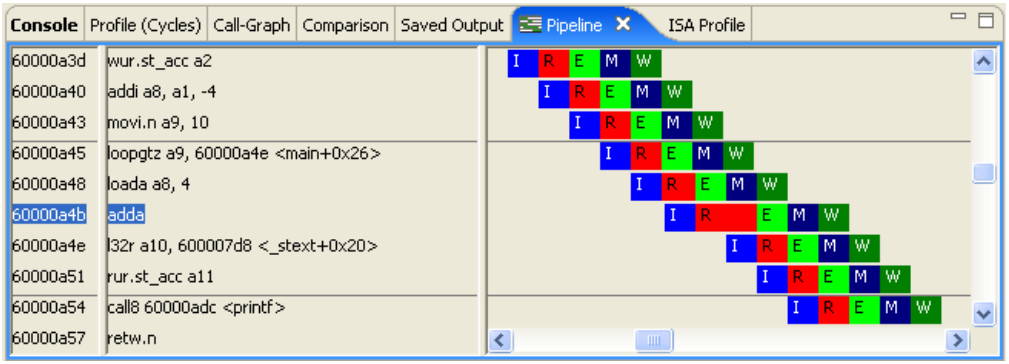

Figure 7–21. Pipeline View in Xtensa Xplorer

The pipeline view is very useful when you have identified a small section of your code (such as an inner loop) that you wish to analyze and optimize further. By viewing the profile of the disassembly, you can identify such sections of code that deserve further attention. For example, Figure 7–22 shows the assembly sequence for this program in the **Profile Disassembly** view of the **Benchmark Perspective** of Xtensa Xplorer. This view shows the assembly sequence of the instructions executed, their address (that is, program counter value) and the number of clock cycles spent executing each instruction.

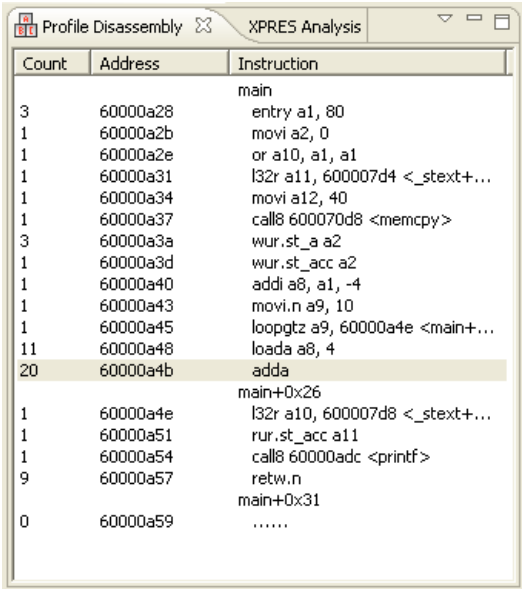

Figure 7–22. Xtensa Xplorer Profile Disassembly View

The cycle counts in Figure 7–22 show that some instructions use more cycles than others. For example, the two TIE instructions in the loop take the greatest number of cycles because they are executed 10 times each. Figure 7–22 also shows that the first instruction of the highlighted loop consumes 11 cycles, while the loop is executed 10 times. This is because the first execution of a loop return causes a one-cycle delay. In the **Profile Disassembly** view, this delay is added to the target of the return, which is the first instruction. The same thing occurs with branches, function calls, and returns. Even though it is the branch instruction that causes the branch delay, it is the target of a taken branch that is delayed. Figure 7–22 also shows that the AddA instruction consumes 20 cycles, which is almost double the number of cycles compared to the load instruction. This is because of the pipeline bubble illustrated in the Pipeline View of Figure 7–21.

The Profile view also allows you to check directly for branch delays and source interlocks by looking at their profile output. Xplorer will run the source interlock and branch delays profile, along with the execution cycle profile. These profiles show which functions and assembly lines incur penalties.

If you need more profiling information than the Profile View provides, use the ISS's trace capability. Full tracing shows which instructions need more than one cycle and shows why it needs more cycles. Because the generated trace file is large, Tensilica recommends using the trace method for small programs only. For information about tracing, see the *Xtensa Instruction Set Simulator (ISS) User's Guide*.

7.3.2 Removing Pipeline Hazards

The Xtensa C/C++ Compiler (XCC) usually prevents pipeline bubbles by trying to schedule independent instructions before the instruction that would be stalled by the hardware. For example, in the case of the load example, XCC could insert an instruction that is independent of the load after the load instruction. Thus, the instruction that uses the load does not need to stall.

Using a lot of TIE to optimize a piece of code tends to result in fewer overall instructions and strong dependencies between instructions. In the case of the load delay example, XCC will not be able to find an independent instruction because there are only two instructions inside the loop. Following are descriptions for two techniques that remove the dependency and the pipeline bubble.

Software Pipelining

XCC applies software pipelining when optimizing your program. However, there are cases when XCC cannot optimize your application any further. These cases can occur when states are being used or when the compiler runs into aliasing problems. See the Controlling Alias Analysis section of the *Xtensa C and C++ Compiler User's Guide* for more information.

For code executed inside a loop, the pipeline hazard can be removed by applying the software pipelining technique on TIE instructions. For the load delay slot example, the critical part of the code is a loop instruction containing a load into state st_A. This is followed by an add of state st_A with an accumulation state st_Acc. The load has a dependency to the add, which will cause a pipeline bubble. If a register file is used, the compiler can prevent this pipeline bubble by unrolling the loop once.

To break this dependency, you can change the TIE instructions by modifying the load instruction to move state A into another state B. Write State B at the end of the execute cycle. Then modify the add instruction to use this new state B. This is shown in the following modified TIE file for the load delay slot example.

```
state st_A     32   add_read_write
state st_B     32   add_read_write
state st_Acc   32   add_read_write

immediate_range imm4 0 60 4

operation LoadA_mvA2B {inout AR *address, in imm4 offset}
               {inout st_A, out st_B, out VAddr, in MemDataIn32}{
     wire [31:0]TmpAddr  = address + offset;
     assign VAddr        = TmpAddr;
     assign st_A         = MemDataIn32;
     assign st_B         = st_A;
     assign address      = TmpAddr;
}

operation AddB {}{inout st_Acc, in st_B}{
     assign st_Acc  = st_Acc + st_B;
}
```

The C code, which now adds 10 numbers, is shown below. Note that in this implementation 11 loads are performed and the last load data is never used. To prevent the extra load, execute the loop only nine times, and then add a new instruction after the loop that adds st_A to st_Acc.

```
     LoadA_mvA2B(p, 4);
     for(i=0;i<10;i++){
          LoadA_mvA2B(p, 4);
          AddB();
     }
```

AddB now uses state st_B, which is free of any dependency, thus no pipeline bubble will occur. The dependency that existed between two instructions is now separated by a loop iteration by introducing an intermediate storage element in state st_B. Note that the load data is not used until the next iteration of the loop. For more information about software pipelining, see Section 7.5.

Skewing the Pipeline by Specifying a Schedule

The pipeline bubble in the previous example can also be removed by skewing the pipeline. Assume that there is an instruction X that takes multiple cycles to generate its result (rX), which causes one or more pipeline bubbles for the next instruction (Y) that uses rX. Another way to solve the pipeline hazard caused by the source interlock is to generate a schedule for instruction Y to use rX in the pipeline stage in which X generates the result. Instruction Y executes after instruction X, therefore the value of rX can be forwarded from Instruction X to Instruction Y.

Now apply this technique to the load delay slot example. The pipeline hazard is caused because the `loada` result is not available until the end of the M stage. At this time the `adda` instruction should already be in the execute cycle. This is not possible; therefore the Xtensa pipeline delays the execution of the `adda` instruction.

Delaying the execution of the `adda` instruction to the M stage can prevent the pipeline bubble. There are no pipeline hazards because the result of the load is available to be used in the M stage of the `adda` instruction. To move the execution of the `adda` instruction to the M stage, add the following schedule:

```
schedule AddA_Schedule {AddA}{
     use st_A  2;
     use st_Acc2;
     def st_Acc2;
}
```

Take care when using this optimization technique because any instruction following, and depending on, the result of AddA must delay its execution, as well as any instruction depending on this instruction. Skewing the pipeline can cause a domino effect, which can lead to some pipeline bubbles. An example of this is in the following sequence in C:

```
LoadA(p, 4);
AddA();
RUR_st_Acc()
```

The read user register instruction, `RUR_st_Acc()`, uses the `st_Acc` state in the execute stage. A source interlock now exists between the AddA and the `RUR_st_ACC` instruction. This causes a one-cycle delay for a 5-stage pipeline (two cycles for a 7-stage pipeline).

7.4 Removing Address Increment Instructions

Critical loops tend to contain data load operations followed by some computation, and data stores. The color conversion example uses three data loads and three data stores for each loop iteration. Each of these load instructions requires an address increment, which requires six instructions out of 21.

In general, you can optimize software by removing the need for an address increment operation. To do so, generate load/store instructions that perform the address increment as well. For TIE register files, the XCC compiler can perform this optimization when the address update prototypes for the register file is specified.

For the color conversion example, add address update prototypes for the `vec128` register file (see the Prototype chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual* for more information). Following are the pre-increment TIE instructions and their protos for the `vec128` register file:

```
regfile vec128 128 8 vec128
immediate_range imm16x16 -128 112 16

operation st_vec128_ui {in vec128 Data, inout vec128 *Addr, in
imm16x16 offset}{out VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr             = tmpaddr;
     assign Addr              = tmpaddr;
     assign MemDataOut128     = Data;
}

proto vec128_storeiu {in vec128 v, inout vec128 *p, in immediate o}{}{
     st_vec128_ui v, p, o;
}

operation ld_vec128_ui {out vec128 Data, inout vec128 *Addr,
                    in imm16x16 offset}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr             = tmpaddr;
     assign Addr              = tmpaddr;
     assign Data              = MemDataIn128;
}

proto vec128_loadiu {out vec128 v, inout vec128 *p, in immediate o}{}{
     ld_vec128_ui v, p, o;
}
```

Adding these two instructions and their prototypes removes four `addi` instructions. To remove the two other address increment instructions, modify the `load64_extend` instructions to also increment the address.

The resulting assembly code, as shown in the profiling window of the Xplorer Benchmark view, is shown in the following example. The first column contains the cycle count for each assembly instruction. The second column is the instruction address.

```
32 60000762    loopgtz a15, 60000792 <ConvertYCbCrToRGB+0x82>
544 60000765    load64_extend vec1280, a8, 8
512 60000768    load64_extend vec1281, a9, 8
512 6000076b    vec_sub_scalar vec1280, vec1280, a14
512 6000076e    vec_mac_r acc1, vec1280, c1
512 60000771    vec_sub_scalar vec1281, vec1281, a14
512 60000774    vec_mac_r acc0, vec1281, c0
512 60000777    ld_vec128_ui vec1282, a10, 16
512 6000077a    vec_mac_r acc2, vec1280, c3
512 6000077d    vec_mac acc1, vec1281, c2
512 60000780    vec_addsat vec1280, vec1282, acc2
512 60000783    vec_addsat vec1281, vec1282, acc1
512 60000786    st_vec128_ui vec1281, a12, 16
512 60000789    st_vec128_ui vec1280, a13, 16
512 6000078c    vec_addsat vec1280, vec1282, acc0
512 6000078f    st_vec128_ui vec1280, a11, 16
  ConvertYCbCrToRGB+0x82
```

There are only 15 instructions in the innerloop. The innerloop is executed 32 times, so the first instruction of the loop should use 32 + 16*32= 544 cycles. All other instructions use 512 cycles, thus no pipeline bubbles are present in the innerloop.

Similarly, the address increment instructions can be removed when the post-increment TIE instructions and protos are specified:

```
regfile vec128 128 8 vec128
immediate_range imm16x16 -128 112 16

operation st_vec128_pi {in vec128 Data, inout vec128 *Addr, in
imm16x16 offset}{out VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr= Addr+offset;
     assign VAddr   = Addr;
     assign Addr    = tmpaddr;
     assign MemDataOut128= Data;
}
proto vec128_storeip {in vec128 v, inout vec128 *p, in immediate o}{}{
     st_vec128_pi v, p, o;
}

operation ld_vec128_pi {out vec128 Data, inout vec128 *Addr, in
imm16x16 offset}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr= Addr+offset;
     assign VAddr   =Addr;
     assign Addr    = tmpaddr;
```

```
        assign Data= MemDataIn128;
    }
    proto vec128_loadip {out vec128 v, inout vec128 *p, in immediate o}{}{
        ld_vec128_pi v, p, o;
    }
```

The inner loop also contains 15 instructions, as shown in the assembly code:

```
        loopgtz   a9, 82 <ConvertYCbCrToRGB+0x82>
            load64_extend  vec1280, a2, 8
            load64_extend  vec1281, a3, 8
            ld_vec128_pi   vec1282, a10, 16
            vec_sub_scalar vec1280, vec1280, a8
            vec_sub_scalar vec1281, vec1281, a8
            vec_mac_r      acc0, vec1281, c0
            vec_mac_r      acc1, vec1280, c1
            vec_mac_r      acc2, vec1280, c3
            vec_mac        acc1, vec1281, c2
            vec_addsat     vec1280, vec1282, acc0
            vec_addsat     vec1281, vec1282, acc2
            st_vec128_pi   vec1280, a13, 16
            st_vec128_pi   vec1281, a6, 16
            vec_addsat     vec1280, vec1282, acc1
            st_vec128_pi   vec1280, a7, 16
    <ConvertYCbCrToRGB+0x82>:
```

7.5 Using FLIX Instructions to Optimize Performance

For LX cores

As explained in Section 5.9, FLIX technology provides wide instructions, which contain multiple slots. Each slot can contain one or more Xtensa ISA or TIE instructions. Therefore, a FLIX instruction can execute multiple operations in parallel, thus achieving higher performance.

7.5.1 Manually Optimizing the Color Conversion C Example using FLIX

The innerloop of the color conversion example, as shown in Section 7.4, contains 15 instructions. These 15 instructions can be reduced to one FLIX instruction if all instructions could be executed in parallel using 15 or more slots. However, this would require a huge amount of hardware. Alternatively, if five or more slots are used, the innerloop can be executed in three FLIX instructions.

The following steps explain how to manually optimize the color conversion example using FLIX.

Step 1: Preparing the C Code

Pointers in C code can cause problems for FLIX, especially if multiple load/stores can occur per cycle. A simple example is when data could be loaded and stored in the same cycle. If the compiler cannot verify that these memory transfers are to different addresses, then it cannot put these into one FLIX instruction. This problem, called aliasing, is described in detail in the *Xtensa C and C++ Compiler User's Guide*. It also describes how to give hints to the compiler to solve this problem.

Step 2: Calculating the Number of Innerloop Instructions

The goal is to reduce the number of cycles of one iteration of the innerloop. The number of cycles should be the number of 16/24 core/TIE instructions and FLIX instructions inside the optimized loop.

The decision for how many instructions is influenced by Xtensa core features, cycle count requirements, and imposed TIE area limits. Take all these constraints into account to calculate the minimum number of instructions and the desired number of instructions. The number of desired instructions can be more than the minimum amount, if you want to trade off hardware area cost versus software performance.

The impact of FLIX on the hardware area is discussed in Chapter 9. For the color conversion example, you can limit the area increase of FLIX by using only one slot for SIMD multiply accumulate instructions (such as `vec_MAC` and `vec_MAC_R`). There are four SIMD multiply accumulate instructions per innerloop, which results in a four-cycle minimum caused by this restriction. The next important group of instructions is the load/store instructions, in which the example uses three loads and three stores per innerloop iteration. A base Xtensa core can execute one load/store per cycle, requiring a minimum of six instructions/cycles to execute the innerloop. Enabling the second load/store unit configuration option, as discussed in Section 5.2, can reduce this to three instructions/cycles.

The color conversion example uses a configuration with two load/store units with only one slot that contains SIMD multiply accumulate instructions.

By using two load store units, four is the minimum number of instructions inside the inner loop. This minimum number is the result of limiting the multiply accumulate instruction to one slot.

Step 3: Calculating the Number of Slots and Instructions per Slot

The number of instructions to be used (discussed in Step 2) provides a lower bound on the necessary number of slots; namely, the original number of instructions inside the innerloop, divided by the number of target instructions. Software pipelining is now used to assign the operations that are going to be used in each slot.

The previous step determined that the color conversion innerloop could be achieved in four FLIX instructions/cycles. The non-FLIX innerloop contained 15 instructions. Therefore, a minimum of four slots (15 divided by 4) is needed. The next step is to indicate which instructions are required in each slot. The best way to accomplish this is to schedule the original instructions (which henceforth will be called operations) into FLIX instructions.

The first step is to look at the code and place the instructions into groups based on the dependencies between instructions. The number of FLIX instructions/cycles determines the maximum number of instructions in a group. Therefore, for the Color Conversion example, four instructions can belong to any group. There can be dependencies between instructions in a group, but be aware that they can cause pipeline hazards.

The first group of the Color Conversion example executes three loads. The second group contains two vec_sub_scalar instructions because they are dependent on the result of two of the three load (`load64_extend`) instructions. This allows the multiply accumulate instructions in the third group that are dependent on the result of the vec_sub_scalar instructions to occur. There is one dependency between two multiply-accumulate instructions, but because this does not cause a pipeline bubble, these instructions can belong to the same group. The three vec_addsat instructions can generate the R, G and B values from the multiply-accumulate results in the fourth group. The fifth (and last) group can then write the results to memory.

The second step of software pipelining is to unroll the loop.[3] Next, schedule the different groups in different FLIX slots. Software pipelining is now used to prevent pipeline bubbles (as described in “Software Pipelining” on page 106) by separating groups by one loop iteration. Therefore, start with four FLIX instructions containing only group 1, which loads data for iteration 0 (remember that one loop iteration is four FLIX instructions). The next four FLIX instructions can now load data for iteration 1 in slot 0 and can perform the vec_sub_scalar for iteration 0. When 4 FLIX instructions contain all groups, this becomes the innerloop.

The following shows loop unrolling and scheduling for the color conversion example. In the example, instructions are reported only with the data they use or compute. The data names `y`, `cb`, `cr`, `R`, `G`, and `B` are followed by an iteration index.

```
load64_extend
load64_extend
ld_vec128_ui
NOP
load64_extend  vec_sub_scalar
load64_extend  vec_sub_scalar
ld_vec128_ui
NOP
load64_extend  vec_sub_scalar  R0=vec_MAC_R
load64_extend  vec_sub_scalar  vec_MAC_R Cr0
```

3. Software pipelining does not require unrolling. This step is more conceptual.

```
ld_vec128_ui   NOP             G0=vec_MAC
NOP            NOP             B0=vec_MAC_R
load64_extend  vec_sub_scalar  R1=vec_MAC_R   vec_addsat
load64_extend  vec_sub_scalar  vec_MAC_R Cr1  vec_addsat
ld_vec128_ui   NOP             G1=vec_MAC     vec_addsat
NOP            NOP             B1=vec_MAC_R   nop

INNERLOOP:
load64_extend  vec_sub_scalar  R2=vec_MAC_R   vec_addsat    st_vec_ui
load64_extend  vec_sub_scalar  vec_MAC_R Cr2  vec_addsat    st_vec_ui
ld_vec128_ui   NOP             G2=vec_MAC     vec_addsat    st_vec_ui
NOP            NOP             B2=vec_MAC_R   NOP           NOP
```

The result of loop unrolling requires five slots, whereas the non-FLIX innerloop contained 15 operations. The schedule using FLIX requires four instructions of five slots, which provides 20 operations. There are five NOP operations in four FLIX instructions. Therefore, it is possible to use only four slots. To achieve four slots, put the vec_sub_scalar operation into both the load and store slots. The four slots contain the following:

- Slot 0: load instructions and vec_sub_scalar
- Slot 1: multiply-accumulate instructions
- Slot 2: vec_ADDSAT instruction
- Slot 3: store instructions and vec_SUB_SCALAR instruction

Even though it is possible to pack the 15 operations into four FLIX instructions, it is still necessary to verify that the compiler can also pack these instructions into four FLIX instructions. The XCC compiler will deal with any resource (register/states) constraints.

Step 4: Reducing Hardware Cost & Power Consumption

Reducing hardware cost and power consumption is an optional step. After achieving an initial schedule, take the hardware cost factor into account and decide if you want to minimize the duplication of semantic blocks. For more information about reducing hardware costs, see Chapter 9.

An example of reducing hardware cost is the restrictions on all multiply accumulate instructions in the third slot as shown in the previous section. If this restriction was not in place, then executing multiply accumulate instructions in different slots results in duplication of the multiply accumulate semantic.

Step 5: Implementing the FLIX Instruction

To implement the FLIX instruction for the color conversion innerloop, add the following statements to the TIE file:

```
format f64 l64 {slot_ld, slot_mac, slot_add, slot_store}
```

```
slot_opcodes slot_ld {load64_extend, ld_vec128_ui, vec_SUB_SCALAR}
slot_opcodes slot_mac {vec_MAC_R, vec_MAC}
slot_opcodes slot_add {vec_ADDSAT}
slot_opcodes slot_store {st_vec128_ui, vec_SUB_SCALAR}
```

Step 6: Verifying the Schedule with XCC Compiler

Recompiling the color conversion example with the FLIX TIE added produces the following innerloop in pseudo code:

```
Instruction 1: {
     load64_extend vec1280, a8, 8;
     vec_mac_r acc1, vec1281, c1;
     vec_addsat vec1282, vec1284, acc1;
     vec_sub_scalar vec1283, vec1282, a14
}
Instruction 2: {
     ld_vec128_ui vec1284, a10, 16;
     vec_mac_r acc0, vec1283, c0;
     vec_addsat vec1282, vec1284, acc0;
     st_vec128_ui vec1282, a12, 16
}
Instruction 3: {
     load64_extend vec1282, a9, 8;
     vec_mac acc1, vec1283, c2;
     nop;
     st_vec128_ui vec1282, a13, 16
}
Instruction 4: {
     vec_sub_scalar vec1281, vec1280, a14;
     vec_mac_r acc0, vec1281, c3;
     vec_addsat vec1285, vec1284, acc0;
     st_vec128_ui vec1285, a11, 16
}
```

XCC was able to generate the four-cycle innerloop using a preamble that contains two partial innerloops. The four innerloop instructions are executed 32 * (16-2) = 448 times, which is the number of cycles for each of the innerloop instructions. Remember that the first instruction will have 448+32 cycles due to the loop instruction.

Step 7: Evaluating FLIX TIE

Now that the performance goals are achieved, check the other goals, such as the hardware area. It is a good idea to synthesize the resulting TIE and check that these area goals are met as well.

If the performance goals are not met, it might be necessary to modify the original TIE instructions or assign more instructions to FLIX slots.

Note: If you changed the TIE file to fix a constraint, return to Step 1.

8. Simulating TIE Instructions in a Native Environment

When you develop C/C++ code on an Xtensa processor, you compile it using the XCC compiler and simulate it using the Xtensa Instruction Set Simulator. This simulation provides an accurate representation of the code running on real Xtensa hardware. However, in some situations, you may want to compile and run the code on host or native development platforms, such as a personal computer running a Linux or Windows operating system. For example, you may have your own functional system simulation environment to only verify the functionality of the system, or you may want to verify the functionality of the application code, or want to verify the functionality of a specific TIE instruction on a huge data set.

If the code is written using ANSI C/C++, this is easily done by using any host compiler such as GNU gcc or g++. However, this is not possible if the code contains any TIE constructs such as TIE instruction intrinsics, ctypes corresponding to user defined register files, and so forth. If you try to compile such code using gcc/g++, the compiler will not recognize these constructs and the compilation will fail.

The TIE compiler provides a special mechanism to enable you to simulate C/C++ code with TIE intrinsics on an x86 platform running a Linux or Windows operating system. When you compile a TIE file and generate a TDK, the TIE compiler can create a set of C files, a set of C++ files, and a header file to enable this native simulation. The C/C++ files contain functional implementations of the TIE instructions. They are descriptions of the computations performed by the corresponding instructions, generated from the semantic description of the user TIE instructions. The header file contains declarations of all these functions and macros for TIE intrinsics. By including the header file in your application code and adding the TIE compiler-generated C/C++ source files to your project, you can successfully compile your application using native compilers. The resulting executable can be used to simulate your application, including TIE intrinsics, on the x86 platform.

8.1 Recommendations for Native Simulation

This section summarizes some recommendations for compiling and running Xtensa target code using cstub files on an x86 platform. For more information on each of these issues, refer to the sections that follow.

Tensilica supplies cstub files for both C and C++ compilers. However, Tensilica strongly recommends using a C++ compiler to compile your native simulation. Even if your Xtensa application is written in C, in many cases you may need to use the C++ cstub file to properly emulate Xtensa instructions. If the Xtensa application utilizes any of the internal modules provided by Tensilica (such as Vectra LX, HiFi2, BBE16, and others),

you must use the C++ cstub file. Note that both the target code and the cstub file must be compiled with a C++ compiler. Refer to your compiler manual to determine how to force C++ compilation. Refer to Section 8.3 “Use of C++ v/s C Files for cstub Simulation” on page 120 for more information.

Because the x86 processor is little-endian, you can natively simulate little-endian Xtensa processors. If the Xtensa processor is big-endian, the TIE file may require additional constructs to simulate natively. Refer to Section 8.4 “Simulating Applications for Big-Endian Xtensa Configurations” on page 121 for more information.

The following recommendations refer to Section 8.8 “Pitfalls of Native Simulation” on page 144:

- **Alignment**: The Tensilica XCC compiler automatically aligns wide data types to the size of the data type. It also aligns data arrays to the size of the array, up to the maximum memory interface width, or 16 bytes, whichever is wider. This makes it easier to use wide data types in your application. However, native x86 compilers do not automatically align data arrays in this way. If your application expects data to be aligned, you should explicitly align arrays and static data with an alignment attribute. Refer to your compiler manual to determine how to force data alignment.
- **Addressing**: Xtensa is a 32-bit address architecture. To run on a 64-bit x86 host platform, you must compile your application and cstub file in 32-bit mode. You cannot link other 64-bit libraries to a 32-bit native simulation.
- **Type casting**: Native simulation enforces strict type checking on arguments to TIE intrinsics, and does not allow type casting of those arguments.
- **Performance**: Native simulations emulate the behavior of many Xtensa instructions using cstub files. As a result, native simulations can be used for functional simulation, but not for accurately predicting the performance of Xtensa applications. Unlike the Xtensa ISS, native simulations do not provide profiling or cycle count information for Xtensa code.

