cadence®

Xtensa® Microprocessor Programmer's Guide

For Xtensa Tools Version 11

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

Issue Date:05/2014

RF-2014.0
PD-14--1630-10-00

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

Contents

1. Introduction to Xtensa Processor Programming 1
- 1.1 Overview 1
  - 1.1.1 The Xtensa Processor 1
  - 1.1.2 Tools to Manage Configurability 2
  - 1.1.3 The Xtensa Hardware Abstraction Layer (HAL) 2
- 1.2 The Configuration 3
- 1.3 What If My Results Are Not Identical? 3
- 1.4 Xtensa Exception Architecture Variants 3

2. Writing Xtensa Assembly Code 5
- 2.1 Reasons to Write Assembly Code 5
- 2.2 A Simple Example 5
- 2.3 Windowed Calling Convention 12
  - 2.3.1 Parameter Passing 12
  - 2.3.2 Function Invocation and Return 17
- 2.4 Preprocessing Assembly Code 20
- 2.5 Assembler Relaxations (and Literals) 22
- 2.6 Locating Code at Fixed Locations 24
- 2.7 Writing Efficient Assembly Code 28
  - 2.7.1 Branch Instructions 28
  - 2.7.2 Scheduling Instructions by Hand 29
  - 2.7.3 Scheduling Instructions with the Assembler 30
  - 2.7.4 Avoiding Window Overflows 30
- 2.8 Other Tips and Tricks 31
  - 2.8.1 Obtaining the Program Counter 31
  - 2.8.2 Should I use `bbsi` or `bbsi.l`? 31
  - 2.8.3 Abstracting Configuration Options Using `coreasm.h` 32
  - 2.8.4 The GNU Assembler for Xtensa Processors 33

3. Xtensa Exception Architecture 35
- 3.1 The `PS` Register 35
  - 3.1.1 Exception Mode and Masking Interrupts 38
  - 3.1.2 Exception Semantics 38
  - 3.1.3 Exception Vectors 38
  - 3.1.4 Windowed Code and Xtensa Processor State 39
- 3.2 Low-Priority Interrupts (a.k.a. Level-One Interrupts) 40
- 3.3 Medium-Priority Interrupts 41
- 3.4 High-Priority Interrupts 41
- 3.5 Can I Write High-Priority Interrupt Handlers In C? 41

4. Resetting the Processor 43
- 4.1 Reset State Requirements 43
- 4.2 The First Instruction: Making Space for Literals 48
- 4.3 Interrupts 49
- 4.4 Getting a True Cycle Count 49
- 4.5 Debug State 49

4.6 Window Registers Initialization 50
4.7 Region Protection and Cache Initialization 50
4.7.1 Disabling the Caches and Creating Address Mappings 51
Pitfalls 53
Simplification for Region Protection 53
4.7.2 Initializing the Instruction Cache 53
4.7.3 Initializing the Data Cache 55
4.7.4 Setting Memory-Access Modes 56
Simplification for Region Protection 58
4.8 ROM Unpacking 58
4.9 Calling the First C Function 61
4.10 What Happens if the First C Function Returns? 63
4.11 Building the Reset Vector 63
5. Window Exception Handlers 65
5.1 Architectural Overview of Windowed Registers 65
5.2 Stack Frame Layout 76
5.3 Window Exception Handlers 77
6. Timer Interrupts and Interrupt Handling 85
6.1 Setting the Timers 85
6.1.1 C-Callable Assembly to Read CCOUNT 85
6.1.2 Reading CCOUNT with Inline Assembly 86
6.1.3 Reading CCOMPARE 88
6.1.4 Setting CCOMPARE 89
6.1.5 Setting the First Timer Value 90
6.2 Enabling Interrupts 91
6.3 Handling the First Interrupt 92
6.3.1 The Interrupt Handling Plan 93
6.3.2 The UserExceptionVector 94
6.3.3 The UserExceptionHandler Prolog 95
6.3.4 Saving the Current State 96
6.3.5 Saving the User Stack Pointer 99
6.3.6 Moving the Base Save Area 99
6.3.7 Setting the Stack Pointer 100
6.3.8 Calling the C Handler 101
6.3.9 Return Overview 102
Processor Exception Mode 103
Moving the Base Save Area 103
Restoring the Processor State 103
Returning from the Interrupt 104
6.4 The C Interrupt Handler 105
6.5 Nested Interrupt Handling 107
6.5.1 Interrupt Prioritization 107
6.5.2 Interrupt Nesting Example 108
6.5.3 Interrupt Stack Use 109
6.5.4 Managing Software Prioritized Interrupts 109
6.5.5 Allowing Nested Interrupts via `INTENABLE` 112
6.5.6 Addition of an Interrupt Dispatching Mechanism 116
6.5.7 Handling Kernel Exceptions 119

6.5.8 Examples of Nested Interrupts 120
7. Context Switching 125
7.1 Task Creation 125
7.2 Handling the Tick 128
7.3 Switching the Context 131
7.3.1 Notes on Context-Switching Address Registers 134
7.4 Using and Testing the Switch 135
8. Supporting Configurable Processors 141
8.1 HAL Support 142
8.2 Lazy Context Switching 142
8.3 Custom State Save Area 145
8.4 Save and Restore Mechanisms 148
9. Memory Management and Protection 153
9.1 Simple Configurations 153
9.2 MMU with Translation Lookaside Buffer and Autorefill 154
9.2.1 MMU Versions 154
9.2.2 TLB Ways 154
9.2.3 TLB Hit, Miss, and MultiHit 155
9.2.4 Address Space Identifier (ASID) 155
9.2.5 Protection and Privilege 156
9.2.6 ASID Management 156
9.2.7 Access Modes 157
9.2.8 Successful Translation 158
9.2.9 Wired Ways 159
9.2.10 About Miss Exceptions and Interrupt and Exception Handlers 159
9.2.11 Autorefill Ways and Page Tables 160
9.2.12 How Large is the Page Table? 162
9.2.13 The Whole TLB 162
9.2.14 Memory Map 164
9.3 Example TLB Miss Handler 165
A. Full Context Switching Source Code 169
A.1 Makefile 169
A.2 interrupts.h 169
A.3 intdisp.c 170
A.4 kernv.S 171
A.5 l1h.S 171
A.6 main.c 180
A.7 reset.S 181
A.8 task.c 186
A.9 task.h 187
A.10 timer.h 187
A.11 userv.S 189
A.12 windv.S 189
B. MMU Code Examples 195
B.1 simple.S 195
B.2 mmu_context.h 196
B.3 fault.c 199

B.4 mmu.c 208
C. Xtensa Exception Architecture Version 1 211
C.1 XEA1 Versus XEA2 211
C.2 Differences in Register Window Exception Vectors 212
C.3 Xtensa Exception Architecture 1 Description 212
C.3.1 The PS Register for XEA1 212
C.3.2 Exception Semantics 214
C.3.3 Exception Vectors for XEA1 214
C.3.4 Program Code for XEA1 215
C.3.5 Windowed Code and Xtensa Processor State for XEA1 215
C.3.6 XEA1 Interrupts and Window Overflows 216
C.3.7 Controlling the Caches for Processors with XEA1 216

List of Figures

Figure 1. PS Register Format—Xtensa Exception Architecture 2 35
Figure 2. WxTLB Input Register Format for *Region Protection with Translation* 52
Figure 3. Xtensa Stack Frame Layout 77
Figure 4. User Stack Layout 97
Figure 5. Interrupt Stack Layout 101
Figure 6. PS Register Format (XEA2) 101
Figure 7. Nested Interrupt Stack Layout 109
Figure 8. BAD User Stack Layout for Custom State 146
Figure 9. User Stack Layout with Custom State 147
Figure 10. Example TLB Way 155
Figure 11. RASID Special Register Format 156
Figure 12. Example MMU Successful Translation 158
Figure 13. Example Autorefill Operation 161
Figure 14. Example PTE-Load Computation 162
Figure 15. Example MMU Way Configuration for MMU versions 1 and 2 163
Figure 16. MMU Way Configuration for MMU version 3 164
Figure 17. ITLBCFG Register Format 164
Figure 18. DTLBCFG Register Format 164
Figure 19. Example MMU Memory Map for MMU versions 1 and 2 165
Figure 20. PS Register Format—Xtensa Exception Architecture 1 212

List of Tables

Table 1. Stack Frame Space for the Window Save Area 19
Table 2. PS Register Fields—Xtensa Exception Architecture 2 36
Table 3. Register State at callmain 69
Table 4. Register State at main 70
Table 5. Register State at func 72
Table 6. Register State at func1 74
Table 7. Register State Following Overflow 75
Table 8. Vector Offsets 78
Table 9. Lazy Context Switching Example Sequence 142
Table 10. PS Register Fields—Xtensa Exception Architecture 1 213

Preface

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.
- **[**`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).
- `4`'b`0010` is a 4-bit value specified in binary.
- `12`'o`7016` is a 12-bit value specified in octal.
- `10`'d`4839` is a 10-bit value specified in decimal.
- `32`'h`ff2a` or `32`'`HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- 0x at the beginning of a value indicates a hexadecimal value.
- b means bit.
- B means byte.
- flush is deprecated due to potential ambiguity (it may mean write-back or discard).
- Mb means megabit.
- MB means megabyte.
- PC means program counter.
- word means 4 bytes.