8.2 Running a Native Simulation

This section describes how to run a native simulation of your application code with TIE intrinsics. This section assumes that you have downloaded and installed Xtensa tools, downloaded and installed an Xtensa configuration without any TIE, and have written the TIE description (*foo.tie*) that you would like to simulate.

The first step is to run the TIE compiler to generate a TIE Development Kit (TDK) as follows with cstub:

```
tc -d tdk -cstub foo.tie
```

This creates a `tdk` directory in the current working directory, which contains several files, including a directory named `cstub`. The directory includes all the files are needed to execute a native simulation, which are collectively referred to as cstub files[1]. Note that the generation of these files when compiling a TDK is optional, and you must use the `-cstub` command line argument to generate them.

If you compile the TIE file in Xtensa Xplorer, double-click the TIE file for which you want the cstub files generated, from the Configuration TIE Overview tab, click **Compile Options**, and then select the TC compile time option, Generate cstub files for native simulation for host processor. After the compilation is finished, the cstub files are saved in the `XtensaInfo/Models/<`*`config`*`>.tdk/cstub` directory. Export the files to the desired directory using the standard exporting procedure.

If you build and install a processor configuration in Xtensa Xplorer, the cstub files for the configuration are automatically generated. The cstub files are saved in the *<config dir>*`/src/cstub` directory. Export the files to the desired directory using the standard exporting procedure.

The header file `xtensa/tie/foo.h` includes several type definitions and function prototypes[2].The files `cstub-foo-<N>.cpp` are C++ files that contain a functional implementation of the TIE instructions and data types for simulating on an x86 platform. Similarly, the file `cstub-foo-<N>.c` are C files that contain a functional implementation of the TIE instructions and data types[3]. You must choose to use either the C++ or the C compiler for all source files that contain TIE intrinsics or ctypes. If you use the C compiler, some TIE features cannot be supported with cstubs. If you use the C++ compiler, your application source code must be compatible with C++. Section 8.3 on page 120 discusses the trade-offs in the use of a C++ versus a C compiler, and explains the advantages of using C++. The rest of this section assumes that the C++ file is used. If your TIE description contains external interfaces such as import wires, exported states, input queues, output queues, or lookups, a fourth file (`xtensa/tie/cstub-extif.h`) is generated. This file contains the description of the callback functions for the external interfaces and is usually used by the modeling functions of such devices. See Section 8.5 on page 124 for details.

When you run an Xtensa simulation using the ISS, you include a header file for the TIE instructions such as:

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

in every C/C++ file that contains TIE intrinsics, as explained in Section 4.1.2 on page 22. When running a native simulation, the `#include` statement above stays the same.

In the case where there is only one application source file, for example, `app.cpp`, you can compile your application as:

1. Before RE-2013.2, the cstub files are in the `tdk` directory.

2. Before RE-2013.2, the header file is named `cstub-foo.h` and it is in the same directory of cstub source files.

3. Before RE-2013.2, both C/C++ cstub sources are single files named `cstub-foo.cpp` and `cstub-foo.c`.

```
g++ -I<tdk>/cstub app.cpp <tdk>/cstub/cstub-foo-*.cpp
```

This creates an executable for the host platform which can then be run like any other application on that platform. You can use other g++ compiler switches (such as `-O2` for optimization level 2) with the above command.

8.3 Use of C++ v/s C Files for cstub Simulation

The TIE compiler generates a C++ file in the TDK directory that contains a functional implementation of the TIE instructions and data types. It also generates a C file in the same directory, which contains a functional implementation of the TIE instructions and data types. You only need to use one of these files, depending upon whether you intend to use a C++ or a C compiler to compile your application code.

Tensilica recommends that you use the C++ files generated by the TIE compiler for native simulation. In fact, if your application requires implicit type conversions, casts between a TIE type (including xtbool) and any other type or operator overloading, you must use the C++ files. Thus if your TIE code has protos like `<from>_mtor_<to>`, `<from>_rtor_<to>` or `<from>_rtom_<to>`, and your application code relies on the compiler's use of these protos for type conversion, you cannot use the C files generated by the TIE compiler. If you do use the C files, you will get a compile failure when compiling your application code. Note that if your TIE has the conversion protos, and they are always explicitly used as intrinsics in your application code, you may still be able to use the C files for native simulation.

When you use the C++ files generated by the TIE compiler for native simulation, you must use a C++ compiler to compile all of your application code that uses TIE extensions. It is not enough to use the C++ compiler to compile only the cstub files. Note that your application code does not have to be written in C++. It can be written in C, but has to be compiled using a C++ compiler. C++ compilers perform stricter type checking compared to C compilers, so it is possible that your C application code may generate compilation errors when compiled with a C++ compiler. For example, a common error is missing explicit type conversion for pointers. Most of these compilation errors are easy to fix.

While the use of C++ cstub files is preferred for native simulation, for many designs, use of the C files will work just fine. If you do use the C files, be aware of one additional restriction. In your application code: you cannot dereference ctype pointers when any of the following conditions apply. Instead, you need to use the appropriate intrinsic directly.

- The ctype's load or store protos contain more than one instruction.
- The instructions implementing the ctype's load or store protos contain computations other than direct assignment.
- The instructions implementing the ctype's load or store protos contain `cstub_swap` constructs or `ByteDisable` interfaces.

8.4 Simulating Applications for Big-Endian Xtensa Configurations

You can configure the Xtensa processor to either a little-endian (LE) processor or a big-endian (BE) processor. The Xtensa ISS supports the simulation of applications on the processors of either endianess. However, for native simulation, it is assumed that the host processor is an x86 processor, which is little endian. When simulating an application for a little-endian Xtensa processor on an x86 host, the endianess is matched and no special care is necessary. However, simulating an application for a big-endian Xtensa processor on an x86 host presents some special challenges because of the mismatched endianess. This section discusses the conditions where it is possible to perform such a simulation, and conditions where it is not possible to do so.

8.4.1 Endian-Dependent Application Code

If your application code is endian dependent, you get different results when you run it on a big-endian processor and a little-endian processor. If you intend to run your code on a big-endian Xtensa processor, you will not be able to run native simulations on an x86 platform. Consider the following piece of C code, which is endian dependent:

```
unsigned int data = 0x11223344; // data is a 32-bit integer
unsigned int *pdata = &data;    // int pointer pointing to data
char *pchar = (char *) pdata;   // char pointer pointing to data
char res = *pchar;              // print "first" char of data

printf ("Value of char is %x \n", res);
```

Executing this code on a little-endian processor prints the value 44, while executing it on a big-endian processor prints the value 11. If the application code is written to run on a big-endian processor and expects the value 11, it will fail when executed on a little-endian processor. This failure is independent of any TIE, thus the problem cannot be addressed by the TIE compiler. Application code which is endian dependent and written for big-endian Xtensa processors cannot be simulated natively using the cstub mechanism directly. You need to port the C code (software part) of the application to a little-endian processor. A TIE construct is introduced so that you can avoid porting the TIE code (hardware part) of the application to a little-endian processor, as described in the following sections.

8.4.2 Endian-Dependent TIE Code

Endian-dependent TIE code can cause problems when simulating on the native platform using the cstub code generated by the TIE compiler. For example, if your TIE instructions load or store multiple data values in one memory access, you are likely to have endian dependence in your TIE code. Consider the following TIE code, which is written for a big-endian Xtensa processor:

```
state st0 8 add_read_write
state st1 8 add_read_write
state st2 8 add_read_write
state st3 8 add_read_write

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;
}
```

This instruction loads a 32-bit value from memory, and saves it to four different 8-bit TIE states. Assuming the values being read from memory were stored as four characters, this TIE code is endian dependent. Simulating it on a big-endian Xtensa processor will generate one set of values for the states `st0-st3`. Simulating it on a little-endian x86 host will result in a different set of values being assigned to states `st0-st3`. The value assigned to state `st0` in one simulation is assigned to `st3` in the other.

For endian-dependent load/store instructions, the TIE language provides a construct that facilitates simulating applications for a big-endian Xtensa processor on an x86 host. This construct, `cstub_swap`, is used to swap the bits of the `MemDataIn` and `MemDataOut` interfaces to account for the endianness change, so that the instruction semantic treats the `MemDataIn` or `MemDataOut` interface in big-endian order even though its memory layout is in little-endian order. The detailed syntax of the construct is in the Cstub Swap Sections chapter of the *Tensilica Instruction Extension (TIE) Language Reference Manual*. For the example of the `simd_load` instruction, this construct is used as follows:

```
cstub_swap myswap {simd_load} {MemDataIn32[ 7: 0],  /* st3 */
                               MemDataIn32[15: 8],  /* st2 */
                               MemDataIn32[23:16],  /* st1 */
                               MemDataIn32[31:24]   /* st0 */
                              }
```

The `cstub_swap` construct for the `simd_load` instruction specifies that the 32-bit `MemDataIn` coming from the memory must be swapped at the byte boundary before being presented to the instruction semantic in the native simulation environment. In general, swapping mechanisms depends on the sizes of the data values and the number of data values to be loaded or stored. Following is the procedure to determine the swapping mechanism:

1. Implement the instruction for target processor configuration.

2. Determine the size and position of each data value in the MemDataIn or MemDataOut interface. Write a short comment to visualize the size and position. For instruction simd_load in the example, the comment is:
```
/* 31...24 23...16 15...8 7...0  <-- Bits of MemDataIn32
     st0    st1     st2    st3   <-- Loaded data */
```
3. Write the cstub_swap construct. Follow the direction from right to left of the comment to write the concatenation of the MemDataIn or MemDataOut interface. See the simd_load example for details.

This swap is performed in the generated code, and only when the TIE code is compiled with a big-endian Xtensa configuration. Further, the cstub_swap construct has no impact on hardware and it is not used for Xtensa ISS; it is only used for native simulation on the x86 platform.

The cstub_swap construct can also be used with store instructions. In the following example, the mystore instruction stores one short, two char, and one int variables in one cycle. The instruction assumes that the variables are packed in 8-byte boundaries.

```
state st0 16 add_read_write
state st1 8  add_read_write
state st2 8  add_read_write
state st3 32 add_read_write

/*  63......48 47...40 39...32 31............0
       st0       st1     st2         st3          */
operation mystore {in AR *addr, in AR disable}
                  {out VAddr, out MemDataOut64, out StoreByteDisable,
                   in st0, in st1, in st2, in st3} {
       assign VAddr = addr;
       assign MemDataOut64 = {st0, st1, st2, st3};
       assign StoreByteDisable = disable;
}

cstub_swap mystore { mystore } {
       MemDataOut64[31: 0],  /* st3 */
       MemDataOut64[39:32],  /* st2 */
       MemDataOut64[47:40],  /* st1 */
       MemDataOut64[63:48]   /* st0 */
}
```

The cstub_swap construct specifies how the bits of MemDataOut coming from the instruction semantic should be swapped before being written to memory. Note that for the mystore instruction, the swapping is not all done at byte boundaries; swapping is done based on the width of the variable, which is the same as the width of each state st0-st3 that is concatenated to provide data for the MemDataOut64 interface. The mystore instruction uses the StoreByteDisable interface to selectively enable and disable the writing of certain bytes. Note that no cstub_swap specification is required for

the StoreByteDisable interface; it is automatically derived from the cstub_swap specification of the MemDataOut interface. However, one restriction applies to the values of the StoreByteDisable interface: you can only disable writing the entire data value—you cannot disable writing portions of the data values, for example, the first byte of state st0 or the second byte of state st3.

8.4.3 stub_swap Construct Limitations

With the cstub_swap construct, you can successfully simulate most applications with TIE intrinsics for big-endian processors on x86 host machines. However, if your application has the following attributes, you still cannot simulate them natively, even with the help of the cstub_swap construct. In those situations, you need to either rewrite your application to avoid using those TIE instructions, or rewrite the TIE code for little-endian processor for native simulation purpose only.

- A TIE instruction loads or stores multiple data values from or to memory. However, the number of data values, or the sizes of the data values are determined at runtime. This usually happens when you find that you need to write two different cstub_swap constructs for the same instruction.
- A TIE instruction loads or stores multiple data values from or to memory. The values specified in a LoadByteDisable or StoreByteDisable interface may disable load or store portions of the data values.
- The application uses unaligned load instruction to load multiple data values from memory. It relies on the load store unit of the target processor to automatically rotate the loaded values to align the address. That is, you select the "Align address" configuration in the XPG.

8.5 Simulating External Interfaces

The TIE language allows you to create instructions that communicate with devices external to the Xtensa processor. For example, an import wire reads from an external interface, while an exported state makes its value visible outside of the processor on an external interface. Instructions that access TIE queues and TIE lookups also fall into this category. When generating a cstub, the TIE compiler can only create a partial model of the instructions that access external interfaces. The TIE compiler can model the computation being performed inside the Xtensa processor, but it does not model the external devices.

Simulation of TIE instructions that interface to external devices is facilitated through a callback function mechanism. The designer needs to implement the behavior of the external device through the callback functions and register them to the cstub code. The TIE compiler generates a function prototype for each kind of external device, and the appropriate functions are called during a native simulation of the TIE instructions that ac-

cess external interfaces. If your TIE file contains an external interface, the TIE compiler will generate header file `cstub-extif.h`, which contains the external interface data structures and callback function prototypes. You may need to include the header file when you implement the external devices.

8.5.1 Import Wire

An import wire allows a TIE instruction to read the value of an external interface. The data structure used by `cstub` for an import wire interface is:

```
typedef struct cstub_ImportWire_struct {
   void *user_object;  /* ImportWire device */
   const char *name;   /* Name of ImportWire */
   unsigned width;     /* Width of ImportWire */
} cstub_ImportWire_t;
```

- `user_object` points to the external device that provides the data for the import wire interface.
- `name` is the name of the `import_wire` as declared in the TIE file.
- `width` is the width of the `import_wire` as declared in the TIE file.

The callback functions for import wires take the following form:

```
typedef void (*cstub_ImportWire_func_t)
     (const cstub_ImportWire_t *data_struct, unsigned *in_data);
```

- `data_struct` is the import wire data structure described above. A pointer to this data structure is passed as an argument to the callback function.
- `in_data` is a pointer to the place holder of the import wire value. The callback function must store the import wire value to this address. The value to be saved must be an array of unsigned int type. The number of entries of the array must be the smallest integer greater than or equal to (width + 31)/32, where width is the width of the import wire. The first entry of the array maps to bit 31 to 0 of the import wire; the second entry maps to bit 63 to 32, and so on. The least significant bit of the first integer in the array corresponds to bit 0 of the import wire. This mapping is independent of the endianess of the Xtensa processor. All the bits in the array that are outside the range of the import wire must be zero.
- Each import wire device data structure and its callback function must be registered before it can be used in the application code. If an unregistered import wire is accessed in the native simulation, the simulation will terminate with an error message. It is not necessary to register an import wire that is not accessed in the simulation. By providing a registration mechanism, you can reuse the same callback function for different import wire devices. The following function is provided by the cstub code to register import wires:

```
extern void cstub_register_<import wire name>
     (void *user_object, cstub_ImportWire_func_t callback);
```

- *import wire name* is the name of the import wire.
- `user_object` is a pointer to the object that represents the external device. It will be stored in the `user_object` field in `cstub_ImportWire_t`. It needs to be live throughout the simulation.
- `callback` is a function pointer to the callback function.

The following example illustrates how the import wire device is used in the native simulation.

In this example, a processor is connected to an external device via a 20-bit import wire `iw`. The TIE instruction `get_iw` retrieves data from `iw`. A small program running on the processor prints out the read data to the screen. For native simulation purpose, each time the import wire is accessed, the external device reads a value from a file and passes it to the processor.

The TIE file `import_wire.tie` is:

```
import_wire iw 20
operation get_iw {out AR data} {in iw} {
     assign data = iw;
}
```

The C file `import_wire.c` follows. Because of the external device and callback function implementations, the code cannot run on the Xtensa platform. You need to modify the code for XTMP or XTSC simulation environment:

```
#include <stdio.h>
#include <stdlib.h>
#include "xtensa/tie/import_wire.h"

#define NUM 100

/* Callback function */
static void ImportWire(const cstub_ImportWire_t *iw,
                       unsigned int *data) {
  FILE *fp = (FILE *) iw->user_object;
  unsigned value;
  fscanf(fp, "%x", &value);
  data[0] = value & ((1 << iw->width) - 1);
                                     /* Mask to correct # bits */
}

int main() {
  unsigned int value, i;
  FILE *fp;
```

```
  fp = fopen("iw_in.txt", "r");       /* Open file for import wire */
  if (fp == NULL) {
    fprintf(stderr, "ERR: cannot open file iw_in.txt\n");
    exit(1);
  }

  cstub_register_iw(fp, &ImportWire); /* Register import wire */
                                      /* Attach fp as user_object */

  for (i = 0; i < NUM; i++) {
    value = get_iw();                 /* Get import wire value */
    printf("0x%x\n", value);          /* Print import wire value */
  }

  fclose(fp);                         /* Close file for import wire */
  return 0;
}
```

8.5.2 Exported State

An exported state exposes the value of the state to an external interface. The data structure used by cstub for an exported state interface is:

```
typedef struct cstub_ExportState_struct {
  void *user_object;  /* ExportState device */
  const char *name;   /* Name of ExportState */
  unsigned width;     /* Width of ExportState */
} cstub_ExportState_t;
```

- `user_object` points to the external device that the value of the exported state is sent to.
- `name` is the name of the `export_state` as declared in the TIE file.
- `width` is the width of the `export_state` as declared in the TIE file.

The callback functions for exported states take the following form:

```
typedef void (*cstub_ExportState_func_t)
     (const cstub_ExportState_t *data_struct,
      const unsigned *out_data);
```

- `data_struct` is the exported state data structure described above. A pointer to this data structure is passed as an argument to the callback function.
- `out_data` is a pointer to the exported state value. The callback function should only retrieve n entries of the array pointed by `out_data` where n is the smallest integer greater than or equal to (width + 31)/32. The first entry of the array maps to bit 31 to

0 of the exported state; the second entry maps to bit 63 to 32 and so on. The least significant bit of the first integer in the array corresponds to bit 0 of the exported state. This mapping is independent of the endianess of the Xtensa processor.

- Similar to import wire, each exported state device data structure and the callback function must be registered before it can be used. The following function is provided by the cstub code to register exported states:

```
extern void cstub_register_<exported state name>
     (void *user_object, cstub_ExportState_func_t callback);
```

- *exported state name* is the name of the exported state.
- `user_object` is a pointer to the object that represents the external device. It will be stored in the `user_object` field in `cstub_ExportState_t`.
- `callback` is a function pointer to the callback function.

The following example illustrates how the exported state device is used in native simulation.

In this example, a processor has a 40-bit exported state `exs`. For native simulation purpose, each time the state `exs` is assigned a new value (the new value may be the same as the old value), the external device connected to the processor via the exported state interface records the value in a file. A small program running on the processor randomly generates values and put them to the exported state.

The TIE file `export_state.tie` is:

```
state exs 40 40'b0 add_read_write export
operation put_exs {in AR dh, in AR dl} {out exs} {
     assign exs = {dh, dl};
}
```

The C file `export_state.c` is:

```
#include <stdio.h>
#include <stdlib.h>
#include "xtensa/tie/export_state.h"

#define NUM 100

static int get_num_words(int bits) {
  return (bits + 31) >> 5;
}

/* Export state callback function */
static void ExportState(const cstub_ExportState_t *exs,
                        const unsigned *data) {
  FILE *fp = (FILE *)exs->user_object;
```

```
  int i, first = 1;
  for (i = get_num_words(exs->width) - 1; i >= 0; i--) {
    if (first)
      first = 0;
    else
      fprintf(fp, "\t");
    fprintf(fp, "0x%x", data[i]);
  }
  fprintf(fp, "\n");
}

int main() {
  unsigned int ldata, hdata, i;
  FILE *fp;

  fp = fopen("exs.txt", "w");   /* Open file for export state */
  if (fp == NULL) {
    fprintf(stderr, "ERR: cannot open file exs.txt\n");
    exit(1);
  }

  cstub_register_exs(fp, &ExportState); /* Register export state */
                                        /* Attach fp as user_object */

  for (i = 0; i < NUM; i++) {
    ldata = rand();                     /* Randomly generate data */
    hdata = rand();
    put_exs(hdata, ldata);              /* Put data to export state */
    printf("0x%x\t0x%x\n", hdata & 0xFF, ldata);
  }

  fclose(fp);                         /* Close file for export state */
  return 0;
}
```

8.5.3 Input Queue

An input queue allows TIE instructions to read data from a FIFO-like device. The detailed information on input queues can be found in the Queues Sections chapter of the *Tensilica Instruction Extension (TIE) Language Reference Manual*. In the cstub environment, the data structure for an input queue interface is:

```
typedef struct cstub_InputQueue_struct {
  void *user_object;  /* InputQueue device */
  const char *name;   /* Name of InputQueue */
  unsigned width;     /* Width of InputQueue */
} cstub_InputQueue_t;
```

- `user_object` points to an external FIFO-like device that can be viewed as an input queue.
- `name` is the name of the `input_queue` as declared in the TIE file.
- `width` is the width of the `input_queue` as declared in the TIE file.

The callback functions for input queues take the following forms.

```
typedef unsigned (*cstub_InputQueue_Empty_func_t)          /* Empty */
                 (const cstub_InputQueue_t *data_struct);
typedef void (*cstub_InputQueue_Data_func_t)               /* Data  */
             (const cstub_InputQueue_t *data_struct,
              unsigned is_peek, unsigned *in_data);
```

- `data_struct` is the input queue data structure described above. A pointer to this data structure is passed as an argument to the callback functions.
- `is_peek` indicates whether the queue access is a peek, which means the data in the queue is accessed by the processor, but the data is not popped from the queue. If `is_peek` is non-zero, the data in the queue should not be popped. If `is_peek` is zero, the data in the queue should be popped.
- `in_data` is a pointer to the place holder of the input queue value. The callback function must store the input queue data value to this address. The value to be saved must be an array of unsigned int type. The number of entries of the array must be equal to the smallest integer greater than or equal to (width + 31)/32, where width is the width of the input queue. The first entry of the array maps to bit 31 to 0 of the input queue; the second entry maps to bit 63 to 32, and so on. The least significant bit of the first integer in the array corresponds to bit 0 of the input queue data. This mapping is independent of the endianess of the Xtensa processor. All the bits in the array that are outside the range of the input queue must be zero.
- The `Empty` function is called when the queue empty port is accessed. It should check whether the queue is empty and return the status immediately. The `Data` function is called when the queue data port is accessed. It should wait until the queue is not empty, put the data to the address pointed by `in_data`, optionally pop the data from the queue (depending on the value of `is_peek`), and return.

Each input queue device data structure and callback function must be registered before it can be used. The following function is provided by the cstub code to register the input queues:

```
extern void cstub_register_<input queue name>(void *user_object,
    cstub_InputQueue_Empty_func_t empty_callback,
    cstub_InputQueue_Data_func_t data_callback);
```

- *`input queue name`* is the name of the input queue.
- `user_object` is a pointer to the object that represents the external device. It will be stored in the `user_object` field in `cstub_InputQueue_t`.

- `empty_callback` is a function pointer to the empty callback function.
- `data_callback` is a function pointer to the data callback function.

The following example illustrates how the input queue is used in the native simulation.

In the example, the processor has a 32-bit input queue. TIE instructions may check whether the queue is empty, peek the queue data, or pop the queue data. For native simulation purpose, the input queue reads data from a file. The values in the file are presented in a sequence of tuples: <empty data>. The first entry of the tuple indicates whether the queue is empty. An non-zero value means the queue is empty and hence the second value is not used. The second value is the input queue data to be read in.

Following is the TIE file `input_queue.tie`:

```
queue inq 32 in

operation get_inq {out AR data} { in inq } {
     assign data = inq;
}

operation chk_inq_notrdy {out AR notrdy} {in inq_NOTRDY} {
     assign notrdy = inq_NOTRDY;
}

operation nonblock_peek_inq {out AR data} {in inq, out inq_KILL} {
     assign data = inq;
     assign inq_KILL = 1'b1;
}

operation block_peek_inq {out AR data} {in inq, out inq_KILL,
                                        in inq_NOTRDY} {
     assign data = inq;
     assign inq_KILL = ~inq_NOTRDY;
}
```

The header file `iq.h` for input queue device is:

```
#include <stdlib.h>
#include "xtensa/tie/cstub-extif.h"

/* Input queue data structure */
typedef struct queue {
  unsigned *data;        /* First data of the queue */
  unsigned empty;        /* Is queue empty ? */
  FILE *fp;              /* Where queue data are saved */
} tqueue;

/* Init and free functions */
```

```
extern tqueue * InputQueue_inq_init();
extern void InputQueue_inq_free(tqueue *inq);

/* Callback functions */
extern unsigned int InputQueue_Empty(const cstub_InputQueue_t *queue);
extern void InputQueue(const cstub_InputQueue_t *queue,
                       unsigned int is_peek,
                       unsigned int *inq);
```

The C file `iq.c` for input queue device is:

```
#include <stdio.h>
#include <string.h>
#include "iq.h"

extern tqueue * InputQueue_inq_init() {
  tqueue *inq = (tqueue *)calloc(1, sizeof(tqueue));
  inq->data = (unsigned int *)calloc(1, sizeof(unsigned));
  inq->empty = 1;
  inq->fp = fopen("inq_in.txt", "r");
  if (inq->fp == NULL) {
    fprintf(stderr, "ERR: cannot open file inq_in.txt\n");
    exit(1);
  }
  return inq;
}

extern void InputQueue_inq_free(tqueue *inq) {
  fclose(inq->fp);
  free(inq->data);
  free(inq);
}

/* callback function for input queue empty port */
extern unsigned int InputQueue_Empty(const cstub_InputQueue_t *queue)
{
  tqueue *que = (tqueue *)queue->user_object;
  return que->empty != 0;
}

/* callback function for input queue data port */
extern void InputQueue(const cstub_InputQueue_t *queue,
                       unsigned int is_peek,
                       unsigned int *inq) {
  tqueue *que = (tqueue *)queue->user_object;

  /* First get the next valid data from input file */
  while(que->empty) {
    fscanf(que->fp, "%x\t%x", &que->empty, que->data);
```

```
  }
  /* Then pass data to the processor */
  inq[0] = que->data[0];

  if(is_peek) /* Do not pop data on peek */
    return;

  /* Pop data. Get next data from file */
  fscanf(que->fp, "%x\t%x", &que->empty, que->data);
}
```

Following is the application file running on the processor `input_queue.c` :

```
#include <stdio.h>
#include "xtensa/tie/input_queue.h"
#include "iq.h"

#define NUM 200

int main() {
  unsigned int i, notrdy, nonblock_peek, block_peek, get;
  tqueue *que;

  que = InputQueue_inq_init();
  cstub_register_inq(que, &InputQueue_Empty, &InputQueue);

  for (i = 0; i < NUM; i++) {
    notrdy = chk_inq_notrdy();           /* Is queue ready ? */
    printf("notrdy = 0x%x\n", notrdy);
    nonblock_peek = nonblock_peek_inq(); /* Nonblock peek queue data */
    printf("nonblock_peek = 0x%x\n", nonblock_peek);
    block_peek = block_peek_inq();       /* Blocking peek queue data */
    printf("block_peek = 0x%x\n", block_peek);
    get = get_inq();                     /* Pop queue data */
    printf("get = 0x%x\n\n", get);
  }

  InputQueue_inq_free(que);
  return 0;
}
```

8.5.4 Output Queue

An output queue allows TIE instructions to write data to a FIFO-like device. Detailed information on output queues is in the Queue Sections chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. In the cstub environment, the data structure for an output queue interface is:

```
typedef struct cstub_OutputQueue_struct {
  void *user_object;  /* OutputQueue device */
  const char *name;   /* Name of OutputQueue */
  unsigned width;     /* Width of OutputQueue */
} cstub_OutputQueue_t;
```

- `user_object` points to an external FIFO-like device that can be viewed as an output queue.
- `name` is the name of the `output_queue` as declared in the TIE file.
- `width` is the width of the `output_queue` as declared in the TIE file.