Changes from the Previous Version

The following changes have been made to this document for the Tensilica RF-2014.0 Xtensa Tools Version 11.0 release.

- Clarified FRAME_SIZE==124 constant in Appendix A.

Processor Version Compatibility for Xtensa Tools Version 11

The Xtensa Tools version 11 released with the Tensilica RF-2014.0 or later releases supports these versions of Tensilica processors:

- Xtensa LX processors (Releases RA-200X.x to RF-201X.x)
- Xtensa processors (Releases RA-200X.x to RF-201X.x)
- Diamond Standard Series processors (Rev. A to Rev. D hardware)

Xtensa Tools Version 11 and Xtensa LX6 and Xtensa 11

The primary targets for the use of Xtensa Tools version 11 are Xtensa LX6 and Xtensa 11 processors built from the RF-201X.X release versions. All modes of use are supported, from system architecture analysis to code development, hardware co-verification, and silicon bringup. Unless otherwise stated, all features and use modes described in this document are applicable for use with these processors.

Throughout this guide, many references to Xtensa refer generally to any Tensilica processor (Xtensa LX, Xtensa) that implements the Xtensa instruction set architecture, unless otherwise noted.

- Features that are available only for Xtensa LX processors are preceded with the symbol For LX cores .

Xtensa Tools Version 11 and Software Upgrades for Xtensa 6, 7, 8, 9, 10 and LX1, LX2, LX3, LX4, and LX5 Processors

Xtensa Tools Version 11 is compatible and tested for use with these previous versions (that is, releases prior to this RF-2014.0 release) of Xtensa and Diamond Standard processors family (RA-200X.x to RD-201x.x) for code generation, profiling, and debug for post-silicon processors, via the software upgrade process.

1. Introduction to Xtensa Processor Programming

The Xtensa processor architecture is the first major architecture designed expressly for embedded system-on-a-chip (SOC) applications, instead of traditional desktop computer system applications.

1.1 Overview

Xtensa processors have unique programming requirements that differ in some ways from traditional processors.

The purpose of the *Xtensa Microprocessor Programmer's Guide* is to familiarize you with these programming requirements through instruction in the following three areas of knowledge:

- Knowledge of the Xtensa architecture necessary to write or port an operating system (OS), or smaller runtimes in the absence of a full-fledged OS
- Knowledge of the tool chain that supports Xtensa processors
- Knowing how to implement common OS functions, such as context switching, exception handling, and interrupt handling

The *Xtensa Microprocessor Programmer's Guide* is not a replacement for the reference materials that describe the Xtensa processor and its tools. Those manuals cover, in much greater depth, the features and capabilities of the processor and tools. The *Xtensa Microprocessor Programmer's Guide* is intended to offer a step-by-step explanation of the processor and tools—especially as they are used to develop operating systems and other low-level software.

1.1.1 The Xtensa Processor

The Xtensa Instruction Set Architecture (ISA) specifies a 32-bit RISC-like architecture expressly designed for embedded applications.

These aspects of the Xtensa ISA give OS designers additional factors to consider during the design process:

- Varying instruction widths
- Use of window registers
- Processor configurability
- Processor extensibility (with the Tensilica Instruction Extension (TIE) language)

1.1.2 Tools to Manage Configurability

For operating systems that must run on multiple Xtensa configurations, configurability is the biggest challenge. Cadence provides various tools to help OS programmers manage configurability.

Although these tools have different expressions, they all follow a common theme. The tools to manage configurability provide a uniform interface for each processor configuration. The implementation of that interface is different for each processor configuration, but the interface itself is the same. Therefore, the Xtensa Processor Generator produces unique (and processor configuration specific) versions of these common interfaces for each processor configuration.

1.1.3 The Xtensa Hardware Abstraction Layer (HAL)

The Xtensa Hardware Abstraction Layer (HAL) aids in operating system compatibility across different processor configurations. There are really two versions of the HAL. The first is compile-time HAL (CHAL). The CHAL provides source-code compatibility. Generally speaking, the CHAL consists of C preprocessor and assembler macros that represent the particular configuration of the processor.

The second HAL version is the link-time HAL (LHAL), which offers object code compatibility for all processor configurations. Operating systems using this mechanism should be compiled to the base ISA and written to use the interface provided by the HAL. The operating system objects are then linked with the HAL for a particular processor configuration. Assuming that the operating system is fully and correctly using the HAL, the operating system can then fully support that particular processor configuration.

The LHAL also provides routines that are of general use to all low-level software. These routines often deal with Xtensa ISA-specific operations that will be the same for all configurations. For example, the HAL provides routines that will both save live window frames to the stack and save and restore custom processor state.

This guide often refers to the Xtensa HAL generically as the HAL. In those cases, the specific HAL version, CHAL or LHAL, can be understood from the context of the discussion. Sometimes, the discussion applies equally to both HAL versions.

Refer to the *Xtensa System Software Reference Manual* for details on the Xtensa HAL.

1.2 The Configuration

Because the Xtensa processor is configurable, many of the tools that Cadence provides help to manage that configurability. However, to simplify the discussion, this guide uses a single reference configuration: the DF_212GP. Among other things, this configuration includes the following features:

- Little endian
- 32 address registers
- Several interrupts and timers
- Instruction and data caches
- Region protection
- Debug facilities

1.3 What If My Results Are Not Identical?

Your results for the examples may differ from this guide's results. These results can vary for a variety of different reasons: addresses may be different for different processor configurations; different vectors may be in ROM or in RAM; or the compiler may have incrementally changed its code generation.

For the most part, the examples in this guide illustrate a given nuance of Xtensa programming and continue to serve that purpose even if some of the details change for a particular processor configuration or Xtensa processor release.

1.4 Xtensa Exception Architecture Variants

The Xtensa LX processor supports only Xtensa Exception Architecture 2 (XEA2). Previous Xtensa processors also supported Xtensa Exception Architecture 1 (XEA1). Note that XEA1 is no longer supported in Xtensa LX cores. For backward compatibility with older Xtensa cores, Xtensa Tools Version 7.0 continues to support both XEA1 and XEA2, where applicable. Unless you have a specific reason to use the older XEA1, such as for backward compatibility, you should use XEA2.

The examples in this document use a processor configuration that includes XEA2. XEA1 is described in Appendix C.

2. Writing Xtensa Assembly Code

2.1 Reasons to Write Assembly Code

Operating-system programming for Xtensa processors requires that you write or use some assembly code. The user exception handler, the kernel exception handler, the window handlers, the reset handler—all of these low level functions must be written in assembly code for several reasons. The main reason is that entry and exit conventions for these vectors do not correspond to the standard C function entry and exit convention, so the compiler cannot generate them. Also, these vectors require both:

- Access to certain special registers that are not accessible through C
- A level of control over the order of operations not offered by the compiler

There are, of course, additional reasons to write assembly code. Often assembly routines can exhibit better performance, especially when operations must be performed with interrupts locked. This can be critical in certain operating system specific cases.

No matter what the reason, writing assembly code for an Xtensa processor—like writing assembly for any processor—requires understanding of both the processor and its tools.

2.2 A Simple Example

The easiest way to create assembly code is to use the C compiler. Consider the following C code that computes the N'th Fibbonacci number.

```
unsigned int fib( unsigned int val )
{
        int cur;
        int cur1;
        int newval;

        if( val == 0 )
                return 0;
        if( val < 3 )
                return 1;

        val = val - 2;
        cur = cur1 = 1;
        do
        {
                newval = cur + cur1;
                cur1 = cur;
                cur = newval;
```

```
                val--;
        } while ( val );

        return cur;
}
```

The code is quite simple. It checks certain boundary cases and then computes the Fibbonacci number.