The callback functions for output queues take the following form:

```
typedef unsigned (*cstub_OutputQueue_Full_func_t)           /* Full */
                 (const cstub_OutputQueue_t *data_struct);
typedef void (*cstub_OutputQueue_Data_func_t)               /* Data */
             (const cstub_OutputQueue_t *data_struct,
              unsigned reserve_only,
              const unsigned *out_data);
```

- `data_struct` is the output queue data structure described above. A pointer to this data structure is passed as an argument to the callback functions.
- `out_data` is a pointer to the output queue value. The callback function should only retrieve n entries of the array pointed by `out_data`, where n is the smallest integer greater than or equal to (width + 31)/32. The first entry of the array maps to bit 31 to 0 of the output queue; the second entry maps to bit 63 to 32, and so on. The least significant bit of the first integer in the array corresponds to bit 0 of the output queue data value. This mapping is independent of the endianess of the Xtensa processor.
- `reserve_only` indicates whether to push the data to the queue. If `reserve_only` is zero, the data should be pushed to the queue. If `reserve_only` is non-zero, the data should not be pushed to the queue. Instead, an entry is reserved for future queue accesses.

The `Full` function is called when the queue full port is accessed. It should check whether the queue is full and return the status immediately. The `Data` function is called when the queue data port is accessed. It should wait until the queue is not full, optionally push the data to the queue depending on the value of `reserve_only`, and return.

Each output queue data structure and the call back function must be registered before it can be used. The following function is provided by the cstub code to register the output queues:

```
extern void cstub_register_<output queue name>(void *user_object,
     cstub_OutputQueue_Full_func_t full_callback,
     cstub_OutputQueue_Data_func_t data_callback);
```

- *`output queue name`* is the name of the output queue.

- `user_object` is a pointer to the object that represents the external device. It will be stored in the `user_object` field in `cstub_OutputQueue_t`.
- `full_callback` is a function pointer to the full callback function.
- `data_callback` is a function pointer to the data callback function.

The following example illustrates how the output queue is used in native simulation.

In this example, a processor is connected to a 40-bit output queue. A few TIE instructions check whether the queue is full, block push and non-block push data to the queue. For native simulation purpose, the queue writes down every data it receives to a file. A program on the processor pushes random data to the queue and prints out the pushed data to the screen. You can check whether the data the queue receives is the same as the data being pushed.

The TIE file `output_queue.tie` is:

```
queue oq 40 out

operation put_oq_block {in AR hdata, in AR ldata} { out oq } {
     assign oq = {hdata, ldata};
}

operation put_oq_nonblock {out AR finished, in AR hdata, in AR ldata}
                          { out oq, out oq_KILL, in oq_NOTRDY } {
     assign oq = {hdata, ldata};
     assign oq_KILL = oq_NOTRDY;
     assign finished = !oq_NOTRDY;
}

operation chk_oq_notrdy {out AR notrdy} {in oq_NOTRDY} {
     assign notrdy = oq_NOTRDY;
}
```

The header file `oq.h` for the output queue device is:

```
#include <stdlib.h>
#include "xtensa/tie/cstub-extif.h"

/* output queue data structure */
typedef struct queue {
  unsigned full;     /* Is queue full ? */
  FILE *fp;          /* Where queue data are pushed */
} tqueue;

/* Init and free functions */
extern tqueue * OutputQueue_oq_init();
extern void OutputQueue_oq_free(tqueue *oq_fp);
```

```
/* Callback functions */
extern unsigned OutputQueue_Full(const cstub_OutputQueue_t *queue);
extern void OutputQueue(const cstub_OutputQueue_t *queue,
                        unsigned int reserve_only,
                        const unsigned int *data);
```

The C file `oq.c` for the output queue device is:

```
#include <stdio.h>
#include <string.h>
#include "oq.h"

extern tqueue * OutputQueue_oq_init() {
  tqueue *que = (tqueue *)calloc(1, sizeof(tqueue));
  que->fp = fopen("oq.txt", "w");
  if (que->fp == NULL) {
    fprintf(stderr, "ERR: cannot open file oq.txt\n");
    exit(1);
  }
  return que;
}

extern void OutputQueue_oq_free(tqueue *que) {
  fclose(que->fp);
  free(que);
}

/* callback function for output queue full port */
extern unsigned int OutputQueue_Full(const cstub_OutputQueue_t *queue)
{
  tqueue *que = (tqueue *)queue->user_object;
  return que->full != 0;
}

static int get_num_words(int bits) {
  return (bits + 31) >> 5;
}

/* callback function for output queue data port */
extern void OutputQueue(const cstub_OutputQueue_t *queue,
                        unsigned int reserve_only,
                        const unsigned int *data) {
  int i;
  tqueue *que = (tqueue *)queue->user_object;
  /* Wait till queue is not full */
  while (OutputQueue_Full(queue)) {
    unsigned random = rand();
    que->full = random & 0x1;
```

```
  }

  if(reserve_only) /* Do not push data if reserve only */
    return;

  /* Push data */
  for (i = 0; i < get_num_words(queue->width); i++) {
    if (i != 0)
      fprintf(que->fp, "\t");
    fprintf(que->fp, "0x%x", data[i]);
  }
  fprintf(que->fp, "\n");

  /* Randomly set whether queue is full */
  que->full = rand() & 0x1;

}
```

Following is the application code `output_queue.c` running on the processor:

```
#include <stdio.h>
#include "oq.h"
#include "xtensa/tie/output_queue.h"

#define NUM 100

int main() {
  int i;
  int finished;
  unsigned data1, data2;
  tqueue *que;

  que = OutputQueue_oq_init();
  cstub_register_oq(que, &OutputQueue_Full, &OutputQueue);

  for (i = 0; i < NUM; i++) {
    data1 = rand();
    data2 = rand();
    finished = put_oq_nonblock(data2, data1);
    if (finished)
      printf("0x%x\t0x%x\n", data1, data2 & 0xff);
    data1 = rand();
    data2 = rand();
    put_oq_block(data2, data1);
    printf("0x%x\t0x%x\n", data1, data2 & 0xff);
  }

  OutputQueue_oq_free(que);
  return 0;
}
```

8.5.5 Lookup

A lookup allows TIE instructions to send requests to external devices and receive responses in an atomic manner. Detailed information about lookups is in the Lookup Sections chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. In the cstub environment, the data structure for a lookup interface is:

```
typedef struct cstub_Lookup_struct {
  void *user_object;  /* Lookup device */
  const char *name;   /* Name of lookup */
  unsigned out_width; /* Output width of lookup */
  unsigned in_width;  /* Input width of lookup */
} cstub_Lookup_t;
```

- `user_object` points to an external device that can be viewed as a lookup.
- `name` is the name of the `lookup` as declared in the TIE file.
- `out_width` is the output width of the `lookup` as declared in the TIE file.
- `in_width` is the input width of the `lookup` as declared in the TIE file.

The callback function for lookups take the following form:

```
typedef void (*cstub_Lookup_func_t)(const cstub_Lookup_t *data_struct,
                                    const unsigned *out_data,
                                    unsigned *in_data);
```

- `data_struct` is the lookup data structure described above. A pointer to this data structure is passed as an argument to the callback function.
- `out_data` is a pointer to the value of the lookup's output interface. The callback function should only retrieve n entries of the array pointed by `out_data`, where n is the smallest integer greater than or equal to (width + 31)/32. The first entry of the array maps to bit 31 to 0 of the output interface; the second entry maps to bit 63 to 32, and so on. The least significant bit of the first integer in the array corresponds to bit 0 of the output value. This mapping is independent of the endianess of the Xtensa processor.
- `in_data` is a pointer to the place holder of the lookup's input interface. The callback function must store the lookup's response to this address. The value to be saved must be an array of unsigned int type. The number of entries of the array must be equal to the smallest integer greater than or equal to (width + 31)/32, where width is the width of the Lookup. The first entry of the array maps to bit 31 to 0 of the input interface; the second entry maps to bit 63 to 32, and so on. The least significant bit of the first integer in the array corresponds to bit 0 of the input value. This mapping is independent of the endianess of the Xtensa processor. All the bits in the array that are outside the range of the lookup must be zero.

Because a lookup access is atomic, one callback function is provided for lookup interfaces. The processor waits until the result of the previous request is obtained before sending the next request. The `rdy` signal does not have any impact in native simulation.

Each lookup device data structure and the callback function must be registered before it can be used. The following function is provided by the cstub code to register the lookups:

```
extern void cstub_register_<lookup name>
    (void *user_object, cstub_Lookup_func_t callback);
```

- `lookup name` is the name of the lookup.
- `user_object` is a pointer to the object that represents the external device. It will be stored in the `user_object` field in `cstub_Lookup_t`.
- `callback` is a function pointer to the callback function.

The following example illustrates how the lookup is used in native simulation.

In this example, a 5KB memory is connected to a processor via a lookup. The memory takes a 10-bit address and returns a 40-bit value. For native simulation purposes, the data of the memory is read from a file. After the memory receives an address from the lookup output interface, it returns the data saved in that address via the lookup input interface. A program on the processor randomly generates addresses and passes them to the memory. It then prints out the address and the obtained data to the screen.

The TIE file `lookup.tie` is:

```
state lkup_res 40 add_read_write
lookup lkup {10, Mstage - 1} {40, Mstage} rdy
operation get_lkup {in AR addr}
                   {out lkup_Out, in lkup_In, out lkup_res} {
        assign lkup_Out = addr;
        assign lkup_res = lkup_In;
}
```

The C file `lookup.c`, which includes the callback function for the lookup device and the program running on the processor, is:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xtensa/tie/lookup.h"

#define NUM 1024

static unsigned * lkup_init() {
  FILE *fp;
```

```
  int i;
  unsigned *data = (unsigned *)calloc(NUM * 2, sizeof(unsigned));

  /* Obtain memory data from a file */
  fp = fopen("lkup.txt", "r");
  if (fp == NULL) {
    fprintf(stderr, "ERR: cannot open input file lkup.txt\n");
    exit(1);
  }
  for (i = 0; i < NUM * 2; i += 2) {
    fscanf(fp, "%x %x", &data[i+1], &data[i]);
  }
  fclose(fp);
  return data;
}

static void lkup_free(unsigned *data) {
  free(data);
}

/* Lookup callback function */
static void Lookup(const cstub_Lookup_t *lkup,
                   const unsigned int *lkup_Out,
                   unsigned int * lkup_In) {
  unsigned addr;
  unsigned *data = (unsigned *)lkup->user_object;

  /* Retrieve data */
  addr = lkup_Out[0] * 2;
  lkup_In[0] = data[addr];      /* First entry maps to bit 31 to 0 */
  lkup_In[1] = data[addr + 1] & 0xFF; /* Mask bits out of boundary */
}

int main() {
  int i;
  unsigned addr, data[2];
  unsigned *array;

  array = lkup_init();
  cstub_register_lkup(array, &Lookup);

  for (i = 0; i < NUM; i++) {
    addr = rand() & 0x3FF;
    get_lkup(addr);
    data[0] = Rlkup_res_0();
    data[1] = Rlkup_res_1();
    printf("Addr: %d\tData: 0x%x 0x%x\n", addr, data[1], data[0]);
  }
```

```
  lkup_free(array);
  return 0;
}
```

8.6 Simulation with Operator Overloading

The operator construct in TIE language maps TIE protos to C/C++ operators. When C++ cstub file is used to simulate the application containing operators for TIE types, those operators are automatically overloaded in the C++ cstub file, and simulation can be performed smoothly.

C/C++ operators for TIE types can also be mapped to C/C++ functions instead of protos. Detailed description of such mapping can be found in the *Xtensa C and C++ Compiler User's Guide*. Because cstub files are generated from TIE files, cstub files do not contain such information and simulating application utilizing this feature would fail. Modifications to the cstub files are required to perform the simulation. The detailed modification various from application to application, but the general steps are the same. Below is an example that illustrates the modification.

Suppose an application contains the following files:

A TIE file: `c_overload.tie`

```
regfile XR 32 16 xr

operation XR_ADD {out XR a, in XR b, in XR c} {} {
        assign a = b + c;
}
```

A header file declaring the overloading function: `xr_add.h`

```
extern XR __xt_operator_PLUS(const XR a, const XR b);
```

A C file that implements the overloading function: `xr_add.c`

```
#include "xtensa/tie/c_overload.h"

XR __xt_operator_PLUS(const XR a, const XR b) {
  return XR_ADD(a, b);
}
```

A C application file: `test.c`

```
#include <stdio.h>
#include "xtensa/tie/c_overload.h"

int main() {
```

```
  XR a, b, c, *p;
  unsigned int ina, inb, inc;
  ina = 1000;
  inb = 2000;
  a = *((XR *)&ina);
  b = *((XR *)&inb);
  c = a + b; /* operator overloading */
  inc = *((unsigned *) &c);
  printf("c = %d\n", inc);
}
```

Run TIE compiler to generate the cstub files:

```
tc -d tdk c_overload.tie -cstub
```

If we compile the application using cstub:

```
g++ -I. -Itdk/cstub test.c xr_add.c tdk/cstub/cstub-c_overload-*.cpp
```

We get the following errors:

```
test.c: In function `int main()':
test.c:19: error: no match for 'operator+' in 'a + b'
```

This is because the operator is overloaded via C/C++ functions, not via the TIE "operator" construct, and cstub is not aware of this overloading.

To make cstub aware of such overloading, the header file (`c_overload.h`) needs to be modified:

Modify the class that represent the XR ctype as follows:

```
namespace XR_space {
CSTUB_MSC_ALIGN(4) class XR_ {
public:
  unsigned int _[1];
  XR () {}
  inline XR_& operator= (const XR &src);
  template <typename T_> inline operator T_ const;
  inline XR_ operator += (const XR_ c); // added method
} CSTUB_GCC_ALIGN(4);
}
```

Add the following at the ctype conversion functions portion of the header file:

```
/* Ctype convertion functions */
#include "xr_add.h"

namespace XR_space {
```

```
inline XR_ operator+ (const XR_ a, const XR_ b) {
  return __xt_operator_PLUS(a, b);
}
inline XR_ XR_::operator+= (const XR_ c) {
  *this = *this + c;
  return *this;
}
}
```

Then the TIE "+" operator can be successfully picked up by the cstub code. Please note, when both TIE "operator" construct and C/C++ function overload the same C/C++ operator, the C/C++ function is used. Thus, the cstub file needs to be modified to match the behavior of the C/C++ function even though operator overloading functionality is already generated from the TIE operator construct.

8.7 Xtensa ISA Instruction Intrinsics

When developing application code for an Xtensa processor, you typically do not need to use intrinsics for Xtensa ISA instructions. The XCC compiler is able to compile your C/C++ code using an appropriate combination of these instructions to generate efficient and compact code. There may be some special situations in which you want to use a specific instruction intrinsic in your application code, and the XCC compiler supports such use. The intrinsic is simply mapped to the appropriate assembly instruction.

The cstub mechanism provides limited support for simulating Xtensa ISA instruction intrinsics on the x86 platform. A subset of the Xtensa ISA instructions are supported in the same way as TIE instructions - by providing C or C++ functions that mimic the behavior of the instructions. The native compiler views these intrinsics as function calls, and the compiled code works as expected.

The supported Xtensa ISA instructions include:

- Arithmetic instructions, including arithmetic instructions in density and boolean options
- The 16- and 32-bit multiply instructions are supported. However, if the multiply instructions are implemented as iterative multipliers, they are not supported. The MAC16 instructions are not supported.
- Miscellaneous option instructions
- Floating point coprocessor instructions
- HiFi2, HiFi EP, and HiFi3 audio engine coprocessor instructions
- Vectra2 coprocessor instructions
- ConnX Family DSP instructions
- Double-precision floating point assist instructions

To obtain the full list of Xtensa ISA instructions supported for your specific configuration, refer to the cstub header file generated by the TIE compiler in the TDK directory. Note that instructions relating to the micro-architectural implementation of the Xtensa processor are not supported in the cstub environment. Examples of such instructions are exception and interrupt instructions, memory protection and translation instructions, instructions that manipulate several special registers and the like. Further, branch instructions are also not supported.

When you compile your application code with the XCC compiler for an Xtensa ISS simulation, you may have to include multiple header files in your application. This is because the intrinsics are grouped into different header files based on whether it is a designer-defined TIE instruction, a core ISA instruction or an instruction belonging to one of the configuration options. However in cstub simulations, designer-defined TIE instructions, core instructions, and optional instructions are all declared in the same cstub header file. No separate header file is needed.

8.8 Pitfalls of Native Simulation

The Xtensa platform and x86 platform are fundamentally different. This section discusses some of the pitfalls of porting Xtensa applications to an x86 platform, so you can anticipate possible differences between a native simulation and a simulation of the same application code on the Xtensa platform.

- Implicitly defined macros. If you compile your application using the XCC compiler, several macros are implicitly defined, which may be used in the application. When you simulate the application natively, however, those macros are not defined implicitly. You may explicitly define them to achieve the same behavior. Examples of such macros are: __XTENSA_EB__ and __XTENSA_EL__ . Tensilica recommends not setting the __XTENSA__ and __XCC__ macros; this way the native compilation will be differentiated from the XCC compilation.
- Alignment convention. The native compiler you use may have a different alignment convention from the XCC compiler. In the XCC compiler, all arrays are aligned to the size of the array, up to the maximum memory interface width or 16 bytes, whichever is wider. If the datatype size is more than 16 bytes, the array is aligned to the size of the datatype. Dynamically allocated memory follows the same convention. In contrast, the GCC and Visual C++ align arrays on the array element size. On the x86 platform, 256 and 512 bit wide datatypes cannot be guaranteed to be aligned to the size of the datatype. If your application relies on XCC alignment conventions, you may get a different result in native simulation. If your application does not have unaligned memory access, which requires your TIE load instructions to automatically rotate the loaded value based on the address, Tensilica recommends selecting the "Take exception" configuration option in the XPG. This way, you will get a runtime failure with an explicit error message indicating the problem if your host compiler aligns data differently from the assumption of the TIE instructions. After you detect

the differences, you can force the compiler to align data as you expect. In addition, GCC -Os flag aligns the stack aggressively.Cstub cannot be compiled using the -Os flag.

- When using TIE Intrinsics in your application code, you must take extra precaution of the intrinsic arguments (which translate to instruction operands) that are of type `out` or `inout`. The type of the variable used for this argument should match the expected type of the TIE instruction; no casting is allowed on such variables. If there is a mismatch in the type of such variables, then depending on the specific characteristics of the mismatch, and depending on the native compiler used (C or C++) you may get either an error or a warning. It is usually possible to make minor modifications to your application source code to eliminate these errors and warnings. It is important that these warnings be eliminated, because you may get incorrect simulation results if you ignore them.
- Reserved keyword. The prefix `cstub_` is reserved for cstub code. However, this reserved prefix can be easily changed by modifying the following macro definition in the cstub header file:
  ```
  #define CSTUB_(X) cstub_##X
  ```
- Floating point behavior. If your application code has variables of type "float" or "double", and you perform arithmetic computations using standard C language operators, the behavior on the Xtensa platform and on the x86 platform will not be bit exact. The native compiler on the x86 platform will compile the code using the x86 floating point hardware, where 80-bit internal registers are used. The XCC compiler on the Xtensa platform will use the Xtensa floating point hardware (or software emulation library if the hardware does not have the floating point unit), where 64-bit precision is retained.
- When dereferencing a pointer to a ctype, the pointer must point to the beginning of the ctype. Dereferencing a pointer to the middle of a ctype is not supported.
- Simulation of applications running on big-endian Xtensa processors. Due to the endian difference between the target and host architectures, the obtained result may be different. This topic is discussed in detail in Section 8.4.
- If you use the C code generated by the TIE compiler for native simulation, some additional restrictions apply. These restrictions are discussed in Section 8.3 on page 120. Simulating with the C++ code is less restrictive, and thus recommended by Tensilica.
- The Xtensa platform is a 32-bit platform. If your x86 platform is a 64-bit platform, you need to compile both the cstub code and the application code under 32-bit mode.

Note that these differences are not specific to the Xtensa processor. It is common in the embedded world to port application code from one platform to another. Even when the code is written in a high level programming language such as C/C++, this porting may be a non-trivial amount of effort because of the differences in C compilers and the hardware implementation differences between the two processor platforms.

The cstub code generated by the TIE compiler is designed to provide the same functional behavior of the TIE instructions as on the Xtensa platform. This means that if a TIE instruction is simulated with the same input operand values on Xtensa ISS and on x86 using cstub code, you should see the same output value. However, it does not preserve any wire information inside the TIE instruction, nor does it contain valid information on the architectural registers or states of the Xtensa processor, such as the program counter or the stack pointer.

8.9 Compilers Supported in Native Simulation

Native simulation is supported for the following compilers:

- GNU gcc/g++ 4.4.x 4.5.x
- Microsoft Visual C++ 2012

Native simulation can also be compiled with the following compilers with restrictions:

- GNU gcc/g++ 4.6.x 4.7.x 4.8.x: compile with `-Wno-unused-but-set-variable` and `-fno-strict-aliasing`.
- Microsoft Visual C++ 2005 - 2010: may generate incorrect results with `/O2` due to some known bugs.

9. TIE Hardware

The previous chapters showed how TIE instructions can accelerate your applications and you learned how to specify extra functionality to extend the hardware of the Xtensa processor. But the focus was on the software performance.

This chapter discusses how to specify functionality for efficient hardware implementation. It explains the relationship between the TIE specification and the generated hardware. It also explains how you can optimize the generated hardware to limit the impact on Xtensa core speed and area. The last section of this chapter shows how to verify your TIE instructions.

9.1 Specifying TIE Hardware

The main part of each TIE instruction is the computation section. It is this TIE section that describes the hardware that implements the functionality of your instructions.

During hardware generation, computation descriptions are translated into synthesizable HDL descriptions.

9.1.1 Specifying Computational Sections of TIE Instructions

Previous chapters contain computational sections for different instructions. This section discusses the general syntax of this computation section.

Each line is a wire declaration or an assignment statement. See the Computation Section in the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a complete description on specifying computations in TIE.

Wire Declaration

Like Verilog, TIE is a hardware language that uses wires, whereas C programs use variables. Wire declarations introduce a new variable that can be used in an assignment statement. Following is an example of a wire declaration:

```
wire [31:0] myaddresult;
```

This declaration generates a 32-bit wire called `myaddresult`.

Verilog allows for fine definition for the size of the variables and their index (for example, `myaddresult` can be specified as [12:0]). The C language does not provide this fine granularity in its C types. Every bit results in hardware, so it is important to know the minimum size requirements for your variable, because this produces TIE instructions with minimum hardware.

Assignment Statement

Assignment statements contain all the computation that your TIE instruction performs. Following is the general syntax for an assignment statement:

```
assign Left-value = Operation;
```

Valid left-values are outputs of your TIE instructions or declared wires. It is not necessary to have a wire declared before using it in a left-value. You can have a left-value, which declares the wire, but in this case does not use the `assign` keyword. The `assign` keyword is optional for all other wire assignment statements.

Note that any assignment must assign a whole left value. It is illegal to write

```
assign myaddresult[15:0] = ........;
```

where `myaddresult` was declared as a 32-bit wire. But, it is legal to assign a different size wire to a left value. The right-hand side of the assignment will be 0 extended if it contains fewer bits. If it contains more bits, it will assign the lower bits to the left value. If you assign a 50-bit result to the 32-bit `myaddresult` wire, then the lower 32 bits of the result will be assigned to `myaddresult`. The TIE compiler can optionally generate warning messages when you have assignment width mismatches in your TIE code.

Operations can consist of TIE instruction inputs, operators and/or wires. TIE instruction inputs are states, register values, immediates, tables, or interfaces. Every assignment ends with a semicolon. Note that an assignment can cover multiple lines in the TIE file.

9.1.2 Common Mistakes when Specifying TIE Instructions

Sometimes when you simulate your code, which uses TIE instructions, you get simulation results that do not behave as you expect. This is especially true if you have not previously worked with the Verilog language, which is used to specify the TIE functionality.

This section describes some common mistakes a designer can make when specifying TIE instructions.

Specifying Signed Operations

The Verilog subset that is supported in TIE contains a set of operators as described in the Operators section in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. All wires and operators interpret the data as unsigned.

You can create a less-than instruction named `mylessthan` using the Verilog less-than operator (<). Assuming that this instruction operates on AR registers, calling this function in C as `mylessthan(-3, 8)` generates a false result. This occurs because `-3` is translated to 0xfffffffd and 8 becomes 0x00000008. For an unsigned less-than, 0xfffffffd is larger than 0x00000008.

The easiest way to specify signed operations is to use TIE modules. TIE modules are described in detail in the TIE Built-In Modules section in the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

Expression Widths

When writing TIE code, it is possible to perform operations on wires of different lengths. You can even make assignments in which the width of the LHS and RHS are not the same. When the RHS of an assignment is wider than the LHS, it is truncated to the width of the LHS. When the RHS is narrower, it is zero extended to the width of the LHS. Although this syntax is convenient, it can also hide errors. Consider the following example:

```
wire [31:0] tmp_a;
wire [127:0] tmp_b;
.....
.....
assign tmp_a = MemDataIn128;
```

In this example, the designer's intent was to assign the input interface MemDataIn128 to the 128-bit wire tmp_b. But by mistake it was assigned to the 32-bit wire tmp_a. This results in the upper 96 bits of MemDataIn128 being truncated and only the least significant 32 bits being assigned to tmp_a.

As another example, consider the signed addition of a 32-bit value A with a signed 4-bit immediate B. You may be tempted to write this as

```
assign R = A + B;
```

However, this generates the wrong result because for the addition operation, B is 0 extended to 32-bits, after which A is added to it. Zero extending B has changed the value from a signed to an unsigned 4-bit value. The correct way to describe this is to write:

```
assign R = A + {28{B[3]}, B};
```

This description prevents the above problem because the sign bit is replicated 28 times, and then concatenated with the value of B. Thus all operands of the expression are 32 bits wide and the correct result is obtained.

As illustrated in the preceding examples, the bit-width matching rules of Verilog can result in unintended consequences. To avoid such errors, you can have the TIE compiler generate warning messages for such assignment width mismatches in your TIE code. This is done either by enabling all warnings or by enabling warnings of type TIE_ASSIGNMENT_WIDTH_MISMATCH, TIE_EXPRESSION_WIDTH_MISMATCH, and TIE_EXPRESSION_OPERAND_MISMATCH.

Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a precise definition of expression and operand widths, and the rules according to which the TIE compiler will perform bit-width checking when it is enabled.

Endianness

Endianness can cause problems when you work on different vector sizes or when you create a TIE file that should work on little and big endian processors. A typical endian problem is if you load a 128-bit value into a TIE state called `Data`. Creating a TIE instruction that processes the first byte of this 128-bit value is Data[7:0] for a little-endian configuration and Data[127:120] for a big-endian configuration.

To avoid endianness problems use the TIE preprocessor as described in Section 9.1.3.

9.1.3 Using the TIE Preprocessor

The TIE language also supports the use of Perl as a preprocessor language. Using the TIE preprocessor makes it easy to generate a TIE file that supports little and big endian systems. It also simplifies creating SIMD instructions. All preprocessor lines start with a semicolon.

Perl Variables

You can generate Perl variables in a TIE file and use them to define global constants. For example, you can use a constant to indicate the number of data elements a SIMD instruction will work on.

See the TIE Preprocessor chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual* for a list of predefined variables.

Generating Conditional Sections

The TIE preprocessor can also generate conditional sections in your TIE file. For example, in a TIE file that can be used in big and little endian processors, you can create different TIE functionality based on the predefined variable `IsaMemoryOrder`, as follows:

```
; if ( $IsaMemoryOrder eq "LittleEndian") {
     .... little endian code
; } else {
     .... big endian code
; }
```

Using Loops for Replication

Another use of the preprocessor is to create repetitive structures. An example from the color conversion TIE is the `vec_MAC` instruction. It performs eight parallel multiply accumulates. Instead of writing the full computation section, you can use the TIE preprocessor to generate this structure, as follows:

```
operation vec_MAC {inout Acc Accum, in vec128 A, in Coeff C}{}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = Accum[%d:%d];\n", $i, (255-$i*32),
(224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("    wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};
}
```

In general, a TIE file with preprocessor statements is easier to read if you put the whole line into a `printf` statement.

The TIE compiler expands the implementation of the `vec_MAC` instruction into:

```
# tietpp line 140
operation vec_MAC {inout Acc Accum, in vec128 A, in Coeff C}{}{
    wire [31:0] S0 = Accum[255:224];
    wire [15:0] A0 = A[127:112];
    wire [17:0] C0 = C;
    wire [31:0] R0 = TIEmac(A0, C0, S0, 1'b1, 1'b0);
    wire [31:0] S1 = Accum[223:192];
    wire [15:0] A1 = A[111:96];
```

```
    wire [17:0] C1 = C;
    wire [31:0] R1 = TIEmac(A1, C1, S1, 1'b1, 1'b0);
    wire [31:0] S2 = Accum[191:160];
    wire [15:0] A2 = A[95:80];
    wire [17:0] C2 = C;
    wire [31:0] R2 = TIEmac(A2, C2, S2, 1'b1, 1'b0);
    wire [31:0] S3 = Accum[159:128];
    wire [15:0] A3 = A[79:64];
    wire [17:0] C3 = C;
    wire [31:0] R3 = TIEmac(A3, C3, S3, 1'b1, 1'b0);
    wire [31:0] S4 = Accum[127:96];
    wire [15:0] A4 = A[63:48];
    wire [17:0] C4 = C;
    wire [31:0] R4 = TIEmac(A4, C4, S4, 1'b1, 1'b0);
    wire [31:0] S5 = Accum[95:64];
    wire [15:0] A5 = A[47:32];
    wire [17:0] C5 = C;
    wire [31:0] R5 = TIEmac(A5, C5, S5, 1'b1, 1'b0);
    wire [31:0] S6 = Accum[63:32];
    wire [15:0] A6 = A[31:16];
    wire [17:0] C6 = C;
    wire [31:0] R6 = TIEmac(A6, C6, S6, 1'b1, 1'b0);
    wire [31:0] S7 = Accum[31:0];
    wire [15:0] A7 = A[15:0];
    wire [17:0] C7 = C;
    wire [31:0] R7 = TIEmac(A7, C7, S7, 1'b1, 1'b0);
# tietpp line 148

# tietpp line 149
     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};
# tietpp line 150
}
```

Debugging TIE Files that use the TIE Preprocessor

The TIE preprocessor outputs a `<tiefilename>.tppout` file when compiling your TIE file. The TIE compiler uses this file to create the processor extensions.

If there is an error, for example in the `vec_MAC` preprocessor loop from the previous section, the error is reported on the preprocessor output file. When using Xtensa Xplorer, this preprocessor output file is located in the `<workspace>/XtensaInfo/Models/<CoreName>.tdk` directory. Do not change this preprocessor file, because the next TIE compile will overwrite it. Use this file to find the errors caused by your preprocessor sections and then modify your original TIE file accordingly.

9.2 Obtaining Hardware Costs

Tensilica recommends running the TIE compiler often while writing your TIE instructions. Each time you run the TIE compiler, it generates an area estimate report for your TIE file. This report is a rough estimate based on the number of instructions, states, register files, and operators you have specified. Figure 9–23 shows the area report for the TIE file containing the FLIX instructions to accelerate the color conversion example.

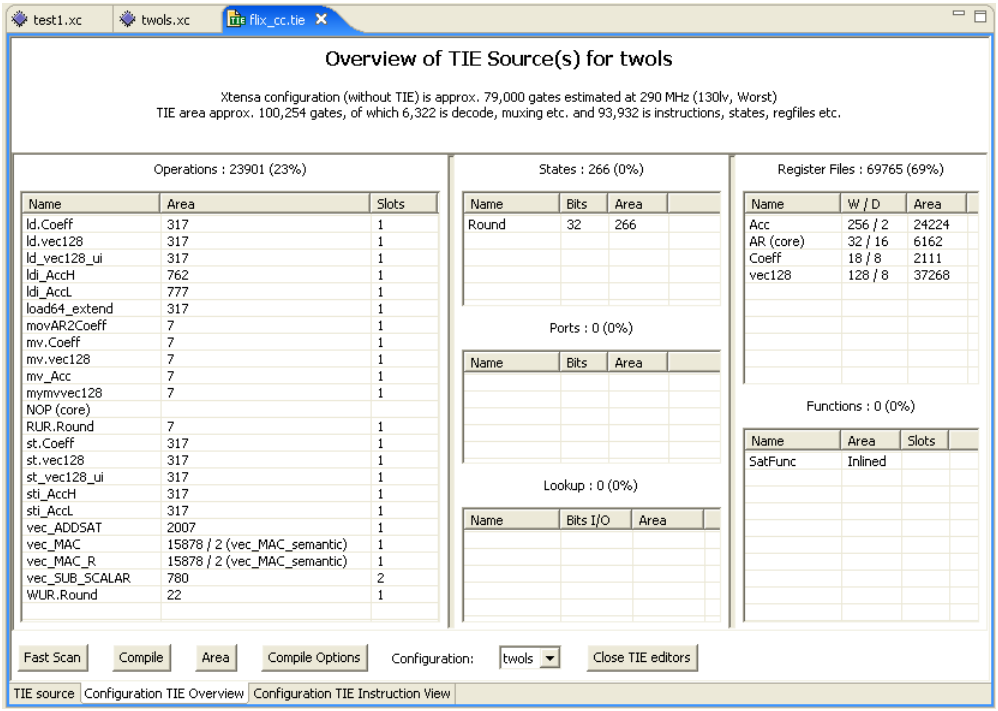

Figure 9–23. TIE Area Estimate

Although the generated TIE area report is a good guideline to use during development for making some area tradeoffs, it is only a rough estimate. To obtain a more accurate estimate, synthesize your TIE file. To synthesize your TIE file, run `perl run_syn.pl` from the directory where the TIE compiler generated its output.

Note that the most reliable data you can obtain is by synthesizing the Xtensa core plus TIE. When you synthesize the TIE file on its own, assumptions are made about the input and output delays. To prevent spending time optimizing paths that are not the critical paths in the optimized Xtensa processor, upload the configuration, build the configuration with TIE, and then run synthesis on the Xtensa Processor RTL including your TIE.

TIE synthesis uses the CadSetup file that is in the configuration directory for your library information. Refer to the *Xtensa Hardware User's Guide* for instructions to set up this file.

9.3 TIE and Xtensa Core Considerations

The previous section described how to obtain the hardware costs of adding TIE. The next section discusses how to optimize the results. This section explains the importance of TIE and Xtensa core interactions that can have an impact on the final results.

9.3.1 TIE Inputs

Each TIE instruction can use register operands, immediates, tables, states, and interfaces as its input. It is important to note that if a large number of TIE operations/semantics use the same values, it can cause a problem for place and route.

For example, consider a TIE design with 10 different semantics that use a 128-bit register file value as one of its inputs. Routing the register file values to all the semantics can decrease the maximum clock frequency of the Xtensa core due to the amount of wires that need to be placed.

Timing on the `MemDataIn*` interface signals is usually tight. To prevent `MemDataIn*` from becoming a timing problem, do not add more than one level of logic before latching the data.

9.3.2 TIE Outputs

TIE instructions generate register, state, or interface variables. When a large number of TIE operations/semantics write the same variable, unexpected timing paths can be caused. For example, in the simple case where there are four TIE operations/semantics that write an AR register, all four values have to be muxed together into one AR result bus. Therefore, two levels of muxes are needed to combine the outputs of these four blocks.

A special case of this is the VAddr interface signal. Every load or store instruction generates an address for the load/store. When all loads/stores are specified as operations, each of these will generate an address generator. Combining the output of these address generators can easily create a timing problem. To solve the timing problem, you can share the address generator over multiple load/store instructions. Section 9.5.3 contains details about this timing problem.

9.4 TIE Area Optimizations

After running synthesis on your TIE file and obtaining the area and performance of your TIE, it might be necessary to fine-tune one or both of these parameters. This section describes how to reduce the area and/or improve the performance.

The next sections focus on improving the area of your TIE description. This section uses the Xplorer Area Estimator to estimate the gate count of the TIE solution.

9.4.1 TIE Modules

Sometimes the area can be reduced by using TIE modules, especially if you add multiple values together. In this case, use `TIEaddn` or `TIEcsa` modules to reduce the area and also improve the timing of the generated hardware.

9.4.2 Controlling State Area Cost

The hardware required to implement a TIE state is dependent on the bit width of the state, but it also is heavily dependent on the schedule of all instructions writing it. Also, reading a state after the W stage can cause an increase in area cost of a state.

However, the common case is writing a state. If there is even one instruction that writes the state at the end of the execute cycle, then two pipeline registers are required besides the flip-flops needed to implement the state itself. Each of these pipeline registers is as wide as the state. In this case, one 32-bit state will result in 3x32 = 96 flip-flops. This area increase is not a problem for a few small states, but if wider and more states are used, then this could become significant.

To limit the state area, you must control the minimum pipeline stage in which all instructions write a state. If one instruction writes a state in the execute stage, then the size of the state increases about three times. If the earliest you write a state (of any instruction) is M, then the state area size is about two times (both these cases are for a 5-stage pipeline). Any state will have a minimum size when all instructions write it in the W stage.

If you have added the `add_read_write` keyword to the definition of the state, then the TIE compiler will add one or more WUR.<*statename*> and RUR.<*statename*> instructions. The TIE compiler specifies a schedule for the WUR instruction, which writes the state, as the earliest pipeline stage any of the other instructions write this state. Therefore, if all the specified instructions write the state in the memory stage (M) or later, then the WUR.<*statename*> instruction will write the state in M as well.

9.4.3 State and TIE Register File Sharing

States and TIE register files can be very expensive in terms of area. If you are creating accelerators for different algorithms, then reuse states or TIE register files for significant area savings.

To enable the sharing of states or register files it might be necessary to change the width of the state/register file to the maximum required width.[1]

Be careful in sharing states and register files. If a large number of instructions use the same state and/or register file, this could create problems for place and route. This problem is described in Section 9.5.

9.4.4 Using Shared Functions

Every `operation` section generates hardware, unless a semantic is available for this instruction (see Section 9.4.5 for a discussion about semantics). TIE allows you to specify expensive operators in TIE functions. Sharing these functions between different TIE instructions can save area.

There are two similar instructions in the color conversion example that use a fair amount of hardware, namely the `vec_MAC` and `vec_MAC_R` instructions. Both instructions use eight multiply accumulate operators. Therefore, the generated hardware will contain 16 multiply accumulate operators.

To reduce the hardware for both instructions, generate a shared function, SIMD_MAC, which performs the eight parallel multiply accumulates. Following is the modified TIE for these two instructions.

```
state Round 32 add_read_write

function [255:0] SIMD_MAC([255:0] Accum, [127:0] A, [17:0] C) shared{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = Accum[%d:%d];\n", $i, (255-$i*32),
                                                     (224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16),
                                                     (112-$i*16));
; printf("    wire [17:0] C%d = C;\n", $i);
;  printf("    wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
                                                   $i, $i, $i, $i);
;}

     assign SIMD_MAC= {R0, R1, R2, R3, R4, R5, R6, R7};
}
```

1. If a register file has a width larger than the Data Memory access width, then custom load/store instructions need to be written.

```
// We need a multiply instruction of 8 elements
operation vec_MAC {inout Acc Accum, in vec128 A, in Coeff C}{}{
     wire [255:0] Result = SIMD_MAC(Accum, A, C);
     assign Accum        = Result;
}

operation vec_MAC_R {out Acc Accum, in vec128 A, in Coeff C}{in Round}{
     wire [255:0] Accum_in = 8{Round};
     wire [255:0] Result = SIMD_MAC(Accum_in, A, C);
     assign Accum             = Result;
}
```

The preceding example shows that the `SIMD_MAC` function is defined first. By specifying the shared keyword before the body of the function, only one hardware implementation exists. The TIE compiler generates all logic necessary to let both the `vec_MAC` and `vec_MAC_R` instruction use the parallel multiply accumulate instructions. In the example implementation the `vec_MAC` and `vec_MAC_R` instructions are single cycle because a shared function is single cycle.

9.4.5 Writing Semantic Sections

Another method to reduce area is to use semantic sections. A semantic section describes hardware that implements a group of instructions. If a TIE instruction has an operation section and is defined in a semantic as well, then the semantic section is used by the Xtensa Instruction Set Simulator (ISS) and is used for RTL implementation.

Following are the differences between a semantic section and shared functions:

- A semantic section defines all logic necessary to implement a group of TIE instructions. A shared function only specifies a subpart of an instruction.
- All instructions that are implemented in a semantic should have the same schedule for all similar inputs and outputs. Instructions that share a function can use the shared function in different cycles.
- Semantics can use less area than shared functions. The area can be lower for semantics because the TIE compiler infers muxing for shared functions that may not be needed inside a semantic.
- When you specify an operation and a semantic section you can run formal verification between the two descriptions. This is not available when using only shared functions. See Section 9.6.1 for more information about formal verification.

It is important to note that a shared function can be used inside a semantic block.

Using the color conversion example, you can create a semantic section that implements both the vec_MAC and vec_MAC_R instructions. The only difference between the vec_-MAC and vec_MAC_R instruction is that the first instruction uses a part of the accumulator as its input, while the latter uses the value from the Round state. Therefore, it is only necessary to change the assignment of S0....S7 as shown in Section 9.4.4, Using Loops for Replication.

The assignment for S0 becomes

```
wire [31:0] S0 = (vec_MAC)?Accum[255:224]:Round;
```

Where vec_MAC is a conditional signal that is true if the vec_MAC instruction is executed, and false if another instruction is executed. Following is the complete description for the semantic block:

```
semantic vec_MAC_semantic {vec_MAC, vec_MAC_R}
{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = (vec_MAC)?Accum[%d:%d]:Round;\n", $i,
                                        (255-$i*32), (224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16),
                                                  (112-$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("    wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
                                                   $i, $i, $i, $i);
;}

     assign Accum   = {R0, R1, R2, R3, R4, R5, R6, R7};

}
```

9.4.6 Sharing Multiplier Hardware

The Xtensa ISA defines several multiply instructions. MUL16S and MUL16U perform 16x16 multiply operations, while MULL, MULUH and MULSH perform 32x32 multiply operations. If your algorithm requires a custom multiplier that is different from the above, it is easy to create multiply instructions in TIE. An Xtensa configuration may include one or more of the Xtensa ISA multiply instructions, and TIE code that includes some designer-defined multiply instructions. In this situation, there are two multiplier semantics in the hardware; one from the Xtensa core that implements the Xtensa ISA multiply instruction(s) and one in the TIE code which implements the designer-defined TIE instruction(s). Thus you may have two multipliers in hardware, even though only one can be used at a time.

In order to address the above problem, the Xtensa Processor Generator (XPG) provides an option to have the Xtensa ISA multiply instructions implemented in the TIE file. With this option, the instruction definition, the encoding and the reference or operation description is still as per the *Xtensa Instruction Set Architecture (ISA) Reference Manual* to preserve binary compatibility. However the designer supplies the instruction semantic in the TIE file. Now you have the option to implement a single multiply semantic in your TIE description, which implements both the designer-defined TIE multiply instructions and the Xtensa ISA multiply instructions.

The following example illustrates how to implement a custom 24x24 multiply in TIE and share the semantic with the 16x16 multiply instructions of the Xtensa ISA:

```
state m0 24 add_read_write
state m1 24 add_read_write
state ACCUM 48 add_read_write

operation MUL24 {} {in m0, in m1, out ACCUM} {
     assign ACCUM = TIEmul(m0, m1, 1'b1);
}

schedule mul24 {MUL24} {
     use m0 1; use m1 1; def ACCUM 2;
}

semantic mul_semantic {MUL16S, MUL16U, MUL24} {
     wire [23:0] tmp0 = TIEsel(MUL24, m0,
                               MUL16U, {8'b0, ars[15:0]},
                               MUL16S, {{8{ars[15]}}, ars[15:0]});

     wire [23:0] tmp1 = TIEsel(MUL24, m1,
                               MUL16U, {8'b0, art[15:0]},
                               MUL16S, {{8{art[15]}}, art[15:0]});

     wire [47:0] prod = TIEmul(tmp0, tmp1, 1'b1);
     assign ACCUM = prod;
     assign arr = prod[31:0];
}
```

The example illustrates a designer-defined multiply instruction, `MUL24`, which performs a 24x24 multiply. The TIE code includes the operation, schedule and semantic description for this multiply instruction. However, the semantic also implements the `MUL16S` and `MUL16U` Xtensa ISA instructions. Note that the TIE code only includes the semantic for Xtensa multiply instructions, this is because their definition comes from the Xtensa ISA description.

MUL16S performs a signed 16-bit multiply and MUL16U performs an unsigned 16-bit multiply. Both instructions operate on operands from the AR register file (operands ars and art) and the result is also written to the AR register file (operand arr). The semantic performs a sign or zero extension of the ars and art operands to make them 24-bits wide and then uses the 24-bit multiplier used by the MUL24 instruction to perform the 16-bit multiply operation.

In the design illustrated by this example, there is only one 24x24 multiplier in the hardware. If the shared semantic option was not chosen on the XPG, there would be two multipliers in hardware; one 24x24 multiplier to implement the MUL24 instruction and one 16x16 multiplier to implement the MUL16S and MUL16U instructions.

When selecting this option, you must pay attention to the multiplier schedules. In the above example, the MUL24 instruction reads its input operands in stage 1, and produces the result in stage 2. This schedule matches with the schedule of the Xtensa ISA multiply instructions (in which input operands ars and art are read in stage 1 and output operand arr is assigned in stage 2). If however the schedule of the MUL24 instruction is different, you need to match the Xtensa ISA multiply instruction to match it. For example, consider the following schedule for the MUL24 instruction:

```
schedule mul24 {MUL24} {
     use m0 2; use m1 2; def ACCUM 3;
}
```

It is not possible to have the above schedule for the MUL24 instruction, and have the "use 1, def 2" schedule for the MUL16U and MUL16S instructions while sharing the semantic. The multiply logic would need to be spread over cycles 1 and 2 for MUL16 instructions and over cycles 2 and 3 for the MUL24 instructions, which is not possible. The way to solve this problem is to change the schedule of the Xtensa ISA multiply instructions to match the schedule of the designer-defined TIE instruction. When you select the XPG option to provide your own implementation for the Xtensa ISA multiply instruction, you have the option to specify the use and def stages for these instructions as well. This is illustrated in Figure 9–24 where selecting **User Provided MUL Semantic** enables this feature, and allows you to choose the **Read Stage** and **Write Stage** options. You must ensure that the schedules for the Xtensa ISA multiply instructions and the designer-defined multiply instructions are matched. Not doing so will either result in a TIE compiler error or RTL code that will not synthesize according to expectation.

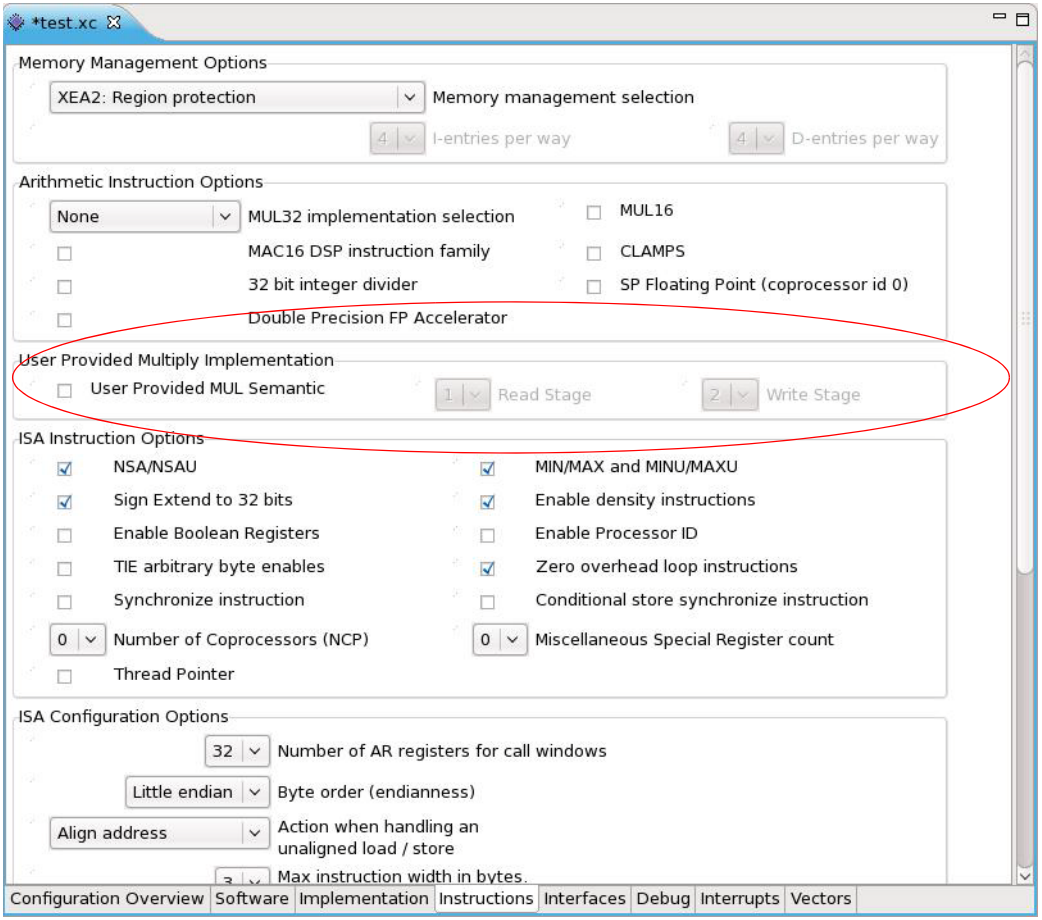

Figure 9–24. User Provided Multiply Implementation Option

If you select the option to provide your own semantic for the Xtensa ISA multiply instructions, please be aware that you are responsible for the correct implementation of those instructions. You are responsible for verifying the implementation to the same degree as you are for your own TIE instructions. The Xtensa Processor Generator does not verify that the multiply semantic correctly implements the Xtensa ISA multiply instructions. To assist you in writing this semantic, the TIE code below provides an example semantic that implements all the Xtensa ISA multiply instructions, with the exception of `MAC16` instructions. You cannot provide a semantic for the `MAC16` instructions in your TIE code, and thus you cannot share the `MAC16` multiplier with your TIE multiply instructions.