You can see the assembly code generated by the compiler in several different ways. One way is to disassemble the compiled file. To do this, first compile the example using this command:

```
$ xt-xcc –O2 -Os –c fib.c
```

Next, disassemble the resulting object file (called “fib.o”) with the following command:

```
$ xt-objdump –dr fib.o
```

which produces output similar to the following:

```
fib.o:     file format elf32-xtensa-le

Disassembly of section .text:

00000000 <fib>:
   0:   004136          entry   a1, 32
   3:   22cc            bnez.n  a2, 9 <fib+0x9>
                        3: R_XTENSA_SLOT0_OP    .text+0x9
   5:   020c            movi.n  a2, 0
   7:   f01d            retw.n

00000009 <fib+0x9>:
   9:   0332f6          bgeui   a2, 3, 10 <fib+0x10>
                        9: R_XTENSA_SLOT0_OP    .text+0x10
   c:   120c            movi.n  a2, 1
   e:   f01d            retw.n

00000010 <fib+0x10>:
  10:   140c            movi.n  a4, 1
  12:   150c            movi.n  a5, 1

00000014 <fib+0x14>:
  14:   fec222          addi    a2, a2, -2
  17:   654a            add.n   a6, a5, a4
  19:   220b            addi.n  a2, a2, -1
  1b:   054d            mov.n   a4, a5
  1d:   065d            mov.n   a5, a6
  1f:   ff4256          bnez    a2, 17 <fib+0x17>
```

```
                        1f: R_XTENSA_SLOT0_OP   .text+0x17
  22:   062d            mov.n   a2, a6
  24:   f01d            retw.n
```

The compiler generates the above code for the `fib` function.

Using the disassembler does not produce the most readable disassembly, but it is useful because it does not require access to the source code. Also, the assembler sometimes adjusts the code produced by the compiler. For example, the assembler might align branches or work around limitations of a branch instruction's range.

However, if the source code is available, as it is in this example, you can use the compiler to create a much more readable version of the assembly code. This command asks the compiler to produce an assembly file rather than a completed object code.

Use the following command to produce a file called `fib.s`:

```
$ xt-xcc –O2 -Os –S fib.c
```

To filter out compiler-generated heuristic and performance analysis comments for something more readable, use the following command:

```
$ grep -v '^#' fib.s

 #  xt-xcc::7.0.0-development

        #------------------------------------------------------------
        # Compiling fib.c (/tmp/cc0M#1861f7f8.oXQX5V)
        #------------------------------------------------------------

        #------------------------------------------------------------
        # Options:
        #------------------------------------------------------------
        #  Target:xtensa, ISA:xtensa, Pointer Size:32
        #  -O2  (Optimization level)
        #  -g0  (Debug level)
        #  -m2  (Report advisories)
        #------------------------------------------------------------


        .text
        .align  1
        .literal_position

        # Program Unit: fib

        # Optimized for space
```

```
        .type   fib, @function
        .align   4
        .global fib
fib:
        .frame  a1, 32
.LBB1_fib:
        entry   a1,32
        bnez    a2,.Lt_0_2

        movi.n  a2,0
        retw.n

.Lt_0_2:
        bgeui   a2,3,.Lt_0_4

        movi.n  a2,1
        retw.n

.Lt_0_4:
        movi.n  a4,1
        movi.n  a5,1
        addi    a2,a2,-2

.Lt_0_7:
        add.n   a6,a5,a4
        addi.n  a2,a2,-1
        mov.n   a4,a5
        mov.n   a5,a6
        bnez    a2,.Lt_0_7

        mov.n   a2,a6
        retw.n

        .size   fib, . - fib
```

Much of the information in this output is irrelevant to this discussion. However, here the assembly code is easier to read and includes information that is lost in the compilation, such as assembler directives and compiler generated loop labels. Consider in greater detail the disassembly portion of this ouput.

The first section to examine is the header for this function. This header consists of a series of assembler directives. These directives do not actually generate any instructions, but they tell the assembler information about the instructions to be generated.

```
        .text
        .align  1
```

The `.text` directive tells the assembler to emit its output into the `.text` section while the `.align` directive tells the assembler to put the next object byte into the text section at a single-byte aligned location.

Now that the section is set up, the code addresses the function itself, including setting up its alignment, type, and name:

```
            .type   fib, @function
            .align   4
            .global fib
    fib:
```

The `.type` directive puts information in the object file that identifies the symbol `fib` as a function. Debuggers can use this type of information to know how to display information about the symbol. The `.align` directive tells the assembler to place the next object byte at a four-byte boundary. PC-relative call instructions require target addresses be four-byte aligned. Therefore, this align directive makes the function callable. The `.global` directive makes the symbol `fib` externally available, so it can be used outside of this object file. Finally, the `fib` symbol itself is declared.

Now that the symbol for the function has been properly declared, there is one final directive. In particular, it specifies which register contains the frame pointer[1] and the initial size of the stack frame.

```
            .frame  a1, 32
```

Now consider the generated code itself. Recall that the example C code is structured into two different sections. The first section checks for certain special cases of input. In particular, the loop need not be executed if the input value is less than three.

Every function begins with an `entry` instruction. This instruction is required by the windowed register calling convention as discussed in Section 2.3 on page 12.

```
    .LBB1_fib:
            entry   a1,32
```

The body of the function follows the entry. The first guard in the C-code is:

```
            if( val == 0 )
                    return 0;
```

1. The frame pointer is the stack pointer for frames without a call to `alloca()`. Functions that call `alloca()` must move the stack pointer to create new space on the stack. The frame pointer is the register that is used as a base for local variables and arguments passed on the stack. These do not move when an `alloca()` call is made. Because the stack pointer is moving, another register must be used to hold the base pointer for the local variables.

This is handled in the following lines of assembly code:

```
        bnez    a2,.Lt_0_2
        movi.n  a2,0
        retw.n
.Lt_0_2:
```

This code sequence demonstrates that the `val` parameter to the function is passed in register a2. The `bnez` instruction checks to see if a2 is not zero. If a2 is not zero, control jumps to the given label (in this case `.Lt_0_2`), which is located immediately after the return. If the branch is not taken, then the return value (in this case zero), is moved into a2 (unnecessary in this case because a2 is already zero) and the `retw.n` instruction returns from the function.

Note that some instructions use the `.n` suffix. Xtensa processors have an optional code density feature that provides 16-bit versions of some commonly used instructions. The mnemonics for these narrow 16-bit instructions all end with `.n`. The compiler and the assembler use narrow instructions where possible to achieve better code density. Also, the assembler may relax density instructions into no-density instructions (that is, convert a 16-bit instruction into its 24-bit equivalent instruction) to meet loop-instruction alignment requirements and to align branch targets for performance.

If a2 was not zero, then control reaches the next section of code. In C:

```
        if( val < 3 )
                return 1;
```

The assembly code for this C sequence follows.

```
.Lt_0_2:
        bgeui   a2,3,.Lt_0_4
        movi.n  a2,1
        retw.n
.Lt_0_4:
```

The Xtensa processor offers an extensive set of branch instructions; one of them is used above. The `bgeui` instruction branches to the destination if the register is greater than or equal to a particular immediate value. In this case, if a2 is greater than or equal to three, control is transferred to the .Lt_0_4 label. If it is less than three, `movi.n` loads the return value 1 into a2, and the function returns.

All of the guard conditions have now been handled and now the real computation begins. This section of the code has two components: setup for the loop and the body of the loop. The C code for the setup consists of two assignments.

```
        val = val - 2;
        cur = cur1 = 1;
```

The assembly code for these assignments is quite straightforward.

```
.Lt_0_4:
        movi.n  a4,1
        movi.n  a5,1
        addi    a2,a2,-2
```

Register a4 serves as variable `cur1`, and a5 serves as `cur`; both registers are initialized to 1. The register a2, which serves as the variable `val`, has two subtracted from it. (Note that there are many equivalent ways of getting the same result. Copying a4 to a5 with a `mov` instruction to initialize a5 would have also been correct and equally optimal.)

The loop itself is straightforward in C and assembly code. The C looks like this:

```
do
{
        newval = cur + cur1;
        cur1 = cur;
        cur = newval;
        val--;
} while ( val );
```

The assembly code looks like this:

```
.Lt_0_7:
        addi.n  a2,a2,-1
        add.n   a6,a5,a4
        mov.n   a4,a5
        mov.n   a5,a6
        bnez    a2,.Lt_0_7
```

Once into the body of the loop, the relationship between the C and assembly code is quite clear, although the compiler reorders some operations. The looping variable (`val`) is decremented first with `addi.n`. Adding the other two values together with the `add.n` instruction generates `newval`. The value of `cur` is moved into `cur1` and `cur` is given the value of `newval`, both with `mov.n` instructions. Finally, `val` is checked to be non-zero with a `bnez.n` instruction.

When the loop is finished, the code returns. Again, the value to be returned is placed in register a2. The loop calculated this value into a6, so the return sequence looks like this:

```
        mov.n   a2,a6
        retw.n
```

The value to return is placed in a2, and the function returns.

2.3 Windowed Calling Convention

Occasionally, routines written in assembly will call C functions, and C functions call assembly routines. To do so, the assembly routines must follow the C calling convention, including passing arguments to functions and returning values.

This section highlights only key points of the calling convention. For the complete specification of the ABI and Software Conventions for Xtensa, refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

2.3.1 Parameter Passing

Arguments are passed in both registers and memory. In general, the first six words of arguments go in the AR register file, and any remaining arguments go on the stack. For a `call`*N* instruction (where *N* is 4, 8, or 12), the caller places the first arguments in registers AR[N+2] through AR[N+7]. (Note that this implies that `call12` can be used only when there are two words of arguments or less; only AR[*N*+2] and AR[*N*+3] can be used when N=12.) The callee receives these arguments in AR[2] through AR[7].

If there are more than six words of arguments, the additional arguments are stored on the stack beginning at the caller's stack pointer and at increasingly positive offsets from the stack pointer. That is, the caller stores the seventh argument word (after the first six words in registers) at [sp + 0], the eighth word at [sp + 4], and so on. The callee can access these arguments in memory beginning at [sp + FRAMESIZE], where FRAMESIZE is the size of the callee's stack frame, typically specified in the `entry` instruction.

For an illustration of the calling convention, consider the following C code.

```
int func( int a, int b, int c, int d, int e, int f,
          int g, int h, int i )
{
        int j;

        j = a + b + c + d + e + f + g + h + i;

        return j;
}
```

This function accepts nine arguments and returns their sum. The compiler generates the following code:

```
.align   4
        .global func
func:
        .frame  a1, 32
.LBB1_func:
```

```
        entry   a1,32
        l32i.n  a10,a1,40

.LBB2_func:
        l32i.n  a8,a1,32
        add.n   a12,a5,a6
        add.n   a11,a2,a3
        l32i.n  a2,a1,36
        add.n   a11,a4,a11
        add.n   a11,a11,a12
        add.n   a8,a8,a7
        add.n   a2,a2,a10
        add.n   a8,a8,a11
        add.n   a2,a2,a8
        retw.n
```

Note that the assembly code ends with a series of `add` instructions into a2. Return values of one word or smaller are returned in register a2.

Registers a2 through a7 contain parameters `a` through `f`, so the function body can simply use the registers as they are. Because all parameters in this example are 32-bit integers, a single parameter fits into a single register.

The remaining parameters are passed through memory. The caller knows the number of parameters to pass, but the callee does not always know (for example, functions with variable numbers of arguments). Thus, the remaining parameters are passed in the caller's stack frame.

The argument to the `entry` instruction in `func` is 32. This means that FRAMESIZE for this function is 32 bytes. The remaining parameters are loaded into unassigned registers so they can be added to the sum at the end of the function. Note that a2 also where the return value is set.

Here are the load instructions only:

```
        l32i.n  a10,a1,40
        l32i.n  a8,a1,32
        l32i.n  a2,a1,36
```

All of the loads are relative to the stack pointer and are outside of the current stack frame, at the lowest addresses of the caller's frame. The caller stores the additional arguments `g`, `h`, and `i` at the lowest addresses in the caller's stack frame.

You can check this by writing a function that calls `func`. Consider the following function:

```
int caller()
{
        return func( 1, 2, 3, 4, 5, 6, 7, 8, 9 );
}
```

The resulting assembly code looks like this:

```
        .type   caller, @function
        .align   4
        .global caller
caller:
        .frame  a1, 48
.LBB1_caller:
        entry   a1,48
        movi.n  a11,2
        movi.n  a12,3
        movi.n  a13,4
        movi.n  a14,5
        movi.n  a15,6
        movi.n  a10,7
        movi.n  a8,9
        movi.n  a9,8
        s32i.n  a9,a1,4
        s32i.n  a8,a1,8
        s32i.n  a10,a1,0
        movi.n  a10,1
        .type   func, @function
        call8   func

.LBB3_caller:
        mov.n   a2,a10
        retw.n
```

Consider the instructions that are setting up the arguments in memory.

```
        movi.n  a10,7
        movi.n  a8,9
        movi.n  a9,8
        s32i.n  a9,a1,4
        s32i.n  a8,a1,8
        s32i.n  a10,a1,0
```

The values of 7, 8, and 9 are being written to the lowest addresses in the caller's stack frame.

This example also illustrates passing arguments through registers. The `func` routine is invoked with the `call8` instruction toward the end of the assembly sequence. The `call8` instruction sets up a logical window rotation that will leave the highest eight registers of the caller as the lowest eight registers of `func`. Thus, `func`'s a2 through a7 are the caller's a10 through a15. As a result, a10 through a15 in `func` have the values of 1 through 6, or the first six parameters.

Of course, not all arguments are 32 bits in size and do not fit snugly into one 32-bit register. The case of smaller arguments is easily examined with a slight change to the previous example. The type `short` is 16 bits to the Xtensa compiler, so we rewrite the example above to use parameters of type `short` instead of `int`.

```
short func( short a, short b, short c, short d, short e,
            short f, short g, short h, short i )
{
        short j;

        j = a + b + c + d + e + f + g + h + i;

        return j;
}
```

This simple change to the C code yields the following assembly code from the compiler:

```
        .type   func, @function
        .align  4
        .global func
func:
        .frame  a1, 32
.LBB1_func:
        entry   a1,32
        l16si   a10,a1,42

.LBB2_func:
        l16si   a8,a1,34
        add.n   a12,a5,a6
        add.n   a11,a2,a3
        l16si   a2,a1,38
        add.n   a11,a4,a11
        add.n   a11,a11,a12
        add.n   a8,a8,a7
        add.n   a2,a2,a10
        add.n   a8,a8,a11
        add.n   a2,a2,a8
        slli    a2,a2,16
        srai    a2,a2,16
        retw.n
```

The assembly generated for this function remains the same except for the `l16si` instruction and the two shift instructions at the end of the code. This shows that integer values smaller than 32 bits (for example, `char` and `short`) are stored in the least-significant portion of the argument word and sign-extended, if signed. The minimum argument size is one word.

The additional instructions handle addition overflow. They ensure that the return value is properly sign-extended. That is, the top 16 bits in the 32-bit word are all zeros or all 1s by shifting left by 16 and then arithmetically right by 16. The callee truncates the return value.

When arguments are larger than 32 bits, parameters are passed somewhat differently. This test case examines both how large values are passed, and how these values interact with smaller values.

```
struct bigstruct {
        int a[16];
};

int func( int a, struct bigstruct b, int c )
{
        return a + b.a[0] + c;
}
```

Note that `bigstruct` is quite large and that it is passed by value as the second argument of `func`. Consider the assembly code that is produced:

```
        .type   func, @function
        .align  4
        .global func
func:
        .frame  a1, 32
.LBB1_func:
        entry   a1,32
        mov.n   a9,a2

.LBB2_func:
        l32i.n  a8,a1,32
        l32i    a2,a1,96
        add.n   a8,a8,a9
        add.n   a2,a2,a8
        retw.n
```

There are two loads. The first is for `b.a[0]`, and the second is for `c` because of the offsets involved. The array reference in `bigstruct` is to element 0. This element is located at the first location in memory. The size of the stack frame for `func` is 32, which means that the first load (the one into a8) is from the first location in the caller's stack frame. The second load is from 64 bytes into the caller's stack frame because the size of `bigstruct` is 64 bytes.

Note that `c` is passed in memory, while `a` is passed in a register. All arguments are passed in registers until the first argument must be put into memory. All subsequent arguments are placed in memory.