```
semantic mul_semantic {MULL, MULUH, MULSH, MUL16U, MUL16S} {
  // Decide whether this is a signed or unsigned multiply
  wire use_signed = MULSH || MUL16S;
```

```
    // Decide whether this is a 16x16 or 32x32 multiply
    wire m32 = MULL || MULUH || MULSH;

    // Select the upper 16 bits of the multiplicand
    wire [15:0] mulin0 = TIEsel(MUL16U, 16'b0,
                                MUL16S, {16{ars[15]}},
                                m32,    ars[31:16]);

    // Select the upper 16 bits of the multiplier
    wire [15:0] mulin1 = TIEsel(MUL16U, 16'b0,
                                MUL16S, {16{art[15]}},
                                m32,    art[31:16]);

    // Perform a 32x32 multiply to generate a 64-bit product.
    // The mulpp is scheduled in the first cycle, the add in the second.
    wire [63:0] p0, p1;
    assign {p0, p1} = TIEmulpp({mulin0, ars[15:0]}, {mulin1, art[15:0]},
                                use_signed, 1'b0);
    wire [63:0] product = TIEadd(p0, p1, 1'b0);

    // Assign the final 32-bit result based on which instruction
    // is being executed.
    assign arr = (MULL || MUL16U || MUL16S) ? product[31:0] :
                                              product[63:32];
}

schedule mul_schedule {MULL, MULUH, MULSH, MUL16U, MUL16S} {
  use ars Estage;       // Inputs used in stage Estage
  use art Estage;
  def p0  Estage;       // Register partial product outputs
  def p1  Estage;
  def arr Estage + 1;   // Final output is in Estage + 1;
}
```

The XPG feature that allows you to provide your own semantic for Xtensa ISA multiply instructions is disabled by default. If you want to use this feature, request your Tensilica support engineer to enable this feature for your account.

9.4.7 Tuning Your FLIX Specification For LX cores

Section 7.5 describes how to generate FLIX instructions to further accelerate the color conversion example. This section explains why specifying FLIX instructions increases area. With this knowledge, you can limit the area increase while receiving the large performance improvement that FLIX provides.

FLIX instructions increase the area due to:

- Duplication of instruction hardware
- An increase in register file and state area

Duplicating Instruction Hardware

Instruction hardware can be duplicated using FLIX instructions. A simple example is one FLIX instruction that can execute the same operation in multiple slots. The only way this is possible is by having multiple copies of the hardware available in different slots.

The Slot Indices of a Format section in the *Tensilica Instruction Extension (TIE) Language Reference Manual* provides a detailed description of how operation assignments to FLIX slots can lead to hardware duplication.

In the color conversion example, the area increase due to duplication is controlled by only duplicating instructions that use little hardware. The `vec_MAC` and `vec_MAC_R` instructions are limited to one slot, and only this slot, so that the eight multipliers were not duplicated. The `vec_SUB_SCALAR` instruction is the only duplicated instruction. It is duplicated because it is needed in Slot 0 and in Slot 3 to provide good software performance.

Increasing Register File and State Area

Register file area can also increase in a way similar to instruction hardware. When multiple slots in a format read from or write to a register file, then multiple read or write ports on the register file are needed. Each added write port significantly increases the register file area.

The TIE compiler analyzes the minimum required number of ports for the register file and tries to map multiple operands to the same port. If the TIE compiler can decide that two operands are never used at the same time, it may choose to map them to the same port. The TIE compiler uses the following rules to identify shared register operands:

- Operands used in one operation are not mapped to the same port.
- Operands used in different slots of the same format are not mapped to the same port (the same operand in the same slot index is mapped to the same port).
- If two operations contain the same register argument name and they share a semantic, their register arguments are mapped to the same port.

In the color conversion example, register Acc is written by ldi_AccL, ldi_AccH in Inst format with slot index 0, and `vec_MAC` and `vec_MAC_R` in `f64` format with slot index 1. Because operations ldi_AccL and ldi_AccH can never execute in parallel with operations `vec_MAC` and `vec_MAC_R`, the TIE compiler only generates one write port for register file Acc, (only in RC2009.0 and later releases).

A description of the number of write ports and read ports is given in the TIE Compile Report file, which is located in your TDK directory (`<tiefilename>.report`). It is important to check this file, especially if you have wide register files.

It is important to note that states also experience the problem of increased area due to FLIX. An advantage of states is that states are usually much more dedicated to a small number of instructions, and therefore fewer states are used than registers. This is against the significant disadvantage of not having the flexibility and ease of programming that register files provide.

9.5 TIE Timing Optimizations

After running synthesis, you may find that your TIE instructions are preventing the core from running at the required clock frequency. This section discusses how to find the timing problems and improve them.

9.5.1 Understanding the Synthesis Timing Report

The TIE compiler compiles your TIE description. One of the outputs of this process is the HDL description of the processor extension. When you then synthesize the TIE design or the whole processor, plus TIE, timing violations may occur. At times it may be hard to relate this timing report back to where in your TIE description this path is because synthesis tools lose hierarchy information during optimizations steps.

This section provides examples of translating a TIE feature to an HDL name where the TIE feature is the input and/or output of a timing path. The following names are examples and might differ, depending on your synthesis tool.

Core and TIE Register Files

Register files always have the name of the register file and `Regfile` in the cell name. An example of this is the beginning of the critical path into the `Acc` register file, as follows:

```
Startpoint: DRam1Data0[31]
              (input port clocked by CLK)
  Endpoint:
TIE_TIE_Acc_Regfile_Xm_base128_2LD_FLIXTIE_Acc_Regfile_bypass_TIE_Acc_
Regfile_bank0_ird3_data_C1_i0_tmp_reg_16_
            (rising edge-triggered flip-flop clocked by CLK)
```

The input is a port on the Xtensa processor that contains the data coming from the dataRAM. The destination is the `Acc` register file. The only two instructions that fit this description are the `ldi_AccL` and `ldi_AccH` instructions.

To be exact, the data is forwarded to read port 3 because the signal name contains bypass in the name. This means the data is forwarded to an execution unit. The destination is known because rd3 says it is for read port 3. If a register file is written, then the naming would be wr3, for writes port 3. The operands mapped to a read or write port can be found in the report file generated by the TIE compiler.

The same naming convention is used for Xtensa core register files, such as the AR register file. In this case, the name is AR_Regfile.

State Timing Paths

State timing paths are similar to register files. Instead of using Regfile in the name, it contains State. Another difference is that state cell names do not contain slot and read port information. An example of a state timing path follows.

```
Startpoint:
TIE_TIE_Round_State_Xm_base128_2LD_FLIXTIE_Round_State_bypass_TIE_Roun
d_State_bank0_icore_iword0_tmp_reg_7_
              (rising edge-triggered flip-flop clocked by CLK)
  Endpoint:
TIE_TIE_Acc_Regfile_Xm_base128_2LD_FLIXTIE_Acc_Regfile_bypass_TIE_Acc_
Regfile_bank0_ird0_data_C1_i0_tmp_reg_63_
            (rising edge-triggered flip-flop clocked by CLK)
```

The timing path starts with the state register Round, which is used in the vec_MAC_semantic semantic. The output is the Acc register file described in the previous section. Notice that now the result is forwarded to rd0. This path is through the vec_MAC_semantic and is the vec_MAC_R instruction because this is the only instruction that would read the Round state and write an Acc register. But in this case, there is more information inside the timing path, as it contains the following line

```
TIE_TIE_slot1_semantic_vec_MAC_semantic_iTIE_tmp6_C2_i184_i/S
```

In most cases these hierarchical names are lost in the final synthesis step, when the core is flattened and then synthesized again.

Multi-Cycle TIE Timing Paths

Multi-cycle TIE instructions are implemented by specifying a schedule for its inputs and outputs. The instruction is a multi-cycle instruction when there is at least one path from an input to output that takes more than one cycle. Each input that is part of a multi-cycle path will contain the required pipeline register(s). Retiming is then used by the synthesis tool to create a balanced pipeline by moving the pipeline register(s) into the multi-cycle path. Scheduling intermediate TIE wires can also create multi-cycle instructions. Following is an example of a path to a retimed register:

```
Startpoint:
TIE_TIE_vec128_Regfile_Xm_base128_2LD_FLIXTIE_vec128_Regfile_bypass_TI
E_vec128_Regfile_bank0_ird1_data_C1_i0_tmp_reg_17_
              (rising edge-triggered flip-flop clocked by CLK)
  Endpoint:
TIE_TIE_slot1_semantic_vec_MAC_semantic_Xm_base128_2LD_FLIXTIE_semanti
c_vec_MAC_semantic_REG376_S1
            (rising edge-triggered flip-flop clocked by CLK)
```

The input of the path is a bypassed value of the `vec128` register file. The destination is a pipeline register that is inside the `vec_MAC_semantic`. If you have a critical path from a pipeline register to another pipeline register, you can use the cycle information of the cell name as an indication. Note that the hierarchical cell name is determined by the synthesis tool and can change from tool to tool (or even between versions of the same tool).

All TIE cycle indications contain a C followed by a number inside the hierarchical name. Thus, the input of the path has a C1 full cell name. This indicates that it is in execution cycle 1. Also, the output has a cycle indication, which is generated by the synthesis re-timing tool. In this case it contains an S followed by the cycle number. Therefore, the endpoint of the timing path is at the end of cycle 1 (the full name contains S1), which makes sense as the beginning was also in cycle 1.

TIE Ports and Queues Timing Paths

TIE ports and queues produce timing paths that are similar to timing paths to or from states. A TIE output port `QoutAddr` produces the following timing path:

```
Startpoint:
TIE_TIE_QoutAddr_State_Xm_base128TQueueTIE_QoutAddr_State_bypass_TIE_Q
outAddr_State_bank0_icore_iword0_tmp_reg_0_
              (rising edge-triggered flip-flop clocked by CLK)
  Endpoint: TIE_QoutAddr[0]
            (output port clocked by CLK)
```

This path is from a state `QoutAddr` to the Xtensa port belonging to this state. The advantage of all TIE queues and ports is that these are signals that are going into or leaving the Xtensa processor.

In an example TIE file, there is an instruction that takes the input of an input queue `Qin`, and assigns it to a TIE register file named `QR`. Following is the timing path for this instruction:

```
Startpoint: TIE_TIE_Qin_inbuf_staging_data_tmp_reg_108_
              (rising edge-triggered flip-flop clocked by CLK)
```

```
  Endpoint:
TIE_TIE_QR_Regfile_Xm_base128TQueueTIE_QR_Regfile_bypass_TIE_QR_Regfil
e_bank0_iwr0_result_C3_i0_tmp_reg_108_
            (rising edge-triggered flip-flop clocked by CLK)
```

Remember that TIE ports and queues are buffered. Thus, the value from the outside queues are put in a register named `TIE_<TIE queue_name>_inbuf_staging_ data`. This value is then used by the above mentioned instruction as the input.

9.5.2 Optimizing TIE Hardware Timing

You can optimize any path violations in the TIE files. The following methods improve your TIE file's timing.

TIE Modules

All TIE modules use an efficient implementation of different functions. For a list of all TIE modules, see the TIE Modules chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual*. Replacing Verilog operators with TIE modules improves timing.

In general, Tensilica recommends using TIEmux, TIEsel, or TIEpsel instead of the Verilog conditional select operator "?". This is because using the conditional select operator can easily lead to unbalanced condition trees, which cause timing problems. In these cases, use one of the TIE mux modules to improve timing.

Semantics

Semantics can improve timing by reducing the number of hardware blocks that are in your design. For example, there are four TIE instructions that write to a register file. Muxes are needed to multiplex these four outputs to one write port (this example does not use FLIX). By creating a semantic that executes these four instructions, all of these muxes come inside the TIE design. Often, these muxes are not necessary, or fewer of them are needed.

The downside of a semantic block is its strength. Multiple instructions are combined in one hardware description, which can mean that muxes are needed on the inputs to select which input to use. For example, the `vec_MAC_semantic` in Section 9.4.5 needs a multiplexor to decide if the `Round` state or the `Acc` is used as the accumulator. When implementing a semantic, take care that this input and output muxing does not cause a timing problem.

Multi-Cycle TIE Instructions

Frequently, the timing paths are caused by too much logic in one processor cycle; in which case, the previous methods may not provide the improvement you need. In these cases, it is necessary to increase the number of clock cycles needed to execute the instruction.

The color conversion example uses an 18x18 bit multiply-accumulate. Trying to execute this instruction in one cycle would create the longest timing path in the design. And using a TIEmac TIE module does not solve the problem. There is just too much logic for one cycle.

The solution is to make this a two-cycle instruction. To do so, add the following schedule to the `vec_MAC` instruction:

```
schedule vec_MAC_schedule {vec_MAC}
{
     use Accum 2; def Accum 2;
}
```

The `vec_MAC_R` instruction uses a similar schedule, only it uses `Round` in cycle 2, instead of `Accum`. This instruction can also be made multi-cycle by specifying a cycle for a TIE wire in the schedule as discussed in Section 4.1.4.

Although the timing of the hardware design will significantly improve using multi-cycle instructions, it can reduce the performance of the software. The results of the multiply accumulate instructions are now available at the end of the second execution cycle. Therefore, any instruction wanting to use the result of a previous multiply-accumulate instruction is delayed for one cycle because the result is not yet available (see Section 7.3).

9.5.3 Common Timing Problems

The previous section shows how to find the part of your TIE description that is causing a timing problem and offers techniques to optimize the timing. This section discusses a couple of common timing problems in TIE and how to optimize them using one or more of the techniques discussed in the previous section. The end goal is to have an optimized TIE description that provides maximum software performance using minimum hardware, which runs at your required processor clock speed.

Memory Load Data Timing Problems

Frequently, one of the main initial timing problems is a path from memory to a register file or state. This path is caused by too much user-specified logic in the memory path before it is registered.

A mux on the input data is probably acceptable. The reason that this timing path occurs is because the memory data goes through multiple levels of logic in the Load/Store Unit before it reaches the TIE operation/semantic block. This, plus the fact that the memory data is only available late in the clock cycle, makes this a very timing sensitive path.

If you want to perform some simple operations on memory data, run synthesis on the Xtensa core with TIE. But even if synthesis meets timing you could have timing problems after place and route. It is preferable to use the memory timing of the physical memories you will be using.

If you need extra logic after the load, but cannot fit it in one cycle, make the result of the load available at the end of the W stage. To achieve this, add a schedule for your load instruction and specify `def Wstage` for the instruction output. This method generates instructions that load data, and then performs a computation.

Memory Address Timing Problems

The color conversion TIE example has multiple TIE instructions that perform a load. Each instruction generates the address for the load and assigns it to VAddr. When the TIE file is compiled, all VAddrs need to be muxed together to one VAddr, which goes into the load/store unit.

Thus, if you see timing paths throughout your TIE that end up at an address port of the Xtensa processor, try to reduce the number of hardware blocks that write to VAddr by combining multiple load instructions into one semantic. This reduces the amount of muxes needed on VAddr, and it also reduces the area because only one adder is used to generate the address per semantic.

Large TIE Files

Large TIE files refer to the area of the resulting hardware implementation. A TIE file can be large for different reasons. There may be too many instructions, or too many state and register files, or complex operations per instructions. A large TIE file does not imply a timing problem, but take care when designing an Xtensa processor with a large TIE file.

When you plan to create many TIE instructions, frequently run a synthesis run on all TIE instructions. When multiple designers create TIE instructions independent of each other, a mistake can easily be made. Each of the TIE files can meet timing individually, but when combined the TIE files can have a timing problem. It is important to do synthesis often and early providing designers with early feedback.

A related problem is when a large number of state and register bits are used by multiple instructions. If all these instructions access these resources, then a problem is created for the Place and Route tools because so many wires are needed to link all the resourc-

es to all the instruction execution blocks. If possible, reduce the amount of instructions that access a resource by creating duplicate resources. This reduces the number of wires from a resource, which reduces the place and route problem.

If you are having timing problems due to complex instructions that perform a large number of computations, use the optimization techniques in this chapter. In particular, making complex instructions use multiple cycles will solve the timing problem.

9.6 TIE Hardware Verification

Section 2.1 provides an overview of the TIE design methodology. In the function description phase, TIE is described without regards to timing and area goals. In this phase, TIE functionality is verified in the software environment with the Instruction Set Simulator. However, functional errors can be introduced in later design phases. TIE hardware verification is necessary to find subtle errors that may not have been detected using the software tools. The following sections describe the TIE hardware verification features.

9.6.1 Formal Verification of TIE Instructions

The TIE operation construct describes the functionality of an operation without concern for implementation efficiency. There is one operation section for every instruction. Use the semantic construct for hardware-optimized versions of an operation. The semantic of an operation describes the actual implementation of the data path logic and typically describes the logic for several operations that share some hardware. However, if no semantic is provided for an operation, the TIE compiler uses the operation as the semantic for implementation. Tensilica recommends writing operation descriptions first to verify the functionality. When operation descriptions are complete, create hardware-optimized versions using semantic descriptions. Tensilica recommends running a formal equivalence check between the operation and semantic descriptions to ensure that the two implementations of the same instruction are equivalent.

The TIE compiler automatically generates formal verification scripts to verify the operation description of an instruction with its semantic description. These scripts are for the Incisive Conformal formal verification tool from Cadence Design Systems. Although the scripts are automatically generated, users should understand the Incisive Conformal tool to interpret formal verification results.

When you run the TIE Compiler on a TIE file, it generates a TDK, which has the following files for formal verification in a directory named `verify`.

- `src`: containing the netlists `verify_ref.v` and `verify_sem.v` for the operation and semantic description respectively.

- `verplex`: contains a Makefile and scripts for running the Incisive Conformal tool to verify each instruction.

The TIE Compiler creates a `do` file (an Incisive Conformal script that sets constraints on both operation and semantic descriptions) for each operation. If FLIX is used, an operation will have a `do` file for each slot to which the operation is assigned. The name of the operation is prefixed with the format and slot name for which the operation is assigned. For example, following is the Incisive Conformal script for the `ld_Coeff` operation:

```
l24_x24_Inst_ld_Coeff.do
```

The prefix, `l24_x24_Inst`, is given to all 24-bit instructions. The prefix for FLIX instructions is somewhat different, as follows:

```
l64_f64_slot_mac_vec_MAC_R.do
```

The prefix `l64_f64_slot` indicates that this operation is part of a 64-bit FLIX instruction. The `vec_MAC_R` operation is designated to the `slot_mac` slot belonging to the `f64` instruction format.

To run equivalency checking, set your environment and path appropriately for the Incisive Conformal tool, then change the directory to <*`tdk_directory`*>`/verify/verplex` and run `make`.

If the operation and semantic based designs are formally equivalent, a message similar to the following appears on the console:

```
l64_f64_slot_mac_vec_MAC_R : The designs are formally equivalent
```

If the designs are not formally equivalent, the script also writes out the test vector(s) for all mismatched points in the design. A post-processor script is run on the report to translate the signal names in the netlist to signal names in the user TIE description. Consider the following example, in which a single semantic section implements an ADD and a SUB instruction by sharing the `TIEadd` module. To demonstrate the post-processor, an error is intentionally added.

```
// operation section
operation MADD {out AR outR, in AR inpR1, in AR inpR2} {}
{
assign outR = inpR1 + inpR2 ;
}
operation MSUB {out AR outR, in AR inpR1, in AR inpR2} {}
{
assign outR = inpR1 - inpR2;
}
//semantic section
semantic add_sub { MADD, MSUB }
```

```
{
wire [31:0] tmp = (MSUB) ? ~inpR2 : inpR2;
assign outR = TIEadd(inpR1,tmp,~MSUB); /* Intentional Error : should
be MSUB
*/
}
```

Run the equivalence check for the above example by running `make` in the `verplex` directory. The following message appears:

```
============= Report Summary =============
l24_x24_Inst_MADD : The designs are not formally equivalent
Instruction l24_x24_Inst_MADD, the designs are not formally
equivalent.
See logfile l24_x24_Inst_MADD.vector.do.log for mismatched data points
and
test vectors

l24_x24_Inst_MSUB : The designs are not formally equivalent
Instruction l24_x24_Inst_MSUB, the designs are not formally
equivalent.
See logfile l24_x24_Inst_MSUB.vector.do.log for mismatched data points
and
test vectors
```

This message indicates that the reference and the semantic are not equivalent for the instructions MADD and MSUB. The log file lists the data points that are not equivalent and lists the input test vector for each mismatched point. In the following report, the "Golden(G)" design refers to the reference and the "Revised(R)" design refers to the semantic. For more information about the tool, refer to the product documentation for Cadence Design Systems' Incisive Conformal.

Following is an excerpt from the log file:

```
LEC> report test vector outR[0]

Functional test vector for mapped points:
  (G) + 7889 PO   outR[0]
  (R) + 7522 PO   outR[0]

  (G) + 549  PI  inpR2[0] = 0
  (R) + 549  PI  inpR2[0] = 0

  (G) + 581  PI  inpR1[0] = 1
  (R) + 581  PI  inpR1[0] = 1


Simulation result:
  (G) + 7889 PO  outR[0] = 1
  (R) + 7522 PO  outR[0] = 0
```

The simulation result shows that the golden (operation-defined) and revised (semantic-defined) designs have different results. The tool points out that `outR[0]` has a mismatch. When the inputs have the `inpR1[0] = 1'b1` and `inpR2[0] = 1'b0` values, the bit `outR[0]` is 1 in the golden design and 0 in the revised design. This occurs because an error was intentionally introduced in the semantic.

10. Working with TIE Files and TDBs

Prior to the RB-2008.3 release of the TIE compiler, you were allowed to compile only one TIE file at a time and to attach only one TIE file to an Xtensa processor configuration. Since the RB-2008.3 release you can compile multiple TIE files together to create a single TDK directory. This allows for modular code development, and allows you to aggregate independently developed TIE code into a single TDK.

You are also allowed to compile one or more TDB files into a new TDK. A TDB file is generated by the TIE compiler as part of the TDK flow and is the file that gets uploaded to the processor generator when you build an Xtensa processor configuration with your TIE. This feature allows you to carry forward a TDB from one design project to another. It also allows you to treat the TDB as a library file, wherein TIE files for different aspects of your design can be created and independently compiled into TDBs, and then combined together during the integration stage of the project.

Both the Xtensa Xplorer environment and the command line interface to the TIE compiler allow you to compile any mix of TIE and TDB files together into a TDK.

10.1 Working with Multiple TIE Files

The TIE compiler allows you to compile multiple TIE files together to create a single TDK directory. These files can be independent TIE files such that each is a complete TIE file in itself, or they can be dependent TIE files. Dependent TIE files are files that together form a complete TIE description but any one of them by itself is not a valid TIE description. The primary purpose of breaking up the TIE extensions into multiple TIE files is to enable modular code development. Thus as an example, you can have one engineer or group create a TIE file for image processing and another engineer or group create a TIE file for internet protocol processing, and then combine them together after the individual development is complete.

10.1.1 Specifying a Base Name for Compilation

When compiling a single TIE file into a TDK, the name of the TIE file is used as the base name of several of the files inside the TDK directory. For example, if you compile the file `foo.tie`, then the TDK directory will contain the file `foo.v` (the Verilog implementation of your TIE instructions) and the file `foo.tdb` (the TDB file that is uploaded to the processor generator when you build your Xtensa configuration). When compiling multiple TIE files into a TDK, it is not obvious as to what base name should be used for the files in the TDK directory. You are thus required to provide this name explicitly as follows:

```
tc -d tdk -name bar aaa.tie bbb.tie ccc.tie
```

The preceding command will compile three TIE files, `aaa.tie`, `bbb.tie` and `ccc.tie` together into a single TDK directory. Files in the TDK directory will have the base name `bar`, as specified with the `-name` argument. Thus this TDK will have the file `bar.v` (which has the Verilog hardware implementation for all the instructions in the three input TIE files), and the file `bar.tdb` (which will be uploaded to the processor generator when building the Xtensa configuration with TIE).

The `-name` argument is mandatory when compiling more than one TIE file. You can use the `-name` argument when compiling a single TIE file to select a base name that is different from the name of the TIE file. However with a single TIE file, this argument is optional.

When working in the Xtensa Xplorer environment, you can specify the base name on the **TIE Compiler Options Page** in the box labeled **TDB Name**. If working with a single TIE file, the default value of this field is the name of your TIE file, but can be changed to any other name that you want. When you attach multiple TIE or TDB files to a configuration in Xplorer, you are automatically prompted to provide this name in the **Add TIE and TDB Files to a Configuration** dialog box.

10.1.2 Header File Names with Multiple TIE Files

When you compile a TIE file into a TDK, a header file is created by the TIE compiler for use in C/C++ programs. When compiling a single TIE file, the default base name of this header file is the same as the base name of the TIE file. Thus compiling the file `foo.tie` will generate a header file `foo.h`, which needs to be included in any C/C++ program file that uses intrinsics for any instructions defined in `foo.tie`.

When compiling multiple TIE files into a TDK directory, the default behavior is to create a single header file that includes the instructions from all the input TIE files. This header file has the base name specified with the `-name` argument as described in Section 10.1.1. For example, the command:

```
tc -d tdk -name bar aaa.tie bbb.tie ccc.tie
```

creates a single header file named `bar.h`, which includes intrinsics for instructions in all the three input TIE files.

It is possible to create separate header files for each of the input TIE files. This is done by using the `-unique_headers` command line argument as follows:

```
tc -d tdk -name bar -unique_headers aaa.tie bbb.tie ccc.tie
```

With this command, three separate header files are created inside the TDK directory. The header file `aaa.h` will only include intrinsics for instructions defined in file `aaa.tie`, file `bbb.h` will only include intrinsics for instructions defined in file `bbb.tie`, and file `ccc.h` will only include intrinsics for instructions defined in file `ccc.tie`. If all the intrinsics used by a particular C/C++ program are from TIE file `aaa.tie`, this program need only include the header file `aaa.h`.

In the Xtensa Xplorer environment, the **TIE Compiler Options Page** provides a check box for generating unique header files corresponding to each input TIE file.

Note that the generation of single or multiple header files is an option that only applies to the TIE files that you compile. When you import a TDB file, the header files are imported from the TDB without any change. This is explained in detail in Section 10.2.4 on page 181.

10.1.3 Dependent TIE Files

When compiling multiple TIE files, each TIE file can be an independent TIE file, or one or more TIE files may be dependent on others. Dependent TIE files are files that by themselves are not a legal TIE description. For example, you can have one TIE file define all the `operation` definitions of instructions (perhaps developed by the system architect or algorithm designer) and another TIE file define the `semantic` implementation of the instructions (perhaps developed by a hardware designer). The TIE file with the semantic descriptions is a dependent TIE file because it depends on the operation definitions in the other TIE file, and thus cannot be compiled independently.

When compiling dependent TIE files, the order of TIE files specified in the command line is important. All TIE language description sections must be written in a *define-before-use* order; thus forward references are not allowed. The order in which TIE files are listed on the command line must follow this rule, such that a file specified earlier in the listing cannot depend on one specified later. For example, if the operation description of an instruction is specified in file `oper.tie` and the semantic description is specified in file `sem.tie`, the following command is legal:

```
tc -d tdk -name bar oper.tie sem.tie
```

but the following command is illegal:

```
tc -d tdk -name bar sem.tie oper.tie
```

The operation description of an instruction must be specified before its semantic description, and thus `oper.tie` must be listed before `sem.tie`.

This restriction also implies that cyclical dependencies between two files are not allowed. Such dependencies are rare, but can happen — for example, if one file specifies the `opcode` and `reference` description of an instruction and the other specifies the `iclass` description.

In the Xtensa Xplorer environment (with respect to the preceding example), you would ensure that the file `oper.tie` is listed above the file `sem.tie` in the **Configuration Overview** page. If the order is incorrect, you can use the **up arrow** and **down arrow** icons to reorder the TIE files and place them in the correct order.

10.2 Working with TDB Files

A TDB file is generated by the TIE compiler as part of the TDK flow, and is the file that gets uploaded to the processor generator when you build an Xtensa processor configuration with TIE. A TDB is a binary file that contains your original TIE description, and other information generated by the TIE compiler during the compilation process. In particular, it contains all the field and opcode encodings that were generated by the TIE compiler for all the TIE instructions and their operands.

The field and opcode encodings are generated using a heuristics-based algorithm, and they are likely to be different from one run of the TIE compiler to another. This is especially true if the Xtensa configuration with which the TIE is compiled is different, or if you compare the encodings generated from two different releases of the TIE compiler. Thus if you compile the same TIE file on two different occasions, there is no guarantee that the encodings generated in both instances will be the same.

The ability to provide a TDB file as an input to the TIE compiler allows you to carry forward a TDB from one design project to another. It also allows you to treat the TDB as a library file, wherein TIE files for different aspects of your design can be created and independently compiled into TDBs, and then combined together at the end during the integration stage of the project.

To create a TDK with a TDB, simply use the TDB file as an input to the TIE compiler as follows:

```
tc -d tdk foo.tdb
```

This command creates a new TDK with all the TIE instructions from `foo.tdb` imported along with their field and opcode encodings.

In the Xtensa Xplorer environment, you can attach a TDB file to an Xtensa configuration by clicking on the **Attach TIE** button in the **Configuration Overview**.

10.2.1 Preserving Instruction Encodings Across Projects

There may be situations in which you want to use the same TIE file on two separate projects, and have the same field and opcode encodings be used on both designs to preserve object code compatibility between them. For example, this may happen when you port the TIE code from one generation of the product to the next, or want two similar (but not exactly the same) Xtensa configurations to have the same TIE extensions. This can be achieved through the following steps in the Xtensa Xplorer environment:

1. Build and install the first Xtensa configuration `xtcfg1`.
2. Attach TIE file `foo.tie` to configuration `xtcfg1` in the **Configuration Overview** window. Set the **TDB Name** to be `xtcfg1` on the **TIE Compiler Options Page**.
3. Click the **Compile TDK** button in the **Configuration Overview** window to build the TDK. When asked if you would like to make the TDB available as part of your TIE sources, select Yes. This creates the TDB file `xtcfg1.tdb`, which is available as part of your TIE source files.
4. Click the **Upload/Build** button to upload the TIE description to the processor generator, and build configuration `xtcfg1` with the TIE.
5. Build and install the second Xtensa configuration `xtcfg2`.
6. Click the **Attach TIE** button in the **Configuration Overview** of `xtcfg2`, and select the file `xtcfg1.tdb` in the dialog box that pops up. Select the option to import the TDB with opcode encodings preserved. Provide a name, `xtcfg2`, for the new TDB to be generated.
7. Click the **Compile TDK** button in the **Configuration Overview** window of configuration `xtcfg2` to build the second TDK.
8. Click the **Upload/Build** button to upload the TIE description to the processor generator, and build configuration `xtcfg2` with the TIE.

The preceding steps will result in two Xtensa configurations, `xtcfg1` and `xtcfg2`. These two configurations have the same TIE extensions (described in the file `foo.tie`) and the same encodings for all the instructions.

Note that in step 6, the TDB file `xtcfg1.tdb` is attached to configuration `xtcfg2`, instead of the TIE file `foo.tie`. Because the option to import the TDB with all the encodings preserved is selected, the new TDK created in this step (`xtcfg2.tdb`) includes all the instructions that are present in the file `foo.tie`, and these instructions have the same encodings as in the TDB `xtcfg1.tdb`.

The same results can be obtained when using the command line interface of the TIE compiler as follows:

1. Build and install the first Xtensa configuration `xtcfg1`.
2. Create the TIE description `foo.tie`.

3. Compile a TDK with configuration `xtcfg1` and TIE file `foo.tie` as:
```
tc -d tdk1 --xtensa-core=xtcfg1 -name xtcfg1 foo.tie
```
4. Upload the TDB file `xtcfg1.tdb` from the directory `tdk1` to the processor generator and rebuild configuration `xtcfg1` with the TIE.
5. Build and install the second Xtensa configuration `xtcfg2`.
6. Compile a TDK with configuration `xtcfg2` and TDB file `xtcfg1.tdb` (instead of the TIE file `foo.tie`) as
```
tc -d tdk2 --xtensa-core=xtcfg2 -name xtcfg2 xtcfg1.tdb
```
7. The command in step 6 creates a TDB named `xtcfg2.tdb` in the directory `tdk2`. Upload this TDB and rebuild configuration `xtcfg2` with TIE.

10.2.2 Potential Problems in Compiling TDBs

For the compilation process described in Section 10.2.1 to succeed, the two Xtensa configurations `xtcfg1` and `xtcfg2` must be compatible. If they are not, you may get errors from the TIE compiler when you try to compile `xtcfg1.tdb`, generated with configuration `xtcfg1`, with the configuration `xtcfg2`. For example, if `xtcfg1` is a Little Endian configuration and `xtcfg2` is a Big Endian configuration, this process does not work because assigning compatible encodings is inherently not possible in this situation. While it is difficult to list all possible ways in which configurations may be mismatched, following are a few common reasons:

- The two configurations have different Endianess.
- If the first configuration `xtcfg1` uses wide instructions, the second configuration `xtcfg2` must also enable wide instructions of at least the same width as `xtcfg1`.
- The two configurations must both be either 5-stage pipeline or 7-stage pipeline configurations. This is especially true if the TIE file contains register files, load/store instructions, or TIE lookups. In this situation, instruction schedules are incompatible between the two configurations.
- If the TIE file replicates optional Xtensa ISA instructions in FLIX slots, check to ensure that these optional instructions are enabled in the second Xtensa configuration. For example, if the TIE file `foo.tie` replicates the Xtensa ISA instruction `MUL16S` in a FLIX slot, the first Xtensa configuration `xtcfg1` should have the `MUL16` ISA options selected. The second Xtensa configuration `xtcfg2` must also have this option enabled.
- When compiling file `foo.tie` with configuration `xtcfg1`, you may get warning messages that opcode space for options such as `MAC16` or `FP` (Floating point) option were used. This typically happens when your TIE instructions cannot be encoded in the `CUST0` and `CUST1` opcode space. Thus if the `MAC16` opcode space was used for encoding instructions from `foo.tie`, you would not be able to compile `foo.tdb` with any other Xtensa configuration that has the `MAC16` option enabled.

10.2.3 Reassigning TDB Encodings

It is possible to compile a TDB file into a new TDK, and not import all the automatically generated field and opcode encodings. This is done using an optional flag as follows:

```
tc -d tdk -name bar -reassign_tdb_encodings foo.tdb
```

The preceding command recompiles `foo.tdb` into a new TDK, but without importing the field and opcode encodings that were assigned when `foo.tdb` was created. The TIE compiler reassigns these encodings in the context of the current compile. It is equivalent to the following command:

```
tc -d tdk -name bar foo.tie
```

where `foo.tie` is the original file used to generate `foo.tdb`. Note that if any field and opcode encodings were explicitly specified in the file `foo.tie`, then these are always preserved independent of the `-reassign_tdb_encodings` flag. Only those field and opcode encodings that were automatically generated by the TIE compiler in the process of compiling `foo.tie` into `foo.tdb` are ignored when this flag is used.

In the Xtensa Xplorer environment, you can import a TDB without preserving its opcode encoding by selecting this option in the **Add TIE and TDB Files to a Configuration** pop-up window that displays when you click the **Attach TIE** button on the **Configuration Overview** page.

10.2.4 Header File Behavior when Compiling TDBs

When compiling one or more TIE files into a TDK, you have flexibility in specifying the base name of the header file(s) that is generated, as discussed in Section 10.1.2. When compiling a TDB, you have no control over the header file names that are generated in the new TDK. The header files will have the same name(s) as they did when the original TDB was compiled.

For example, assume that TIE file `foo.tie` is compiled to generate header file `foo.h` and TDB `foo.tdb`. When this TDB is compiled as:

```
tc -d tdk -name bar foo.tdb
```

The header file generated will be `foo.h` because this was the name embedded in the TDB. This is true even though the `-name bar` option is used, and the rest of the files in the TDK have the base name `bar`.

As another example, assume that two TIE files, `aaa.tie` and `bbb.tie` are compiled to generate TDB `foo.tdb`, and separate header files `aaa.h` and `bbb.h` as described in Section 10.1.2. When this TDB is compiled, two separate header files `aaa.h` and `bbb.h` are generated. This behavior is independent of any compile options used during the second compile.

10.2.5 Compiling Multiple TDB Files

It is possible to compile multiple TDB files into a TDK, just as it is possible to compile multiple TIE files. However, compiling multiple TDBs (especially using the default option of preserving all the field and opcode encodings) may cause encoding conflicts.

When you compile a TIE file into a TDB, the TIE compiler assigns field and opcode encodings for all the instructions in the TIE descriptions. This only applies to those instructions for which the encodings are not explicitly specified in the TIE description. The TIE compiler tries to fit all the 24-bit instructions in the `CUST0` and `CUST1` opcode space that is reserved for customer use, and will never be used by Tensilica for any Xtensa ISA instructions. As a result, when you compare the encodings assigned by the TIE compiler to two or more TIE files, they may overlap. Thus if you compile two or more TDBs with their respective encodings preserved, there is a high probability that the compile will fail due to encoding overlap between the two TDBs.

The recommended approach is to compile only one TDB with opcode encodings preserved, or to use the `reassign_tdb_encodings` flag described in Section 10.2.3 when compiling multiple TDB files. Note that if there is no encoding overlap, either by design or by coincidence, the compiling of multiple TDBs with opcode encodings preserved will work as expected.

10.3 Mixing TIE and TDB Files

It is possible to compile a mix of TIE and TDB files into a TDK. Thus the command

```
tc -d tdk -name foo bar.tdb aaa.tie
```

will compile the TIE file `aaa.tie` and the TDB `bar.tdb` into a TDK with base name `foo`. All of the considerations discussed in Section 10.1 “Working with Multiple TIE Files” on page 175 and Section 10.2 “Working with TDB Files” on page 178 continue to apply. In particular you should note the following:

- The `-name` compile option is required because you have more than one input file.
- The `-unique_headers` option is only meaningful if you have more than one TIE file, and does not apply to TDBs.
- The `-reassign_tdb_encodings` flag only applies to TDBs, and applies to all TDBs that are being compiled.

10.4 Restrictions

When you compile multiple TIE files (or TDBs) together, there are several restrictions and coding guidelines that you must follow:

- You are responsible for avoiding all name conflicts across all TIE and TDB files being compiled. For example, you cannot have instructions with the same name in two different files, or two different TIE state with the same name. All naming restrictions that apply to a single TIE file now apply across the multiple TIE files. A recommended approach is to prefix all TIE construct names (including names of schedule, semantic etc.) with a unique prefix in each TIE file to avoid problems when integrating the files together.
- If you provide explicit field and opcode encodings in any of the TIE files, you are responsible for ensuring that they do not conflict with any other encodings in other TIE files. A recommended approach is to not provide any field and opcode encodings and let the TIE compiler automatically assign them for you. Similarly, if you use the `user_register` construct, do not specify the optional user register number, and let the TIE compiler automatically assign it for you.

11. Compile Options for the TIE Compiler

When you run the TIE compiler to compile your TIE code and create a TIE Development Kit (TDK), you can specify several different compile options. Some of these options are convenience options, others alter the flow of how the TIE code is compiled. This chapter describes the available options and their use.

All the options described in this chapter are available as command line flags when you run the TIE compiler using the command line interface. In the Xtensa Xplorer Integrated Development Environment (IDE), several of these options can be directly selected by the user, while others are managed by Xplorer and are not directly visible to the user.

Table 11–4 provides a summary of all the compile options available for use with the TIE compiler. Several of these are self explanatory, and do not need further documentation. Some of the options require further explanation, which is provided in the subsequent sections of this chapter.

Table 11–4. Summary of Compile Options for the TIE Compiler

Compile Option	Description
-a [level]	Generate estimate of hardware area. Optional level is an integer specifying level of detail. In Xtensa Xplorer, this option is enabled by default and can be manually invoked by clicking the button labelled Area on the Configuration TIE Overview page.
-chk_encodings	Check that the TIE description is fully specified with respect to instruction encoding. Generate an error if the TIE compiler needs to assign any opcode, field, format, slot etc. encoding.
-cstub	Generate "cstub" files that simulate your application code with TIE instructions on the native (x86) platform.
-d <name>	Name of the output TDK directory. In Xtensa Xplorer, this option is not directly visible because the directories are managed by Xplorer.
-E	Run only the TIE preprocessor, and print output to STDOUT. This option is not available in Xtensa Xplorer.
-help	Print usage help message along with a brief description of various available compile options. This option is not available in Xtensa Xplorer.
-ignoremsg <msg code(s)>	Comma separated list of specific message codes to be ignored in the compilation. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, there is a tick mark next to each of the messages that are to be ignored.

Table 11–4. Summary of Compile Options for the TIE Compiler (continued)

Compile Option	Description
-ignorewarn <warn code(s)>	Comma separated list of specific warning codes to be ignored in the compilation. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, there is a tick mark next to each of the warnings that are to be ignored.
-libdebug	Generate debug information in simulation libraries. This option is necessary if you want to breakpoint an ISS simulation and view the values of variables in your TIE code. In Xtensa Xplorer, this option is selected on the TIE Compile Options page.
-lint	Run lint checks only on the TIE file. This option is not directly available in Xtensa Xplorer.
-msgall	Show all messages. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, none of the messages have a tick mark next to them.
-name <base>	Set the base name for output files to <base>. If not specified, it defaults to the root name of the TIE file. This option is required when compiling multiple TIE or TDB files. In Xtensa Xplorer, this name is specified in the TIE Compile Options page. Xplorer requires you to specify this name in the dialog box to attach TIE/TDB files to a configuration if you select more than one file to attach.
-nomsg	Do not generate any messages. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, there is a tick mark next to all messages.
-nowarn	Do not generate any warnings. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, there is a tick mark next to all warnings.
-reassign_tdb_encodings	This option is relevant only when compiling TDB file(s). Discard the field and opcode encodings in the TDB, and generate new ones. In Xtensa Xplorer, this selection is made in the Attach TIE and TDB files to a Configuration dialog box that comes up when you click the Attach TIE button on the Configuration Overview.
-reflib	Generate simulation libraries based on the operation (or reference) description of TIE instructions in addition to simulation libraries based on the semantic description. In Xtensa Xplorer, this option is selected on the TIE Compile Options page.
-reportmsg <msg code(s)>	Comma separated list of specific message codes to be enabled during compilation. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, the respective messages do not have a tick mark next to them.
-reportwarn <warn code(s)>	Comma separated list of specific warning codes to be enabled during compilation. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, the respective warnings do not have a tick mark next to them.

Table 11–4. Summary of Compile Options for the TIE Compiler (continued)

Compile Option	Description
-showall	Report all errors, warnings and messages. Overrides all other options for selective enabling and disabling of warnings and messages. In Xtensa Xplorer, this option can be selected on the TIE Compile Options page.
-swtie	Compile the given TIE file as Software-Only TIE. In Xtensa Xplorer, Software-Only TIE can be added on the Configuration Overview page. For more details on Software-Only TIE, refer to Section 3.2.1.
-tdb2tie	Extract the original input TIE file from a tdb. This option is not directly available in Xtensa Xplorer, but is used under the hood where necessary.
-trace [tpp]	Trace execution of the TIE compiler. If the optional string tpp is present, it also traces the TIE preprocessor execution. In Xtensa Xplorer, this option can be selected on the TIE Compile Options page.
-unique_headers	This option is relevant only when compiling more than one TIE file. It creates a unique header file corresponding to each TIE file, instead of the default of a single header file having the name specified with the -name option. In Xtensa Xplorer, this option can be selected on the TIE Compile Options page.
-version	Show TIE compiler version number. This option is not directly available in Xtensa Xplorer. However, the version number of all tools is listed in the file that gets generated when you select the Export State for Support option from the Help menu.
-warnall	Enable generation of all warnings, including several that are off by default. In Xtensa Xplorer, this is done by ensuring that on the TIE Compile Options page, none of the warnings have a tick mark next to them.

11.1 Compile Options for Errors, Warnings and Messages

The TIE compiler may generate errors, warnings, and informational messages when you compile your TIE code. All errors reported by the TIE compiler must be fixed before a TDK can be generated. Error messages cannot be disabled.

Warning messages generated by the TIE compiler usually point to an aspect of your TIE code that deserves attention. Warning messages are not fatal, and a TDK is generated even if the TIE compiler generates warnings. Warning messages fall into two categories. A collection of warnings that are deemed important by Tensilica are turned on by default. A collection of warnings that are deemed less important are turned off by default, and these can be enabled with compiler options as described below.

Messages are informational in nature and do not require any action on your part.

In the Xtensa Xplorer environment, warnings and messages can be enabled or disabled on the **TIE Compiler Options Page** that is accessed by clicking on its icon in the **Configuration Overview**. This page is shown in Figure 11–25, and lists all the warnings and messages generated by the TIE compiler. Selecting a warning and clicking on the Help button opens up a dialog box with details on the warning as shown in Figure 11–25. A warning or message with a tick on the check box next to it is disabled. If there is no tick on the check box, it is enabled. Clicking the check box toggles this state. The **TIE Compiler Options Page** also provides an option to enable all errors, warnings, and messages. When this option is selected, it overrides the individual enabling and disabling of warnings and messages, which is symbolized by the list of warnings and messages being grayed out.

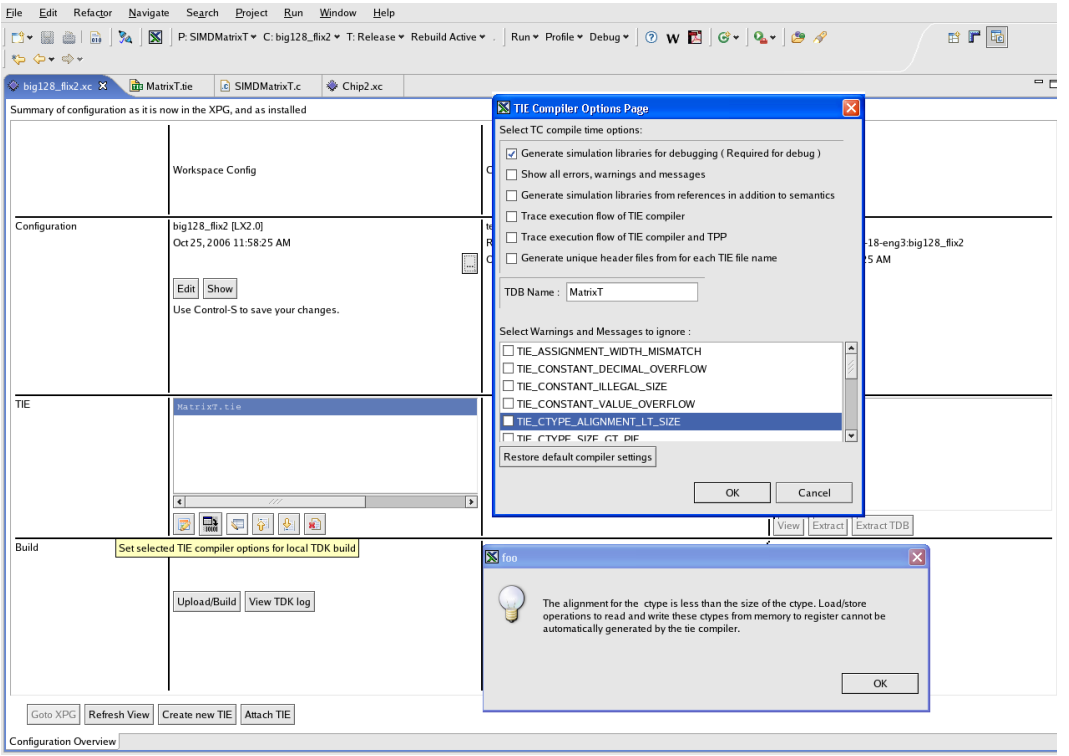

Figure 11–25. TIE Compiler Options Page in Xtensa Xplorer

Subsections 10.1.1 to 10.1.9 describe options to enable and disable various warnings and messages when running the TIE compiler using the command line interface.

11.1.1 -warnall Option

Enables the generation of all warnings, and in particular it enables the generation of warnings that are off by default.

11.1.2 -nowarn Option

Disables the generation of all warnings by the TIE compiler. Use this option with caution. Consider disabling selective warning messages instead, as described in Section 11.1.3.

11.1.3 -ignorewarn Option

Allows you to disable specific warnings by listing the warning code in a comma separated list. Thus the following command:

```
tc -lint -ignorewarn TIE_NO_USE,TIE_NO_ASSIGNMENT foo.tie
```

will run lint check on the file `foo.tie` while ignoring warnings of type `TIE_NO_USE` and `TIE_NO_ASSIGNMENT`. Note that there should not be any white spaces in the comma separated list of warning codes.

This option is redundant in the presence of the `-nowarn` option described in Section 11.1.2. Conversely, when used in conjunction with the `-warnall` option of Section 11.1.1, the `-ignorewarn` option overrides. Thus the following command:

```
tc -lint -warnall -ignorewarn TIE_NO_USE foo.tie
```

will run lint check on the file `foo.tie` while ignoring warnings of type `TIE_NO_USE`, but reporting all other warnings.

11.1.4 -reportwarn Option

Allows you to enable specific warnings, and is most useful to enable some, but not all of the warnings that are turned off by default. The warning codes to be enabled are listed in a comma separated list as follows:

```
tc -lint -reportwarn TIE_FUNCTION_UNUSED,TIE_EXPRESSION_WIDTH_MISMATCH
     foo.tie
```

This command will run lint check on the file `foo.tie`, and will enable the warnings `TIE_FUNCTION_UNUSED` and `TIE_EXPRESSION_WIDTH_MISMATCH`, which are otherwise turned off by default. Note that there should not be any white spaces in the comma separated list of warning codes.

This option is redundant in the presence of the `-warnall` option described in Section 11.1.1. Conversely, when used in conjunction with the `-nowarn` option in Section 11.1.2, the `-reportwarn` option overrides. Thus the following command:

```
tc -lint -nowarn -reportwarn TIE_FUNCTION_UNUSED foo.tie
```

will run lint check on the file `foo.tie` while reporting warnings of type `TIE_FUNCTION_UNUSED` but ignoring all other warnings.

11.1.5 -msgall Option

Enables the generation of all messages by the TIE compiler. This option is useful to enable messages that are turned off by default.

11.1.6 -nomsg Option

Disables the generation of all messages by the TIE compiler.

11.1.7 -ignoremsg Option

Allows you to disable specific messages by listing the message code in a comma separated list. Thus the following command:

```
tc -lint -ignoremsg TIE_OPER_AND_SEM,TIE_GENERATED_OPERATION foo.tie
```

will run lint check on the file `foo.tie` while ignoring messages of type `TIE_OPER_AND_SEM and TIE_GENERATED_OPERATION`. Note that there should not be any white spaces in the comma separated list of message codes.

This option is redundant in the presence of the `-nomsg` option described in Section 11.1.6. Conversely, when used in conjunction with the `-msgall` option in Section 11.1.5, the `-ignoremsg` option overrides. Thus the following command:

```
tc -lint -msgall -ignoremsg TIE_OPER_AND_SEM foo.tie
```

will run lint check on the file `foo.tie` while ignoring messages of type `TIE_OPER_AND_SEM` but reporting all other messages.

11.1.8 -reportmsg Option

Allows you to enable specific messages by the TIE compiler as follows:

```
tc -lint -reportmsg TIE_OPER_AND_SEM foo.tie
```

This command will run lint check on the file `foo.tie` and will enable the messages `TIE_OPER_AND_SEM`, which is turned off by default. This option is redundant in the presence of the `-msgall` option described in Section 11.1.5.

11.1.9 -showall Option

Enables the display of all errors, warnings, and messages. It overrides any other option to selectively enable or disable errors, warnings, or messages.

When processing a TIE description, the TIE compiler by default prints only the first ten errors, warnings or messages that have the same error code. This is to prevent any one type of error code from cluttering the display. When the `-showall` option is used, the TIE compiler prints all instances of all errors, warnings, and messages, and does not restrict the display to the first ten of any type.

Note that certain errors are considered fatal and their occurrence prevents the TIE compiler from any further analysis of the TIE description. When such an error occurs, it is reported immediately, along with any other previously encountered non-fatal errors. No further errors are detected or reported until the fatal error is fixed, and this behavior does not change even if the `-showall` flag is used.

11.2 Compile Options for Multiple TIE Files and TDBs

Before RB-2008.3 release of the TIE compiler, all of your TIE code was required to be in one file. You could only attach one TIE file to an Xtensa configuration. This restriction has been removed on RB-2008.3 release. You can compile multiple TIE files and attach them to a single Xtensa configuration. You can also take a TDB file (which was generated as a result of a previous TIE compiler run) and use it as an input file for a subsequent TIE compiler run. Thus you can specify any number of TIE and TDB files as inputs to the TIE compiler.

In the Xtensa Xplorer environment, you can attach TIE and TDB files to a processor configuration by clicking the **Attach TIE** button in the **Configuration Overview**, and then selecting one or more files from the **Add TIE and TDB Files to a Configuration** dialog box that pops up.

11.2.1 -name <base> Option

The TDK directory generated by the TIE compiler contains several files whose names begin with a common root or base name. By default, this is the name of the input TIE file without the `.tie` extension. It is possible to override this default with the `-name` flag. Thus the following command:

```
tc -d tdk -name bar foo.tie
```

compiles the file `foo.tie`, but uses the name `bar` (instead of the default `foo`) as the base name for the files created in the TDK directory. This option is most useful, and is required, when compiling multiple TIE files. Thus the command:

```
tc -d tdk -name helloworld hello.tie world.tie
```

compiles two TIE files `hello.tie` and `world.tie` and uses the base name `helloworld` for the various files in the TDK directory.

In the Xtensa Xplorer environment, you can specify this name on the **TIE Compiler Options Page** in the box labeled **TDB Name**. When attaching multiple TIE or TDB files to a configuration, Xplorer requires you to specify this name in the **Add TIE or TDB Files to a Configuration** dialog box.

11.2.2 -unique_headers Option

This option is only relevant when compiling more than one TIE file into a TDK. The TDK directory contains a header file that you must include in all the C/C++ program code that uses intrinsics for your TIE instructions. By default, a single header file is created when compiling multiple TIE files. Thus the following command:

```
tc -d tdk -name helloworld hello.tie world.tie
```

creates a single header file named `helloworld.h`, which will include the intrinsics from both the input TIE files. If you would like to generate a unique header file corresponding to each of your TIE files, use the `-unique_headers` option. Thus the command:

```
tc -d tdk -name helloworld -unique_headers hello.tie world.tie
```

creates two header files. The file `hello.h` will include the intrinsics for the instructions defined in the TIE file `hello.tie`. The file `world.h` will include the intrinsics for the instructions defined in the TIE file `world.tie`. Note that C/C++ code that uses intrinsics from only one TIE file need only include the corresponding header file; it is not necessary to include all header files in all the C/C++ files.

In the Xtensa Xplorer environment, you can enable this functionality by selecting the corresponding option on the **TIE Compiler Options Page**.

11.2.3 -reassign_tdb_encodings Option

When you build a TDK with a TIE file named <name>.tie, the TIE compiler generates a binary file named <name>.tdb in the TDK directory. This binary file contains your original input TIE file, along with several other pieces of information needed by the Xtensa Processor Generator to build your customized processor. For example, the TDB file contains all the field and opcode encodings that were generated by the TIE compiler in the process of compiling your TIE description. Thus this TDB file is the one you upload to the processor generator.

When you use a TDB file as an input to the TIE compiler, the fully elaborated TIE description embedded in the TDB file is used for the compilation. This means that all the instructions from the TDB, along with their field and opcode encodings (that were generated during the original compilation) are imported into the new configuration. This allows you to move a TIE description from one project to another while preserving binary compatibility of your TIE instructions.

There may be situations in which you may not want to import the automatically generated field and opcode encodings when you compile a TDB. For example, this may occur when you try to combine two TDB files into a single configuration. If both files have field and opcode encodings that were created automatically, there may be some encoding conflicts between the two files. In this situation, you can use the `reassign_tdb_encodings` compile option. This option discards all the previously assigned field and opcode encodings. The TIE compiler then reassigns the field and opcode encodings in the context of the new configuration. Note that any field and opcode encodings specified in the original input TIE file are still preserved. Only the encodings that were automatically generated by the TIE compiler during the original compilation are discarded, and new ones assigned.

If you are compiling multiple TDB files, this flag applies to all of them. If you have several TDB files and want to import some with encodings preserved and others without, you must do so in two steps. You first need to extract the TIE files from the TDB files whose opcode encodings you do not wish to preserve. This is done using the `-tdb2tie` option described in Section 11.2.4. You then run the TIE compiler with the extracted TIE files and the remaining TDB files (for which you do wish to preserve the opcode encodings).

In the Xtensa Xplorer environment, this selection is made in the **Add TIE and TDB Files to a Configuration** dialog box that displays when you click the **Attach TIE** button in the **Configuration Overview**. For each TDB that is attached to the configuration, a drop down menu gives the option of importing the TDB with or without the opcode encodings preserved. This is shown in Figure 11–26, which also shows the Help dialog box that pops up when you click for help with one of the options from the menu selected. In Xplorer, it is possible in a single step to import some TDB files with encodings preserved and others without encodings preserved.

11.2.4 -tdb2tie Option

When you run the TIE compiler on a TDB file with the `-tdb2tie` option, it extracts the original TIE file that was used to generate the TDB file, and prints it to STDOUT.

This option is not directly available in the Xtensa Xplorer environment. Xplorer uses this option *under the hood* where necessary.

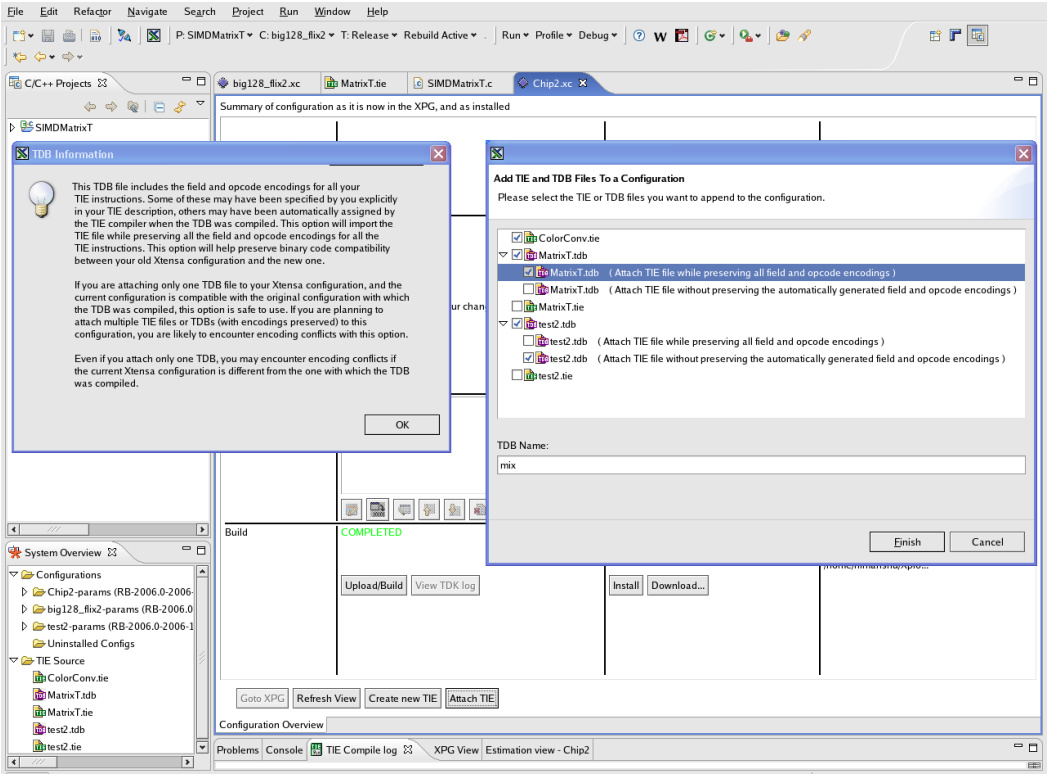

Figure 11–26. Add TIE and TDB Files To a Configuration Menu in Xtensa Xplorer

11.3 Compile Options for Simulation Libraries

The TIE compiler generates DLLs that get linked with the Xtensa ISS, thus allowing you to simulate your TIE instructions. The TIE compiler also supports simulation of TIE instructions on the native x86 platform without using the Xtensa ISS. This section describes some of the compile options that affect the simulator libraries.

11.3.1 -reflib Option

When a TIE file contains both an operation (or reference) description of an instruction and a semantic description, the semantic description is used for the hardware implementation of that instruction. To match this, the simulation libraries are also generated from the semantic description of the instruction.

When the TIE compiler is run with the `-reflib` option, it generates simulation libraries from both the semantic description and the operation description of the instruction. Subsequently, the operation (or reference) description can be simulated in the ISS by using the `--reference` run-time option of the ISS. Note that use of the `-reflib` option does not disable the generation of the simulation libraries from the semantic description, and the default behavior of the ISS is still to simulate the semantic description. Refer to the *Xtensa Instruction Set Simulator (ISS) User's Guide* for more information.

Using the operation description for simulation may result in better ISS run-time performance. This is especially true if the operation description is simple, while the semantic description implements several instructions in the same semantic, and the description is thus more complex. However, you must ensure that the operation and semantic descriptions are equivalent before using this option. The hardware implementation is always generated from the semantic description, and if it is not equivalent to the operation description, the simulation behavior will be different from the behavior of the hardware.

In the Xtensa Xplorer environment, you can generate simulation libraries from the operation description by selecting this option in the **TIE Compiler Options Page**. You must select this option before compiling the TDK. Furthermore, you must specify the `--reference` flag in the **Additional Arguments** section of the **Simulator** tab under **Run Options** to simulate using the operation (reference) description.

11.3.2 -libdebug Option

The default behavior of the TIE compiler is to generate simulation libraries with debug information turned off, because this provides the best run-time performance. However, when debugging your TIE code using the ISS, it is very useful to be able to breakpoint the execution and view the values of various variables in your TIE code. To get this debug information in the simulation libraries, use the `-libdebug` option when compiling the TDK.

In the Xtensa Xplorer environment, you can select this option on the **TIE Compiler Options** page. You must select this option before compiling the TDK.

The TurboXim Engine does not permit viewing TIE variables whether this option is used or not.

11.3.3 -cstub Option

The TIE compiler provides a special mechanism to enable you to simulate C/C++ code with TIE intrinsics on a x86 platform running the Linux or Windows operating system. When you compile a TIE file and generate a TDK, the TIE compiler creates a C/C++ file and a header file to enable this native simulation. Using these files, you can compile your application code with TIE intrinsics using a native compiler such as GCC/G++ or Microsoft Visual C++, and run it as a x86 executable. Refer to Chapter 8, “Simulating TIE Instructions in a Native Environment” on page 117 for more information about this feature.

When you compile a TIE file into a TDK, the files required to run a native simulation are not generated by default. This is done to speed up the compile time. In order to generate these files, you must use the `-cstub` option as follows:

```
tc -d tdk -cstub foo.tie
```

The c-stub files are generated in the TDK directory, and can be copied to any other directory to compile with the rest of the application code.

In the Xtensa Xplorer environment, you can select this option from the **TIE Compiler Options** page. You must select this option before compiling the TDK. You can then use the File Export feature of Xplorer to export the c-stub files to any other directory where they can be compiled with the rest of your application code. Note that Xtensa Xplorer only supports Xtensa ISS simulations; it does not support native simulations.

11.4 Miscellaneous Compile Options

11.4.1 -lint Option

This option performs a lint check of your TIE code, without generating the hardware implementation or the software libraries for various tools. It is most useful for performing a quick check for any errors or warnings related to your TIE code.

This option is not directly available in the Xtensa Xplorer environment, but is used by Xplorer “under the hood” when necessary. The **Fast Scan** click button on the **Configuration TIE Overview** page does a lint check along with a few other things.

11.4.2 -E Option

As described in Section 9.1.3 “Using the TIE Preprocessor” on page 150, you can embed Perl code in your TIE description. This code is processed by the TIE preprocessor, which is the first step in TIE compilation. When you invoke the TIE compiler with the -E option, it only runs the TIE preprocessor and prints its output to STDOUT. The rest of the compilation steps are skipped.

This step is useful if you have complex Perl code embedded in your TIE description, and you are trying to find a bug in this Perl code.

This option is not directly available in the Xtensa Xplorer environment. The TIE preprocessor is invoked by Xplorer when performing a compile, but there is no option available to just invoke the preprocessor.

11.4.3 -chk_encodings Option

When you define a TIE instruction, you may fully specify the encoding of that instruction, partially specify the encoding, or not specify the encoding at all. If the encoding is not specified (or partially specified), the TIE compiler automatically generates this encoding. For example, if you define an `operation` but do not have a corresponding `opcode` statement that defines the opcode encoding for that instruction, the TIE compiler automatically generates an opcode encoding for you.

In order to fully specify the encoding of an instruction, you need to define the opcode encoding as well as the encoding of all the fields used to generate the various operands of that instruction. This is done using the `field`, `operand`, `opcode` and operand_map constructs in addition to the `operation` construct. Further, in the case of a FLIX design, you need to specify the encoding for the `length`, `format` and `slot`(s) that make up your design. While the TIE compiler can generate any and all of these encodings automatically, it is not guaranteed to generate the same encoding across different product releases. Even within a given product release, the encodings generated may be different if the base Xtensa configuration used to compile the TIE is different.

If you prefer to specify all instruction encodings yourself, and want to ensure that the TIE compiler does not automatically generate any field, opcode, format etc. encoding, you should compile the TIE description with the `-chk_encodings` command line flag. This flag will point out any automatic generation that is necessary, and the compile will fail.

A. Color Conversion Example

This appendix describes a color conversion example. This example is used throughout Chapter 5, Chapter 6, and Chapter 7 to explain different TIE features and considerations.

The following section explains the color conversion algorithm and implementation. The next section follows the methodology as described in Section 2.1 to find hotspots to optimize. Appendix B and Appendix C contains the code to apply TIE for optimizing the color conversion implementation.

A.1 YCbCr to RGB Color Conversion

Image and video compression use an internal format to represent a pixel called YCbCr. This widely used format uses a Y component, representing luminance, and two other components that contain color information only. The YCbCr format allows for better compression than other formats because the human eye is less sensitive to color information.

When an image or video frame is decompressed, it often needs to be translated to a different color space. Red, Green, Blue (RGB) is a common color space used. Color conversion translates from one color space to another, in this case from YCbCr to RGB. In mathematical terms the following matrix multiply is necessary.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 91881 \\ 1 & -22554 & -46802 \\ 1 & 116130 & 0 \end{bmatrix} x \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

The above formula is taken from the Independent JPEG Group (IJG) JPEG decoder.

As this function is executed for every pixel, it is executed often. For example, in the case of NTSC video decoding, this means executing the color conversion 720 x 480 x 30 = 10 million times per second.

A.2 Color Conversion C Code Explanation

The previous section described the mathematical operation performed by the YCbCr to RGB color conversion. This section describes how to write the formula into a C program.

The program performs the color conversion on a 256x 32 pixel region. The input pixel region is represented by a 256x32 Y region and a decimated 128x16 region for Cb and Cr. The output of the program is three arrays for the R, G and B components. Every component is an unsigned 8-bit quantity, thus the program uses the following data structures.

```
#define SIZE 256*32

//* Defining 32x32 RGB array
unsigned char R_array[SIZE];
unsigned char G_array[SIZE];
unsigned char B_array[SIZE];

//* Define 32x32 Y array
//* Define 16x16 Cb/Cr arrays
unsigned char Y_array[SIZE];
unsigned char Cb_array[SIZE/4];
unsigned char Cr_array[SIZE/4];
```

The code contains a function that initializes the Y, Cb, and Cr arrays to some random values. It also contains the loop structure that traverses the 256 columns and 32 lines of the pixel region and for each pixel it calls the ConvertYCbCrToRGB function, as follows:

```
int main(){
  int x,y;
  int index_lum;
  int index_chrom;

  // Initialize RGB arrays
  InitYUV(&Y_array[0], &Cb_array[0], &Cr_array[0]);

  // Translate YCbCr to RGB for the 32x32 pixel region
  for(y=0;y<32;y++){
        for (x=0;x<256;x++){
                index_lum        = y*32+x;
                index_chrom = (y>>1)*16+(x>>1);
            ConvertYCbCrToRGB_Orig (Y_array[index_lum],
                                    Cb_array[index_chrom],
```

```
                                    Cr_array[index_chrom],
                                    &R_array[index_lum],
                                    &G_array[index_lum],
                                    &B_array[index_lum]);
        }
  }
}
```

This example uses a simple scheme to handle the color decimation. Every Cb and Cr value is used by four Y values for the color conversion. Figure A–27 shows the relationship between the luminance and chrominance pixel components.

```
   X   X   X   X
     O       O
   X   X   X   X

   X   X   X   X
     O       O
   X   X   X   X
```

Figure A–27. Relationship Between Luminance and Decimated Chrominance Pixels

The color conversion for one pixel is performed by the `ConvertYCbCrToRGB_Orig` function.

```
#define SCALEBITS       16      /* speediest right-shift on some machines */
#define ONE_HALF        ((int) 1 << (SCALEBITS-1))
#define FIX(x)          ((int) ((x) * (1L<<SCALEBITS) + 0.5))

int c_RCr= FIX(1.40200); // 91881
int c_GCb= FIX(0.34414); // 22554
int c_GCr= FIX(0.71414); // 46802
int c_BCb= FIX(1.77200); //116130

void ConvertYCbCrToRGB_Orig (unsigned char Yin,
                             unsigned char Cbin,
                             unsigned char Crin,
                             unsigned char * Rout,
                             unsigned char * Gout,
                             unsigned char * Bout)
{
```

```
  int Y, Cr, Cb;
  int R, G, B;

  /* Cb and Cr must be adjusted to lie in the range -128 .. 127 */
  Cb = (int) Cbin - 128;
  Cr = (int) Crin - 128;

  /* Convert YCbCr to RGB */
  Y = (int)Yin<<SCALEBITS;
  R = Y              + c_RCr * Cr + ONE_HALF;
  G = Y - c_GCb * Cb - c_GCr * Cr + ONE_HALF;
  B = Y + c_BCb * Cb              + ONE_HALF;

  /* Scale down the values and clamp values to the range 0 .. 255 */
  if (R < (1<<SCALEBITS)) *Rout = 0; else if (R > (0xff<<SCALEBITS)) *Rout = 255; else *Rout
= R >> SCALEBITS;
  if (G < (1<<SCALEBITS)) *Gout = 0; else if (G > (0xff<<SCALEBITS)) *Gout = 255; else *Gout
= G >> SCALEBITS;
  if (B < (1<<SCALEBITS)) *Bout = 0; else if (B > (0xff<<SCALEBITS)) *Bout = 255; else *Bout
= B >> SCALEBITS;

} /* ConvertYCbCrToRGB */
```

Notice that the result of the color conversion is saturated to the range of 0 to 255 by the `if` conditions at the end of the function.

All optimizations are performed on the `ConvertYCbCrToRGB` function that combines the loops to go over all pixels in the 256x32 pixel area with the `ConvertYCbCrToRGB_Orig` function shown above. This is done to make it easier to optimize these functions with TIE. The output of this new function will be compared against the previous code to verify that the output is the same. Therefore, the `ConvertYCbCrToRGB` function is the only function that is changed for each TIE optimization.

A.3 C Code for Colorconversion.c:

```
#include <stdio.h>

#define SIZE 256*32
```

```
#define SCALEBITS       16      /* speediest right-shift on some
machines */
#define ONE_HALF        ((int) 1 << (SCALEBITS-1))
#define FIX(x)          ((int) ((x) * (1L<<SCALEBITS) + 0.5))

int verify(unsigned char * Yin,
               unsigned char * Cbin,
               unsigned char * Crin,
               unsigned char * Rout,
               unsigned char * Gout,
               unsigned char * Bout);


static int i_RCr = FIX(1.40200); // 91881
static int i_GCb = -FIX(0.34414); // 22554
static int i_GCr = -FIX(0.71414); // 46802
static int i_BCb = FIX(1.77200); //116130

static int c_RCr= FIX(1.40200); // 91881
static int c_GCb= FIX(0.34414); // 22554
static int c_GCr= FIX(0.71414); // 46802
static int c_BCb= FIX(1.77200); //116130


void ConvertYCbCrToRGB(unsigned char * Yin,
               unsigned char * Cbin,
               unsigned char * Crin,
               unsigned char * Rout,
               unsigned char * Gout,
               unsigned char * Bout){
  int x,y;
  int index_lum;
  int index_chrom;
  int Y, Cr, Cb;
  int R, G, B;
  unsigned char *pY, *pCr, *pCb, *pR, *pG, *pB;

  index_lum = 0;
  index_chrom  = 0;
  pY  = Yin;
  pCr = Crin;
  pCb = Cbin;
  pR = Rout;
  pG = Gout;
  pB = Bout;

  // Translate YCbCr to RGB for the 32x32 pixel region
  for(y=0;y<32;y++){
     for (x=0;x<256;x=x+1){
```

```
        /* U and V must be adjusted to lie in the range -128 .. 127 */
        Cb = (int) (*pCb) - 128;
        Cr = (int) (*pCr) - 128;

        /* Convert YUV to RGB */
        Y = (int)(*pY)<<SCALEBITS;
        pY++;

        R = Y              + c_RCr * Cr + ONE_HALF;
        G = Y - c_GCb * Cb - c_GCr * Cr + ONE_HALF;
        B = Y + c_BCb * Cb              + ONE_HALF;

        /* Scale down the values and clamp values to the range 0 .. 255
*/
        if (R < (1<<SCALEBITS)) *pR++ = 0; else if (R >
(0xff<<SCALEBITS)) *pR++ = 255; else *pR++ = R >> SCALEBITS;
        if (G < (1<<SCALEBITS)) *pG++ = 0; else if (G >
(0xff<<SCALEBITS)) *pG++ = 255; else *pG++ = G >> SCALEBITS;
        if (B < (1<<SCALEBITS)) *pB++ = 0; else if (B >
(0xff<<SCALEBITS)) *pB++ = 255; else *pB++ = B >> SCALEBITS;

        if (x&0x1) {
           pCb++;
           pCr++;
        }
      }
      if ((y&0x1)==0){
           pCb -= 128;
           pCr -= 128;
      }
    }
  }

void InitYCbCr(unsigned char *Y, unsigned char *Cb, unsigned char
*Cr){
  int i;

  for(i=0;i<SIZE;i++){
    *(Y+i) = 16+i%219;
  }

  for(i=0;i<SIZE/4;i++){
    *(Cb+i) = 16+(i+128)%224;
    *(Cr+i) = 240-(i%224);
  }
}

//* Defining 32x32 RGB array
unsigned char R_array[SIZE];
```

```
unsigned char G_array[SIZE];
unsigned char B_array[SIZE];

//* Define 32x32 Y array
//* Define 16x16 Cb/Cr arrays
unsigned char Y_array[SIZE];
unsigned char Cb_array[SIZE/4];
unsigned char Cr_array[SIZE/4];

int main(){
#ifdef TEST
     int ERR;
#endif

     // Initialize RGB arrays
     InitYCbCr(&Y_array[0], &Cb_array[0], &Cr_array[0]);

     // Translate YCbCr to RGB for the 32x32 pixel region
     ConvertYCbCrToRGB (&Y_array[0],
                        &Cb_array[0],
                    &Cr_array[0],
                     &R_array[0],
                     &G_array[0],
                     &B_array[0]);

#ifdef TEST
     ERR = verify(  &Y_array[0],
                         &Cb_array[0],
                    &Cr_array[0],
                    &R_array[0],
                    &G_array[0],
                    &B_array[0]);
     if (ERR){
          printf("ERROR: Incorrect result during verify\n");
     } else {
          printf("Correct result\n");
     }
#endif
}
```

A.4 C Code for VerifyColorConversion.c

```
#define SIZE 256*32

#define SCALEBITS       16      /* speediest right-shift on some
machines */
#define ONE_HALF        ((int) 1 << (SCALEBITS-1))
#define FIX(x)          ((int) ((x) * (1L<<SCALEBITS) + 0.5))

static int c_RCr= FIX(1.40200); // 91881
static int c_GCb= FIX(0.34414); // 22554
static int c_GCr= FIX(0.71414); // 46802
static int c_BCb= FIX(1.77200); //116130

void ConvertYCbCrToRGB_Orig (unsigned char Yin,
                             unsigned char Cbin,
                             unsigned char Crin,
                             unsigned char * Rout,
                             unsigned char * Gout,
                             unsigned char * Bout)
{
  int Y, Cr, Cb;
  int R, G, B;

  /* U and V must be adjusted to lie in the range -128 .. 127 */
  Cb = (int) Cbin - 128;
  Cr = (int) Crin - 128;

  /* Convert YUV to RGB */
  Y = (int)Yin<<SCALEBITS;
  R = Y              + c_RCr * Cr + ONE_HALF;
  G = Y - c_GCb * Cb - c_GCr * Cr + ONE_HALF;
  B = Y + c_BCb * Cb              + ONE_HALF;

  /* Scale down the values and clamp values to the range 0 .. 255 */
  if (R < (1<<SCALEBITS)) *Rout = 0; else if (R > (0xff<<SCALEBITS))
*Rout = 255; else *Rout = R >> SCALEBITS;
  if (G < (1<<SCALEBITS)) *Gout = 0; else if (G > (0xff<<SCALEBITS))
*Gout = 255; else *Gout = G >> SCALEBITS;
  if (B < (1<<SCALEBITS)) *Bout = 0; else if (B > (0xff<<SCALEBITS))
*Bout = 255; else *Bout = B >> SCALEBITS;


} /* ConvertYCbCrToRGB */

int verify(unsigned char * Yin,
               unsigned char * Cbin,
               unsigned char * Crin,
```

```
                unsigned char * Rout,
                unsigned char * Gout,
                unsigned char * Bout){
  int x,y;
  unsigned char R, G, B;
  int index_lum;
  int index_chrom;