For further illustration, consider the following variation of the preceding example:

```
int func( int a, int c, struct bigstruct b )
{
        return a + b.a[0] + c;
}
```

The only change here is that the structure argument is passed as the last parameter. In this case, the compiler generates the following:

```
        .type   func, @function
        .align  4
        .global func
func:
        .frame  a1, 32
.LBB1_func:
        entry   a1,32
        mov.n   a9,a2
        l32i.n  a2,a1,32

.LBB2_func:
        add.n   a2,a2,a9
        add.n   a2,a3,a2
        retw.n
```

Note that there is only one load instruction, whereas the previous example had two. In the first example, `c` was passed in memory. In this example, `c` is passed in a register because of the parameter ordering.

Structures may be passed in registers, provided they fit certain criteria. Additional parameter passing conventions exist for variable arguments and data types defined with the TIE language. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for complete details.

2.3.2 Function Invocation and Return

In addition to passing parameters, the calling convention specifies the location of the return address and stack pointer. It also describes interactions with rotations of the logical window.

Every C function invocation must be made with one of the following call instructions: `call4`, `call8`, `call12`, `callx4`, `callx8`, or `callx12`. In addition to transferring control to the function, these instructions set *Window Increment* in `PS.CALLINC` (that is, the amount by which the logical window rotates) as well as set the callee's a0 register (either the caller's a4, a8 or a12, depending on the call instruction used) to the return address and window increment.

The logical window can be rotated a variable amount by the selection of the call instruction used to invoke a function. The call uses the top two bits of the return address to store a window-increment value. The return instructions rotate the logical window back by that same amount. The return instructions read the window-increment value in the return address to rotate the logical window by the same amount with which it was called. Conversely, the `entry` instruction uses `PS.CALLINC` because it does not know which register (a4, a8, or a12) holds the return address, and therefore the window increment.

Consider the following example:

```
        .text
        .align  4
        .global main
main:
        entry   sp, 48
        call4   func
        call8   func
        call12  func
        retw

        .data
        .align  4
.str:
        .string "%x\n"

        .text
        .align  4
func:
        entry   sp, 32
        movi    a10, .str
        mov     a11, a0
        call8   printf
        retw
```

This example calls `func` three times from `main`: once with a `call4`, once with a `call8`, and once with a `call12`. The function then calls `printf` with its return address. Compiling and running the example yields the following:

```
$ xt-xcc func4.S
$ xt-run a.out
60000c8a
a0000c8d
e0000c90
```

This is the actual disassembly of `main`:

```
60000c84 <main>:
60000c84:       006136          entry   a1, 48
```

```
60000c87:       0000d5          call4   60000c94 <func>
60000c8a:       0000a5          call8   60000c94 <func>
60000c8d:       000075          call12  60000c94 <func>
60000c90:       f01d            retw.n
```

Note that the lower 30 bits of the printed value from the execution of this function are always the return address, and that the top two bits are varied based on the type of call. With the `call4` invocation they are 2‘b01, with the call8 invocation they are 2‘b10, and with the `call12` invocation they are 2‘b11. This is the encoding of how much to rotate the logical window on a return.[2]

After the call is made, the function itself must be written and aligned properly. Although several examples have already been covered, let us discuss the requirements directly.

Every function must be aligned to a 4-byte boundary and must begin with an `entry` instruction. The `entry` instruction does two things. It computes the callee’s stack pointer from the caller’s stack pointer, using the immediate argument as the stack frame size. It also rotates the logical window by the window increment amount in `PS.CALLINC`. The immediate operand to the `entry` instruction must take into account the stack space requirements. A stack frame must be allocated for every function. The size of the stack frame depends on two things: the function’s local variables (if any) and the window save area. The function’s stack space requirements for local variables obviously depend on the function. The following discussion explains the stack-space requirements for the window save area.

Every function must allocate 16 bytes for the base save area. If a function uses any `call8` or `call12` instructions, it must allocate additional space in the stack frame for the extra save area. If it uses any `call8` instruction, it must allocate an additional 16 bytes (32 total); and if it uses any `call12`s, it must allocate an additional 32 bytes (48 total). Table 1 summarizes these requirements.

Table 1. Stack Frame Space for the Window Save Area

Operations Performed in Function	Additional Stack-Frame Space Required
No Calls	16 bytes
`call4`	16 bytes
`call8, call4`	32 bytes
`call12, call8, call4`	48 bytes

A function must allocate stack space for the worst case when it uses a mix of call types. For example, if a function uses both `call4` and `call12` instructions, it must allocate 48 bytes.

2. Windowed calls and returns on Xtensa processors must not cross a 1-GB address boundary because of the destruction of the top two bits of return address by the window call and return mechanisms.

The reasons for the stack-frame requirements listed in Table 1 are much more apparent after gaining an understanding of window overflow exceptions. See Chapter 5 for a detailed discussion of window exceptions.

2.4 Preprocessing Assembly Code

Xtensa Tools Version 7.0 distribution contains several header files that define macros that are often useful in assembly code. These macros conveniently abstract many configuration details, making assembly code more readable, robust, and portable across different Xtensa configurations. However, using these macros requires a preprocessing step prior to assembling the source code.

While assembly files are always assembled with the `xt-as` assembler, it may be useful for you to use the `xt-xcc` compiler driver to invoke the `xt-as` assembler. `xt-xcc` recognizes two different file name suffixes for assembly files. To assemble files ending in .S (upper case), `xt-xcc` invokes the preprocessor before invoking the assembler. To assemble files ending in .s (lower case), `xt-xcc` invokes only the assembler.

It is often useful to give hand-written assembly files a `.S` suffix and use the `xt-xcc` compiler to invoke both the C preprocessor and the assembler to build object code. Alternatively, the `-x assembler-with-cpp` flag to `xt-xcc` has the same effect with other file suffixes. Among other uses, using the preprocessor enables sharing appropriately written header files between assembly and C code. For example, standard Xtensa header files meant to be included by assembly code (or by both assembly and C code) must be included using the C preprocessor mechanism. Additional examples follow.

Although interrupts on the Xtensa architecture have not yet been discussed, consider the following assembly file that implements an addition that is atomic with respect to interrupts. Here are the contents of file *atomic.S*:

```
/*
    atomic.S

        This file implements various atomic operations on data.
        Interrupts are locked to level LOCKLEVEL during these
        operations. Be aware that increasing LOCKLEVEL creates
        additional latency for higher level interrupts.
*/

#define LOCKLEVEL 1

/*
        void atomic_add( int *ptr, int val )
*/
        .text
        .align  4
```

```
        .global atomic_add
        .type   atomic_add, @function
atomic_add:
        entry   sp, 16
        rsil    a7, LOCKLEVEL
        l32i    a4, a2, 0
        add     a3, a3, a4
        s32i    a3, a2, 0
        wsr.ps  a7
        rsync
        retw
```

This simple assembly file creates a single function called `atomic_add`. This function locks interrupts, adds the value passed to the contents of the pointer `ptr`, and then unlocks interrupts.

Consider first the top of this function:

```
/*
    atomic.S

        This file implements various atomic operations on data.
        Interrupts are locked to level LOCKLEVEL during these
        operations. Be aware that increasing LOCKLEVEL creates
        additional latency for higher level interrupts.
*/

#define LOCKLEVEL 1
```

Observe the C-style comments and preprocessor directives, available because this file is an `.S` file. It can be compiled with the following command:

```
$ xt-xcc -c atomic.S
```

`xt-xcc` treats the `.S` file as an assembly file to be passed through the preprocessor. If the assembler is invoked directly, it will fail because it does not recognize the C-style comments and macros.

Following this basic header information, the body of the function itself is declared. The remainder of the file resembles other functions that have already been examined in sections such as *A Simple Example* on page 5.

There is one new item here that bears brief discussion. This function is setting the processor interrupt level to `LOCKLEVEL` to perform certain operations atomically. The processor interrupt level is set with the following instruction:

```
        rsil    a7, LOCKLEVEL
```

The processor interrupt level is restored with the following instruction sequence:

```
wsr.ps  a7
rsync
```

The code in the middle is protected from interrupts of level LOCKLEVEL and below.

Note: The details of this procedure differ slightly for XEA1. For more information, see Appendix C.

2.5 Assembler Relaxations (and Literals)

Some of the most useful instructions in the Xtensa ISA have limited immediate values. Consider the movi instruction. This instruction places an immediate value into a register, but the supported immediate values are limited to 12-bit signed values. The limited immediate size could make writing assembly code quite difficult. Is the following instruction valid?

```
movi a2, CPP_VALUE
```

The answer depends on the definition of CPP_VALUE. The instruction is valid if CPP_VALUE is within the allowable immediate range, but it is invalid otherwise. This makes the use of movi problematic without some assistance from the assembler.

The assembler helps with this issue by doing automatic substitution (or relaxation) of certain instructions. Consider the following:

```
#define CPP_VALUE 2000
        movi    a2, CPP_VALUE
```

Assembling these instructions and disassembling the results gives the following output:

```
$ xt-xcc -c example.S
$ xt-objdump -d example.o

example.o:     file format elf32-xtensa-be

Disassembly of section .text:

00000000 <.text>:
   0:   d0a722        movi    a2, 0x7d0
```

The assembler emitted a movi instruction.

Now let us change the value of `CPP_VALUE` to one outside `movi`'s limited range: 4096 (0x1000). We will also add another instruction.

```
#define CPP_VALUE 0x1000
        movi    a2, CPP_VALUE
        movi    a3, CPP_VALUE + 10
```

We assemble as before, but add the -r flag when dumping the results.

```
$ xt-xcc -c example.S
$ xt-objdump -dr example.o

example2.o:     file format elf32-xtensa-le

Disassembly of section .literal:

00000000 <.literal>:
   0:   00001000 0000100a                                ........
Disassembly of section .text:

00000000 <.text>:
   0:   000021          l32r    a2, 0xfffc0000
                        0: R_XTENSA_SLOT0_OP1   .literal
   3:   000031          l32r    a2, 0xfffc0004
                        3: R_XTENSA_SLOT0_OP1   .literal+0x4
```

The assembler has relaxed both `movi` instructions into `l32r` instructions. The `l32r` loads a 32-bit value from an address that is relative to the address of the instruction itself. The value to load is placed into the object file in a separate section called `.literal`.

The `xt-objdump -r` flag adds information such as 0: R_XTENSA_SLOT0_OP1 .literal to show relocations in the disassembly. The `0:` is the offset of the relocation within the section, and it helps the reader match the relocation information with the correct instruction. The `R_XTENSA_SLOT0_OP1` is an ELF relocation type. This particular type identifies the operand position in the instruction to which the relocation information applies. The symbol plus offset (for example, `.literal+0x4`) identifies the section name plus offset that contains the literal value. This information reveals that the offset is relative to the start of the `.literal` section. There are other, more complex, examples of assembler relaxations, but cataloging all the relaxations is beyond the scope of the current discussion.

The important point to remember is that the assembler may change the instructions that are written for two reasons. First, the assembler may change the actual code to make writing code easier. The `movi` syntax is easier than the `l32r` syntax. In fact,

```
movi  register_name, symbol_name
```

is the preferred method to load an address on Xtensa processors, even though the assembler will relax this syntax to an `l32r` and literal every time. The second reason the assembler may change your code is to align branch targets.

Programmers should remember three main points:

- Convenient syntax (such as a `movi` with an immediate value outside of the valid range) can be used even when the definition of the instruction itself would not allow its use.
- The disassembly of an object file may not match exactly the file from which it was assembled.
- The assembler provides a number of means to disable relaxations (see discussions on the underscore prefix, `.begin` and `.end` assembler directives, and command-line parameters in the *GNU Assembler User's Guide*).

2.6 Locating Code at Fixed Locations

Sometimes code needs to go at a fixed location. For example, a reset handler should reside at the reset vector, and a user exception handler should reside at the user exception vector.

You can locate code at a particular location by using sections in a linker script. You can ensure that sections are declared in such a way that they may use the correct assembler directives to put code and data in those sections. For example, the linker support packages come with linker scripts that are tailored to a particular processor configuration. These linker scripts declare a separate section for each vector in the processor. Refer to the *Xtensa Linker Support Packages (LSPs) Reference Manual* for more details.

As an example of how to put code at a fixed location, we create a useless kernel exception vector that simply returns from the exception:

```
        rfe
```

We might think that the file to do this would look something like this:

```
/*
  KernelExceptionVector

  This is a stub implementation of a kernel exception vector.
*/

        rfe
```

Indeed, compiling such a file creates an object file; and the data in that object file looks correct. Consider the output of `xt-objdump`:

```
kernel.o:     file format elf32-xtensa-le

Disassembly of section .text:

00000000 <.text>:
   0:   003000          rfe
```

However, notice that the `rfe` instruction is in the `.text` section.

The `nort` linker support packages are intended for development of software that does not use the default Xtensa runtime system. The default Xtensa run-time system supplies code for all of the vectors. Hence, we use the `nort` linker support package (LSP) from here onward.

Examining the linker script for the `nort` LSP for a particular processor configuration shows it contains declarations for the following sections, among others:

```
.ResetVector.text
.DebugExceptionVector.text
.srom.text
.KernelExceptionVector.text
.UserExceptionVector.text
.Level2InterruptVector.text
.WindowVectors.text
.sram.text
.text
```

With the exception of `.sram.text` and `.srom.text` (discussed later), these section names are self-explanatory. Code that must reside at the reset vector goes in the `.ResetVector.text` section. For our example, code that needs to go into the kernel exception vector goes in the `.KernelExceptionVector.text` section, which is then placed in the proper location by the linker.

Doing this is simple with the use of a `.section` directive. This directive tells the assembler to place the generated code into a particular section.

```
/*
  KernelExceptionVector

  This is a stub implementation of a kernel exception vector.
*/

        .section .KernelExceptionVector.text, "ax"
        rfe
```

First, notice that the .section directive includes not only the name of the section, but some additional information. The additional information describes properties of the section. In this case, the section is being marked as having executable code (x) and as being allocatable (a). The important thing to know is that sections of executable code must be marked as "ax" (including quotes).

Now that we include the .section directive, the output of xt-objdump looks like this:

```
kernel.o:     file format elf32-xtensa-le

Disassembly of section .text:
Disassembly of section KernelExceptionVector.text:

00000000 <.KernelExceptionVector.text>:
   0:   003000          rfe
```

Note that the rfe instruction is now placed in the KernelExceptionVector.text section.

What happens if the code at the kernel exception vector requires literals? The top of the linker script declares the memories that are to be used. These memories are not necessarily real physical memories, but they simply partition the address space into different areas. However, the declaration of these memories does limit the size of the sections that go into them.

Often the vector areas are very small. Consider the following memory definition for the area to receive the kernel exception vector text data:

```
  KernelExceptionVector.text_seg : ORIGIN = 0x60000300, LENGTH = 0x38
```

The size is set to a mere 56 bytes, but the entire kernel exception handler will probably not fit in 56 bytes. Therefore, the code at the KernelExceptionVector dispatches control to a different location where the real work of the kernel exception vector will be handled. The dispatch code looks like this:

```
/*
  KernelExceptionVector

  This implements the kernel exception vector and transfers
  control to the KernelExceptionHandler.
*/

        .section .KernelExceptionVector.text, "ax"

_KernelExceptionVector:
        wsr.excsave1  a3
        movi    a3, _KernelExceptionHandler
        jx      a3
```

Notice that this code uses the `movi` instruction to load the address of the `_KernelExceptionHandler`. The assembler will relax this `movi` into an `l32r` instruction and create the corresponding literal, as the following dump shows.

```
kernel.o:     file format elf32-xtensa-be

Disassembly of section .literal:

00000000 <.literal>:
   0:   00000000                                        ....
                        0: R_XTENSA_32  _KernelExceptionHandler
Disassembly of section .text:
Disassembly of section .KernelExceptionVector.text:

00000000 <_KernelExceptionVector>:
   0:   03d131          wsr.excsave1    a3
   3:   130000          l32r    a3, fffc0004
<_KernelExceptionVector+0xfffc0004>
                        3: R_XTENSA_SLOT0_OP1   .literal
   6:   0a3000          jx      a3
```

Notice that the literal has been placed in the `.literal` section. This can be problematic because the `l32r` instruction has an offset limited to 256 KB and the vector might be located far from the main code. For this reason, literals for vectors must be in special literal sections that the linker script locates close to the vector itself.