  // Translate YCbCr to RGB for the 32x32 pixel region
  for(y=0;y<32;y++){
     for (x=0;x<256;x++){
          index_lum = y*256+x;
          index_chrom = (y>>1)*128+(x>>1);
         ConvertYCbCrToRGB_Orig (Yin[index_lum],
                                 Cbin[index_chrom],
                            Crin[index_chrom],
                            &R,
                            &G,
                            &B);
          if
((R=Rout[index_lum])||(G=Gout[index_lum])||(B=Bout[index_lum])) {
                return 1;
          }
     }
  }
  return 0;
}
```

B. Color Conversion Optimized Using SIMD

This section contains the TIE code and the `ConvertYCbCrToRGB` C function for the color conversion example optimized using SIMD.

B.1 TIE Code for SIMD optimized Color Conversion

```
regfile Coeff 18 8 c
regfile vec128 128 8 vec128
regfile Acc 256 4 acc


operation mymvvec128{out vec128 R, in vec128 I}{}{
     assign R = I;
}

ctype Acc 256 128 Acc

immediate_range imm16x32 -128 112 16

operation ldi_AccL {inout Acc Data, in AR *Addr, in imm16x32 offset
}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Data              = {Data[255:128], MemDataIn128};
}

operation ldi_AccH {inout Acc Data, in AR *Addr, in imm16x32 offset
}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Data              = {MemDataIn128, Data[127:0]};
}

proto Acc_loadi {out Acc a, in Acc *p, in immediate o}{}{
     ldi_AccL a, p, o;
     ldi_AccH a, p, o + 16 ;
}

operation sti_AccL {in Acc Data, in AR *Addr, in imm16x32 offset }{out
VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign MemDataOut128= Data[127:0];
}
```

```
operation sti_AccH {in Acc Data, in AR *Addr, in imm16x32 offset }{out
VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign MemDataOut128= Data[255:128];
}

proto Acc_storei {in Acc a, in Acc *p, in immediate o}{}{
     sti_AccL a, p, o;
     sti_AccH a, p, o + 16 ;
}

operation mv_Acc {out Acc Data, in Acc Input}{}{
     assign Data = Input;
}

proto Acc_move {out Acc a, in Acc b }{}{
     mv_Acc a, b;
}

state Round 32 add_read_write

immediate_range imm16x16 -128 112 16

operation st_vec128_ui {in vec128 Data, inout vec128 *Addr, in
imm16x16 offset}{out VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Addr              = tmpaddr;
     assign MemDataOut128= Data;
}

proto vec128_storeiu {in vec128 v, inout vec128 *p, in immediate o}{}{
     st_vec128_ui v, p, o;
}

operation ld_vec128_ui {out vec128 Data, inout vec128 *Addr, in
imm16x16 offset}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Addr              = tmpaddr;
     assign Data              = MemDataIn128;
}

proto vec128_loadiu {out vec128 v, inout vec128 *p, in immediate o}{}{
     ld_vec128_ui v, p, o;
}

operation movAR2Coeff {out Coeff R, in AR I}{}{
```

```
     assign R = I;
}
immediate_range imm64 0 120 8

operation load64_extend {out vec128 R, inout AR *Addr, in imm64
offset}
                                 {out VAddr, in MemDataIn64}{
     wire [31:0] tmpAddr = Addr+offset;
     assign VAddr   = Addr;
     assign Addr         = tmpAddr;

     wire [63:0] Data= MemDataIn64;
     assign R            = {  8'd0, Data[63:56],
                              8'd0, Data[55:48],
                              8'd0, Data[47:40],
                              8'd0, Data[39:32],
                              8'd0, Data[31:24],
                              8'd0, Data[23:16],
                              8'd0, Data[15:8],
                              8'd0, Data[7:0]};
}

function [7:0] SatFunc([15:0] Input){
     wire [7:0] Tmp = (Input[15:8]==0)?
Input[7:0]:Input[15]?8'd0:8'd255;
     assign SatFunc = Tmp;
}

operation vec_ADDSAT{out vec128 R, in vec128 Y, in Acc Accum}{}{
;for ( $i = 0 ; $i < 16 ; $i++ ) {
;  printf("    wire [31:0] sum%d = TIEadd({Y[%d:%d], 16'd0},
Accum[%d:%d], 1'b0);\n", $i, (127-$i*8), (120-$i*8), (255- ( $i>>1
)*32), (224- ( $i>>1 )*32));
;  printf("    wire [7:0] R%d    = SatFunc(sum%d[31:16]);\n", $i, $i);
;}

     assign R  = {R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15};
}

operation vec_SUB_SCALAR{out vec128 R, in vec128 A, in AR B}{}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [15:0] B%d = B;\n", $i);
;  printf("    wire [15:0] R%d = A%d - B%d;\n", $i, $i, $i);
;}

     assign R  = {R0, R1, R2, R3, R4, R5, R6, R7};
```

```
}

operation vec_MAC {inout Acc Accum, in vec128 A, in Coeff C}{}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = Accum[%d:%d];\n", $i, (255-$i*32),
(224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("   wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};
}

schedule vec_MAC_schedule {vec_MAC}
{
     use Accum 2; def Accum 2;
}

operation vec_MAC_R {out Acc Accum, in vec128 A, in Coeff C}{in Round}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = Round;\n", $i);
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("   wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};
}

schedule vec_MAC_R_schedule {vec_MAC_R}
{
     use Round 2; def Accum 2;
}

semantic vec_MAC_semantic {vec_MAC, vec_MAC_R}
{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = (vec_MAC)?Accum[%d:%d]:Round;\n", $i,
(255-$i*32), (224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
```

```
;  printf("    wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};

}
```

B.2 C code for ConvertYCbCrToRGB Using SIMD TIE

```
void ConvertYCbCrToRGB (unsigned char *  Yin,
                        unsigned char *  Cbin,
                        unsigned char *  Crin,
                        unsigned char *   Rout,
                        unsigned char *   Gout,
                        unsigned char *   Bout)
{
     int x, y;
     int index_lum, index_chrom;
     int * __restrict AddrCb, * __restrict AddrCr;

     vec128 * __restrict pY;
     vec128 * __restrict pR, * __restrict pG, * __restrict pB;

     vec128 tmpCr, tmpCb, Cr, Cb;
     vec128 Y, OldY, R, G, B;

     Acc Res0, Res1, Res2;

     Coeff c_RCr = movAR2Coeff(i_RCr); // 91881
     Coeff c_GCb = movAR2Coeff(i_GCb); // 22554
     Coeff c_GCr = movAR2Coeff(i_GCr); // 46802
     Coeff c_BCb = movAR2Coeff(i_BCb); //116130

     pY             = (vec128*)&Yin[0];
     AddrCb         = (int*)&Cbin[0];
     AddrCr         = (int*)&Crin[0];

     pR             = (vec128*)&Rout[0];
     pG             = (vec128*)&Gout[0];
     pB             = (vec128*)&Bout[0];

     WRound(ONE_HALF);

     pY--;
     pR--;
     pG--;
```

```
        pB--;

        for(y=0;y<32;y++){
             if (y&0x1){
                  AddrCb-=128/4;
                  AddrCr-=128/4;
             }
             for (x=0;x<256;x=x+16){
                  /* U and V must be adjusted to lie in the range -128 ..
   127 */
                  /* Load 8 U and 8 V elements */
                  load64_extend(Cb, AddrCb, 8);
                  Cb = vec_SUB_SCALAR(Cb, 128);

                  load64_extend(Cr, AddrCr, 8);
                  Cr = vec_SUB_SCALAR(Cr, 128);

                  Y= *(++pY);

                  Res0 = vec_MAC_R(Cr, c_RCr);
                  R         = vec_ADDSAT(Y, Res0);
                  *(++pR)   = R;

                  Res1 = vec_MAC_R(Cb, c_GCb);
                  vec_MAC(Res1, Cr, c_GCr);
                  G         = vec_ADDSAT(Y, Res1);
                  *(++pG)   = G;

                  Res2      = vec_MAC_R(Cb, c_BCb);
                  B         = vec_ADDSAT(Y, Res2);
                  *(++pB)        = B;

             }
        }
   } /* ConvertYCbCrToRGB */
```

C. Color Conversion Optimized Using SIMD and FLIX For LX cores

This section contains the TIE code and the `ConvertYCbCrToRGB` C function for the color conversion example optimized using SIMD and FLIX.

C.1 TIE Code for Color Conversion Example Optimized Using SIMD and FLIX

FLIX is added to the TIE file by the first six lines. The rest of the TIE file is identical to the TIE file for the SIMD only implementation.

```
length l64 64 {InstBuf[3:0] == 15}
format f64 l64 {slot_ld, slot_mac, slot_add, slot_store}
slot_opcodes slot_ld {load64_extend, ld_vec128_ui, vec_SUB_SCALAR}
slot_opcodes slot_mac {vec_MAC_R, vec_MAC}
slot_opcodes slot_add {vec_ADDSAT, mymvvec128}
slot_opcodes slot_store {st_vec128_ui, vec_SUB_SCALAR}

regfile Coeff 18 8 c
regfile vec128 128 8 vec128
regfile Acc 256 4 acc

operation mymvvec128{out vec128 R, in vec128 I}{}{
     assign R = I;
}

ctype Acc 256 128 Acc

immediate_range imm16x32 -128 112 16

operation ldi_AccL {inout Acc Data, in AR *Addr, in imm16x32 offset
}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Data              = {Data[255:128], MemDataIn128};
}

operation ldi_AccH {inout Acc Data, in AR *Addr, in imm16x32 offset
}{out VAddr, in MemDataIn128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Data              = {MemDataIn128, Data[127:0]};
}

proto Acc_loadi {out Acc a, in Acc *p, in immediate o}{}{
     ldi_AccL a, p, o;
```

```
     ldi_AccH a, p, o + 16 ;
}

operation sti_AccL {in Acc Data, in AR *Addr, in imm16x32 offset }{out
VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign MemDataOut128= Data[127:0];
}

operation sti_AccH {in Acc Data, in AR *Addr, in imm16x32 offset }{out
VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign MemDataOut128= Data[255:128];
}

proto Acc_storei {in Acc a, in Acc *p, in immediate o}{}{
     sti_AccL a, p, o;
     sti_AccH a, p, o + 16 ;
}

operation mv_Acc {out Acc Data, in Acc Input}{}{
     assign Data = Input;
}

proto Acc_move {out Acc a, in Acc b }{}{
     mv_Acc a, b;
}


state Round 32 add_read_write

immediate_range imm16x16 -128 112 16

operation st_vec128_ui {in vec128 Data, inout AR *Addr, in imm16x16
offset}{out VAddr, out MemDataOut128}{
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Addr              = tmpaddr;
     assign MemDataOut128= Data;
}

proto vec128_storeiu {in vec128 v, inout vec128 *p, in immediate o}{}{
     st_vec128_ui v, p, o;
}

operation ld_vec128_ui {out vec128 Data, inout AR *Addr, in imm16x16
offset}{out VAddr, in MemDataIn128}{
```

```
     wire [31:0] tmpaddr      = Addr+offset;
     assign VAddr        = tmpaddr;
     assign Addr              = tmpaddr;
     assign Data              = MemDataIn128;
}

proto vec128_loadiu {out vec128 v, inout vec128 *p, in immediate o}{}{
     ld_vec128_ui v, p, o;
}

operation movAR2Coeff {out Coeff R, in AR I}{}{
     assign R = I;
}

immediate_range imm64 0 120 8

operation load64_extend {out vec128 R, inout AR *Addr, in imm64
offset}
                                  {out VAddr, in MemDataIn64}{
     wire [31:0] tmpAddr = Addr+offset;
     assign VAddr   = Addr;
     assign Addr         = tmpAddr;

     wire [63:0] Data= MemDataIn64;
     assign R            = {  8'd0, Data[63:56],
                                  8'd0, Data[55:48],
                                  8'd0, Data[47:40],
                                  8'd0, Data[39:32],
                                  8'd0, Data[31:24],
                                  8'd0, Data[23:16],
                                  8'd0, Data[15:8],
                                  8'd0, Data[7:0]};
}

function [7:0] SatFunc([15:0] Input){
     wire [7:0] Tmp = (Input[15:8]==0)?
Input[7:0]:Input[15]?8'd0:8'd255;
     assign SatFunc = Tmp;
}

operation vec_ADDSAT{out vec128 R, in vec128 Y, in Acc Accum}{}{
;for ( $i = 0 ; $i < 16 ; $i++ ) {
;  printf("    wire [31:0] sum%d = TIEadd({Y[%d:%d], 16'd0},
Accum[%d:%d], 1'b0);\n", $i, (127-$i*8), (120-$i*8), (255- ( $i>>1
)*32), (224- ( $i>>1 )*32));
;  printf("    wire [7:0] R%d    = SatFunc(sum%d[31:16]);\n", $i, $i);
;}
```

```
     assign R  = {R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15};
}

operation vec_SUB_SCALAR{out vec128 R, in vec128 A, in AR B}{}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [15:0] B%d = B;\n", $i);
;  printf("    wire [15:0] R%d = A%d - B%d;\n", $i, $i, $i);
;}

     assign R  = {R0, R1, R2, R3, R4, R5, R6, R7};

}

operation vec_MAC {inout Acc Accum, in vec128 A, in Coeff C}{}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = Accum[%d:%d];\n", $i, (255-$i*32),
(224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("   wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};
}

schedule vec_MAC_schedule {vec_MAC}
{
     use Accum 2; def Accum 2;
}

operation vec_MAC_R {out Acc Accum, in vec128 A, in Coeff C}{in Round}{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = Round;\n", $i);
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("   wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};
}

schedule vec_MAC_R_schedule {vec_MAC_R}
```

```
{
     use Round 2; def Accum 2;
}

semantic vec_MAC_semantic {vec_MAC, vec_MAC_R}
{
;for ( $i = 0 ; $i < 8 ; $i++){
;  printf("    wire [31:0] S%d = (vec_MAC)?Accum[%d:%d]:Round;\n", $i,
(255-$i*32), (224-$i*32));
;  printf("    wire [15:0] A%d = A[%d:%d];\n", $i, (127-$i*16), (112-
$i*16));
;  printf("    wire [17:0] C%d = C;\n", $i);
;  printf("    wire [31:0] R%d = TIEmac(A%d, C%d, S%d, 1'b1, 1'b0);\n",
$i, $i, $i, $i);
;}

     assign Accum= {R0, R1, R2, R3, R4, R5, R6, R7};

}
```

C.2 C Code for ConvertYCbCrToRGB Using SIMD and FLIX TIE

This C code is identical to the one used for the SIMD-only TIE code.

```
void ConvertYCbCrToRGB (unsigned char *  Yin,
                        unsigned char *  Cbin,
                        unsigned char *  Crin,
                        unsigned char *   Rout,
                        unsigned char *   Gout,
                        unsigned char *   Bout)
{
     int x, y;
     int index_lum, index_chrom;
     int * __restrict AddrCb, * __restrict AddrCr;

     vec128 * __restrict pY;
     vec128 * __restrict pR, * __restrict pG, * __restrict pB;

     vec128 tmpCr, tmpCb, Cr, Cb;
     vec128 Y, OldY, R, G, B;

     Acc Res0, Res1, Res2;

     Coeff c_RCr = movAR2Coeff(i_RCr); // 91881
     Coeff c_GCb = movAR2Coeff(i_GCb); // 22554
     Coeff c_GCr = movAR2Coeff(i_GCr); // 46802
     Coeff c_BCb = movAR2Coeff(i_BCb); //116130
```

```
        pY              = (vec128*)&Yin[0];
        AddrCb          = (int*)&Cbin[0];
        AddrCr          = (int*)&Crin[0];

        pR              = (vec128*)&Rout[0];
        pG              = (vec128*)&Gout[0];
        pB              = (vec128*)&Bout[0];

        WRound(ONE_HALF);

        pY--;
        pR--;
        pG--;
        pB--;

        for(y=0;y<32;y++){
             if (y&0x1){
                  AddrCb-=128/4;
                  AddrCr-=128/4;
             }
             for (x=0;x<256;x=x+16){
                  /* U and V must be adjusted to lie in the range -128 ..
    127 */
                  /* Load 8 U and 8 V elements */
                  load64_extend(Cb, AddrCb, 8);
                  Cb = vec_SUB_SCALAR(Cb, 128);

                  load64_extend(Cr, AddrCr, 8);
                  Cr = vec_SUB_SCALAR(Cr, 128);

                  Y= *(++pY);

                  Res0 = vec_MAC_R(Cr, c_RCr);
                  R         = vec_ADDSAT(Y, Res0);
                  *(++pR)   = R;

                  Res1 = vec_MAC_R(Cb, c_GCb);
                  vec_MAC(Res1, Cr, c_GCr);
                  G         = vec_ADDSAT(Y, Res1);
                  *(++pG)   = G;

                  Res2      = vec_MAC_R(Cb, c_BCb);
                  B         = vec_ADDSAT(Y, Res2);
                  *(++pB)        = B;

             }
        }
    } /* ConvertYCbCrToRGB */
```

D. Understanding the TDK Report

There are several trade-offs made during any design process, including optimizing the user TIE for area, performance, *etc*. Before you can tune our TIE for a given set of constraints, you need to understand the hardware implementation generated by the TIE compiler for the current TIE source. Xplorer provides two views for this purpose: Configuration TIE Instruction View and Configuration TIE Overview. Additionally, the TIE compiler produces the TIE Compiler Report file that can be used in conjunction with the Xplorer views to understand the hardware implementation generated by the TIE compiler.

This section reviews the contents of the TIE Compiler Report file. It also discusses how you would use this data to extrapolate some of the most commonly needed information related to the TIE hardware implementation, especially when you are optimizing their TIE for area.

D.1 Location

The TIE Compiler Report file can be accessed in Xplorer from the Configuration TIE Overview view by clicking TDK Report.

D.2 Contents

The TIE Compiler Report file contains information on the following:

- FLIX formats and mapping of your TIE operations (*i.e.*, instructions) to each slot within a FLIX format
- Register files, number of read/write ports, and list of instruction operands using each read/write port
- TIE States, and list of instructions using each state
- Load/Store units in the configuration, and list of instructions using each load/store unit
- Your TIE semantics and pipeline flops within each semantic

D.2.1 FLIX Instruction Information

Information related to FLIX formats and the individual slots within each format is described in this section.

- The slots within each format, and instruction bits used by each slot.
  The code below shows an excerpt from the report file. Here you see that FLIX format f64 has four slots. Slot slot_ld occupies Index 0 and uses bits 44-42, 27-22, and 7-4 of the 64-bit instruction. Similarly, slot slot_store occupies Index 3 and uses bits 53-48, 31-14, and 11-8 of the 64-bit instruction. Note that slots within a format will never share the same instruction bits.

  These are the slots in each format:

```
format <f64> slots:
    slot_ld = f64[44:42] f64[27:22] f64[7:4] (index = 0)
    slot_mac = f64[46:45] f64[13:12] f64[33:28] (index = 1)
    slot_add = f64[47:47] f64[41:34] (index = 2)
    slot_store = f64[53:48] f64[21:14] f64[11:8] (index = 3)
```

- Mapping of your TIE operations, and base Xtensa instructions to the slots.
  In the next excerpt from the report file, you see that slot slot_mac has three instructions: `NOP` (from the Xtensa base ISA), and two user TIE `instructions:vec_MAC` and `vec_MAC_R`.

```
slot <slot_mac> instructions: nop=NOP
    NOP vec_MAC vec_MAC_R
```

- Mapping of your TIE instructions to the default Inst slot.
  Note that Inst is the pre-defined Xtensa processor slot that contains all the base Xtensa instructions by default. The excerpt below from the report file that shows all of your TIE instructions mapped to Inst slot.

```
slot <Inst> instructions: nop=NOP
    RUR.Round WUR.Round ld.Coeff ld.vec128 ldi_AccH ldi_AccL
movAR2Coeff
    mv.Coeff mv.vec128 mv_Acc st.Coeff st.vec128 sti_AccH sti_AccL
```

D.2.2 Register Files

Information related to the Xtensa core AR and BR register files, as well as your TIE register files, can be found in sections "Register File Port Information" and "Register File Port and State Information by Stage".

- Section "Register File Port Information" lists each register file in the core, the number of read/write ports on a register file, and the width of the data bus for each port. In addition, this section also identifies the use and def of every read and write port respectively. This information is especially useful in identifying the length of the read and write pipelines, the stages in which data is consumed (in case of read port) or produced (in case of write port), and which instruction operand is consuming or pro-

ducing the data. Based on your TIE, the TIE compiler will determine the optimal number of read/write ports needed for a given register file, as well as assign ports to a given set of instruction operands.

- Consider the excerpt from the report file shown below. It describes the Acc register file which has been defined by the you. In this example, the Acc register file has 2 read ports and 1 write port, each with a data width of 256 bits. The data from read port rd0 is used in stage 2 (*i.e.*, L stage for a 7-stage pipeline) and provided to operand Accum of semantic vec_MAC_semantic (*i.e.*, instructions vec_MAC and vec_MAC_R) in slot index 1 (*i.e.*, slot slot_mac).

```
regfile Acc:
        port rd0: width( 256) stage( 2) operands(
{opnd_vec_MAC_semantic_Accum,1})
        port rd1: width( 256) stage( 1 3) operands(
{opnd_ldi_AccH_Data,0} {opnd_ldi_AccL_Data,0} {opnd_mv_Acc_Input,0}
{opnd_sti_AccH_Data,0} {opnd_sti_AccL_Data,0}
{opnd_vec_ADDSAT_Accum,2})
        port wr0: width( 256) stage( 1 2 3) operands(
{opnd_ldi_AccH_Data,0} {opnd_ldi_AccL_Data,0} {opnd_mv_Acc_Data,0}
{opnd_vec_MAC_semantic_Accum,1})
```

- **Note 1:**
  When a register file is configured for register groups, the read/write ports can have multiple data bus widths.
- **Note 2**:
  If a TIE operation also has an explicitly-defined semantic, then the semantic name is used to compose the operand name. If you specify the operand directly, then the name you provided is used for the operand.

While this section of the report file only reports on the pipeline stages in which data from a particular read port is consumed (or produced in case of a write port), it does not list which operand is consuming (or producing) the data in a given pipeline stage. For this information, refer to the section "Register File Port and State Information by Stage" in the report file.

As shown in the example below, the operands are listed based on the register file port and pipeline stage in which data is consumed (or produced). We know that data from read port rd1 is consumed in pipeline stages 1 and 3 from the above example. The example in Figure 5 identifies that operand Data of operation sti_AccL and operand Data of operation sti_AccH are consuming the data in pipeline stage 3.

```
regfile Acc:
    Port rd0 Operand opnd_vec_MAC_semantic_Accum Slot 1 Stage 2:
vec_MAC
    Port rd1 Operand opnd_ldi_AccH_Data Slot 0 Stage 1: ldi_AccH
```

```
    Port rd1 Operand opnd_ldi_AccL_Data Slot 0 Stage 1: ldi_AccL
    Port rd1 Operand opnd_mv_Acc_Input Slot 0 Stage 1: mv_Acc
    Port rd1 Operand opnd_sti_AccH_Data Slot 0 Stage 3: sti_AccH
    Port rd1 Operand opnd_sti_AccL_Data Slot 0 Stage 3: sti_AccL
    Port rd1 Operand opnd_vec_ADDSAT_Accum Slot 2 Stage 1: vec_ADDSAT
    Port wr0 Operand opnd_ldi_AccH_Data Slot 0 Stage 3: ldi_AccH
    Port wr0 Operand opnd_ldi_AccL_Data Slot 0 Stage 3: ldi_AccL
    Port wr0 Operand opnd_mv_Acc_Data Slot 0 Stage 1: mv_Acc
    Port wr0 Operand opnd_vec_MAC_semantic_Accum Slot 1 Stage 2:
vec_MAC vec_MAC_R
```

D.2.3 TIE States

TIE states can be read from (and written to) by many TIE operations. The pipeline stage in which a read is performed is indicated by a use. The pipeline stage in which a write is performed is indicated by a def. All the use and def details of a given TIE state can be found in sections "Schedule Information for States" and "Register File Port and State Information by Stage" of the report file.

The following example from section "Register File Port and State Information by Stage" shows that state Round is written to (*i.e.*, def) in pipeline stage 4 (*i.e.*, stage W of 7-stage pipeline) by instruction WUR.Round in FLIX slot 0.

```
states:
    State Round (def) Slot 0 Stage 4: WUR.Round
    State Round (use) Slot 0 Stage 1: RUR.Round
    State Round (use) Slot 1 Stage 2: vec_MAC_R
```

D.2.4 TIE Semantics

The implementation of multi-cycle TIE semantics is spread over multiple pipeline stages. Between each pipeline stage exist staging registers. These staging registers hold values used by the logic in the subsequent stages. For all multi-cycle TIE semantics implemented by you, the report lists all the staging registers, width of each staging register (i.e., number of flip-flops in the register), and the variable whose value is held in the staging register. The example below shows the staging registers in the semantic vec_MAC_semantic. Let's evaluate the following statement from the example:

```
    16 * 1         A0: 1 -> 2
```

Let's deconstruct this statement to analyze its content. "A0" is the variable whose value is held in these staging registers. "1 -> 2" indicates that the staging register is between the Stage 1 (E stage) and Stage 2 (L stage for 7-stage pipeline). "16 * 1" indicates that there is only one staging register, and it consists of 16 flip-flops.

Semantic: vec_MAC_semantic

```
1 * 2          vec_MAC: 0 -> 2
16 * 1          A0: 1 -> 2
18 * 1          C0: 1 -> 2
16 * 1          A1: 1 -> 2
18 * 1          C1: 1 -> 2
16 * 1          A2: 1 -> 2
18 * 1          C2: 1 -> 2
16 * 1          A3: 1 -> 2
18 * 1          C3: 1 -> 2
16 * 1          A4: 1 -> 2
18 * 1          C4: 1 -> 2
16 * 1          A5: 1 -> 2
18 * 1          C5: 1 -> 2
16 * 1          A6: 1 -> 2
18 * 1          C6: 1 -> 2
16 * 1          A7: 1 -> 2
18 * 1          C7: 1 -> 2
```

Index