Looking again at the linker script, we find that there is a special `.KernelExceptionVector.literal` section adjacent to the `.KernelExceptionVector.text` section. We must tell the assembler that, when it creates literals, it should place those literals into the `.KernelExceptionVector.literal` section rather than the default .literal section. This is done with the `.begin literal_prefix` statement and looks like this:

```
/*
  KernelExceptionVector

  This implements the kernel exception vector and transfers
  control to the KernelExceptionHandler.
*/

        .section .KernelExceptionVector.text, "ax"
        .begin   literal_prefix .KernelExceptionVector

_KernelExceptionVector:
        wsr.excsave1  a3
        movi    a3, _KernelExceptionHandler
        jx      a3

        .end    literal_prefix
```

Note that the `.begin literal_prefix` statement requires a matching `.end literal_prefix` statement and that it takes the base name of the section. (The use of `literal_prefix` does imply that the naming of sections follows certain conventions.) All literals generated by the assembler between the `.begin literal_prefix` and `.end literal_prefix` statements will be placed into the `.KernelExceptionVector.literal` section.[3]

The resulting object disassembly demonstrates this point.

```
kernel.o:     file format elf32-xtensa-be

Disassembly of section .KernelExceptionVector.literal:

00000000 <.KernelExceptionVector.literal>:
   0:   00000000                                 ....
                        0: R_XTENSA_32  _KernelExceptionHandler
Disassembly of section .text:
Disassembly of section .KernelExceptionVector.text:

00000000 <_KernelExceptionVector>:
   0:   03d131          wsr.excsave1    a3
   3:   130000          l32r    a3, fffc0004
<_KernelExceptionVector+0xfffc0004>
                        3: R_XTENSA_SLOT0_OP1
.KernelExceptionVector.literal
   6:   0a3000          jx      a3
```

We can write assembly code, and we can locate that assembly at various places in the address space. We now have the tools to start writing the code for the Xtensa processor vectors.

2.7 Writing Efficient Assembly Code

Most architectures have some rules of thumb for writing efficient assembly-language programs. This section lists some rules for the Xtensa architecture.

2.7.1 Branch Instructions

Rule of Thumb: A branch-taken condition is more expensive than a branch-not-taken condition. Optimal assembly code will anticipate a branch-not-taken condition for the common case.

---

3. The `literal_prefix` directive applies to literals referenced from PC-relative L32R instructions. It has no effect for absolute literals. See the Extended L32R Option in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* and the discussion of absolute literals and PC-relative literals in the *GNU Assembler User's Guide*.

The exact penalty varies between implementations of Xtensa processors. The primary delay factor is pipeline length. For example, Xtensa V (T1050) Xtensa processors have a five-stage pipeline. A branch-taken condition results in a two-cycle delay because the processor computes the target address in the E (or third) pipeline stage. An Xtensa processor configured with a seven-stage pipeline has a three-cycle penalty.

To illustrate how assembly code can anticipate the branch-not-taken condition for the common case, consider a function that accepts a single argument. This argument is a memory address and is usually word-aligned. However, the function must also handle cases of unaligned memory addresses. In testing for unaligned addresses, we want to branch rarely and "fall through" branch instructions for the common case. Optimizing for the common case, this function could begin as follows:

```
bbsi.l  a2, 0, unaligned_word_1   // if addr is 1 mod 2
bbsi.l  a2, 1, unaligned_word_2   // if addr is 2 mod 4
// addr is aligned on word boundary
```

Another possible delay factor is the branch's target instruction's alignment. If the target instruction is at an address such that it crosses a fetch boundary, the branch will take an extra cycle. However, most assembly writers can ignore this issue because the assembler aligns most branch targets automatically. It does so by converting nearby wide instructions to their narrow forms or narrow instructions to their wide forms such that the branch target is aligned properly.

2.7.2 Scheduling Instructions by Hand

Rule of Thumb: Anticipate pipeline bubbles and replace them with useful work.

The processor inserts pipeline bubbles if an instruction tries to use data that has not yet arrived (that is, register interlock). For example, load instructions on five-stage processors require at least one extra cycle before the data becomes available. Suppose we want to set variable `foo` to 1 and increment variable `bar`. An inefficient way is shown below.

```
movi    a2, 1
movi    a3, foo         // xt-as turns movi into l32r
// bubble
s32i    a2, a3, 0       // foo = 1
movi    a3, bar         // xt-as turns movi into l32r
// bubble
l32i    a2, a3, 0       // read bar
// bubble
addi    a2, a2, 1       // increment bar
s32i    a2, a3, 0       // store bar

// total cycles = 10
```

Intelligent hand-scheduling of instructions can replace pipeline-bubble cycles with useful work. A better way is shown below.

```
movi    a4, bar         // get &bar
movi    a3, foo         // get &foo
l32i    a5, a4, 0       // read bar
movi    a2, 1
s32i    a2, a3, 0       // foo = 1
addi    a5, a5, 1       // increment bar
s32i    a5, a4, 0       // store bar

// total cycles = 7
```

2.7.3 Scheduling Instructions with the Assembler

`xt-as` can schedule instructions within basic blocks to avoid pipeline bubbles and to bundle instructions automatically for processors that support FLIX. Scheduling is off by default. Refer to the *GNU Assembler User's Guide* for more details.

2.7.4 Avoiding Window Overflows

Rule of Thumb: Always use lowest-numbered registers first to avoid triggering window overflow exceptions.

As the logical window rotates and wraps around the physical register file, window overflows occur to preserve context from other functions. Upon returning from an overflow exception, the registers are available for use by the current function. (See Chapter 5 for more details.)

Window overflow exceptions happen only when necessary, and they always spill one, two, or three window panes. Upon entering a function, registers a0..a3 are guaranteed to be available for use because the `call` instruction writes the return address into a0 and causes an overflow, if needed. The subsequent `entry` instruction writes the new stack pointer into a1, so the entire window pane a0..a3 is available.

If your function needs only two working registers, use a2 and a3. You are then guaranteed that a window overflow will not happen during the course of that function. Using any of a4..a15 might cause a window overflow. If your function needs three working registers, select a2, a3, and one of a4..a7. Because a4 through a7 belong to the same window pane, a4, a5, a6, and a7 have the same probability of triggering a window overflow. Window panes a8..a11 and a12..a15 have increasing probabilities of triggering a window overflow.

2.8 Other Tips and Tricks

Programmers collect tips and tricks for particular computer architectures they often use. The following subsections explain some of the most useful ones.

2.8.1 Obtaining the Program Counter

The Xtensa architecture does not provide an instruction that directly reads the program counter. To obtain the current program counter, use the following code:

```
        mov     a7, a0  // preserve a0 in a temp register
        _call0  2f
1:      .align  4       // call targets must be 4-byte aligned
2:      mov     a2, a0  // save the PC in a2, not a0
        mov     a0, a7  // restore a0
        // a2 contains the value of label 1
```

The following code also works in many cases:

```
        movi    a2, 1f
1:
        // a2 contains the value of label 1
```

The first method has the following advantages:

- It is position independent code. Literals (as generated in the second method) are not position independent.
- It does not load from memory. With the second method, the assembler will convert the `movi` to an `l32r`. Memory subsystem delays may or may not be a factor. However, it does execute a branch, with a minimum 2 cycle penalty.

The second method has the following advantages:

- Shorter, quicker, simpler–that is, as long as factors addressed by the first method are not relevant.

2.8.2 Should I use `bbsi` or `bbsi.l`?

`bbsi.l` is really an assembler macro, not a machine instruction. It makes portable assembly-language programs more readable by handling endian-ordering differences. On little-endian processors, `bbsi.l` is identical to `bbsi`. On big-endian processors, the assembler converts the encoded immediate value to 31-imm.

Consider this code segment:

```
        bbsi.l  a2, 0, 1f       // if addr is 1 mod 2
        bbsi.l  a2, 1, 2f       // if addr is 2 mod 4
```

It is identical to the code segment that follows.

```
#if __XTENSA_EL__
        // Need little-endian bit numbering
        bbsi    a2, 0, 1f       // if addr is 1 mod 2
        bbsi    a2, 1, 2f       // if addr is 2 mod 4
#else
        // Need big-endian bit numbering
        bbsi    a2, 31, 1f      // if addr is 1 mod 2
        bbsi    a2, 30, 2f      // if addr is 2 mod 4
#endif
```

However, the former segment is more gentle on a programmer's eyes.

Programmers should always prefer `bbsi.l` to `bbsi`, even if there are no plans to change the processor's endian order. Portable and more-readable source code is always better.

This same story applies to the inverse instructions `bbci` and `bbci.l`.

2.8.3 Abstracting Configuration Options Using `coreasm.h`

The include file `<xtensa/coreasm.h>` has some useful assembly macros that bring improved portability and readability to assembly-language source code. They abstract functionality whose optimal implementation is affected by processor-configuration options.

Supported functionality includes, among many other things:

- Finding the most- or least-significant bit set in a register
- Looping constructs
- Conditionally read-and-set the interrupt level
- Window-spill functionality

The starkest example of portability benefits is the looping-construct macros, which use the zero-overhead loop instructions, if available. Otherwise, these macros use more verbose decrement and branch instructions. If assembly source explicitly used zero-overhead loop instructions, that code would not assemble on configurations without zero-overhead loops. However, the looping-construct macros would insert equivalent functionality instead, and the assembly would succeed.

See the *Xtensa System Software Reference Manual* for official documentation on these macros.

2.8.4 The GNU Assembler for Xtensa Processors

The GNU assembler has some features unique for Xtensa processors. Assembly-language programmers should carefully review the *GNU Assembler User's Guide* published by Cadence for a discussion of Xtensa-specific features. These features include unique command-line options, assembler directives, instruction relaxations, automatic instruction alignment, and optimizations. They are useful for creating robust, readable assembly-language source code.

3. Xtensa Exception Architecture

This chapter describes the current Xtensa Exception Architecture 2 (XEA2). Appendix C describes the older and somewhat less flexible XEA1. The *Xtensa Instruction Set Architecture (ISA) Reference Manual* defines all details of both variants. This discussion highlights only key points that affect programmers and assumes familiarity with the ISA.

Xtensa LX, Xtensa 6, and later processors support XEA2 only. The Xtensa Exception Architecture 2 (XEA2) has provisions to implement a ring protection model and is slightly more flexible than the Xtensa Exception Architecture 1.

See Appendix C.1 for a discussion of the exact differences between XEA1 and XEA2.

3.1 The PS Register

An understanding of the Miscellaneous Program State Register (PS) is necessary before considering a specific exception example. PS contains miscellaneous fields that are grouped together so they can be saved and restored easily for interrupts and context switching. Figure 1 shows its layout, and Table 2 describes its fields.

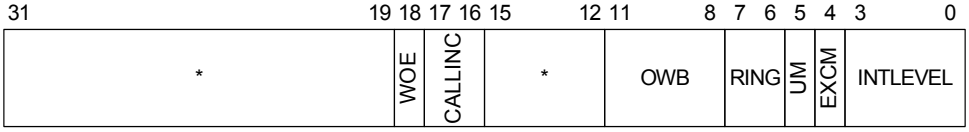

Figure 1. PS Register Format—Xtensa Exception Architecture 2

Table 2. PS Register Fields—Xtensa Exception Architecture 2

Field	Width (bits)	Definition
INTLEVEL	4	Interrupt-Level Disable This field is used to compute the current interrupt level (CINTLEVEL) of the processor via CINTLEVEL ← max (PS.INTLEVEL, PS.EXCM*EXCMLEVEL). Interrupts are disabled at CINTLEVEL and below. To enable an interrupt of level N, CINTLEVEL must be less than N, and the INTENABLE bit for that interrupt must be set to 1. CINTLEVEL is 0 during normal operation if interrupts are enabled. When taking an exception, the processor sets PS.EXCM to 1 and leaves other PS fields unmodified. Therefore, CINTLEVEL is EXCMLEVEL (a configuration parameter), which results in the disabling of interrupts at level EXCMLEVEL and below. The processor sets INTLEVEL to 15 on reset.

EXCM	1	Exception Mode
		0 → normal operation
		1 → exception mode
		The processor sets PS.EXCM to 1 on reset. On exceptions, the processor sets PS.EXCM to 1 and leaves all other PS fields unmodified. PS.EXCM is 1 in high-priority interrupts primarily to disable loopback for proper security. The processor clears PS.EXCM on instructions rfe, rfwo, and rfwu. Instruction rfde does not clear PS.EXCM, but leaves it set to 1. WSR and XSR to PS can also modify PS.EXCM. When set, PS.EXCM overrides the values of certain other PS fields. In particular:
		`CRING ← if PS.EXCM then 0 else PS.RING`
		`CINTLEVEL ←max(PS.INTLEVEL,PS.EXCM*EXCMLEVEL)`
		`CWOE ← if PS.EXCM then 0 else PS.WOE`
		`CLOOPENABLE ← PS.EXCM == 0`
		CRING, CINTLEVEL, CWOE, and CLOOPENABLE are the processor's current ring level, interrupt level, window-overflow enable state, and loop-enable state, respectively.
		Note that PS.EXCM carries major implications to exception handling. When set, the processor operates at the kernel ring level. It masks interrupts at level EXCMLEVEL and below, disables window overflows, and disables zero-overhead loops.
		PS.EXCM overrides certain PS fields while preserving the original PS-field values for software. Although the processor modifies PS.EXCM on exceptions, software can deduce the state of PS.EXCM before the exception. If PS.EXCM was set before the exception, the processor transfers control to the DoubleExceptionVector (more on this later). Otherwise, it transfers control to the UserExceptionVector or KernelExceptionVector, depending on the PS.UM bit. For high-priority interrupts, the previous value of PS.EXCM is saved in the associated EPSn register.

Table 2. PS Register Fields—Xtensa Exception Architecture 2 (continued)

Field	Width (bits)	Definition
UM	1	User Vector Mode 0 → kernel vector mode 1 → user vector mode Conventional use of the user vector mode bit is for controlling stack switching on exceptions. Kernel vector mode indicates the processor is using the kernel exception stack. User vector mode indicates that a different stack is in use, and exceptions will need to switch stacks. Note that hardware does not enforce conventional use, but provides this bit as a convenience to operating systems. Operating-system implementations are free to redefine the meaning of this bit. They may even ignore it entirely, provided that they capture exceptions at the appropriate vector. An alternate view of the UM bit is that it is solely a vector selector bit. If zero, exceptions are transferred to the KernelExceptionVector. If one, exceptions are transferred to the UserExceptionVector. The processor sets UM to zero on reset and never modifies it thereafter.
RING (If RingCount>1)	$\lceil \log_2 (\text{RingCount}) \rceil$	Privilege Level The current privilege level, CRING, is defined as CRING ← if PS.EXCM then 0 else PS.RING. When CRING == 0, privileged operations are allowed; otherwise, a Privilege exception occurs when the processor tries to execute a privileged instruction. With the MMU Option, CRING also controls memory access. The processor sets RING to zero on reset. If RingCount is 1, CRING is always zero, and the processor always runs in privileged mode.
OWB (Windowed Registers Option)	NAREG/4	Old Window Base The value of WindowBase before window overflows and underflows.
CALLINC (Windowed Registers Option)	2	Call Increment call4, call8, and call12 instructions set CALLINC to the window increment amount. entry instructions use CALLINC to rotate the logical window.
WOE (Windowed Registers Option)	1	Window Overflow-Detection Enable 0 → overflow detection disabled 1 → overflow detection enabled This field is used to compute the current window overflow enable of the processor: CWOE ← if PS.EXCM then 0 else PS.WOE
*		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior.

3.1.1 Exception Mode and Masking Interrupts

There is a subtle relationship between exception mode and interrupts. The processor is in exception mode whenever PS.EXCM is 1. The processor enters exception mode on reset, by taking an exception, by taking an interrupt, or by software writing a 1 to the EXCM field of PS. Exception mode affects the processor's current interrupt level, which is computed with the equation

```
CINTLEVEL ← max(PS.INTLEVEL, PS.EXCM * EXCMLEVEL)
```

While the processor is in exception mode, interrupts at level EXCMLEVEL and below are masked, independent of the value in PS.INTLEVEL. This processor functionality adds importance to the goal that exception handlers quickly remove the processor from exception mode, perhaps after saving some processor state.

3.1.2 Exception Semantics

The following pseudo-code describes the exception semantics for non-window exceptions. Of particular interest is the effect of PS.EXCM sending control to the DoubleExceptionVector.

```
EXCCAUSE ← cause
if PS.EXCM then
        DEPC ← PC
        nextPC ← DoubleExceptionVector
elseif PS.UM then
        EPC[1] ← PC
        nextPC ← UserExceptionVector
        -- PS.UM is unchanged
else
        EPC[1] ← PC
        nextPC ← KernelExceptionVector
endif
PS.EXCM ← 1
```

3.1.3 Exception Vectors

XEA2 provides three vectors for exceptions and level-one interrupts:

- UserExceptionVector,
- KernelExceptionVector, and
- DoubleExceptionVector (exceptions only).

An exception occurring while PS.EXCM is 1 (such as, during dispatch or exit of another exception or interrupt) is a *double exception*, and causes the processor to jump to the DoubleExceptionVector (note that level-one interrupts are masked by PS.EXCM=1, so they cannot cause a double exception). Otherwise, when an exception or level-one interrupt condition occurs and PS.EXCM is 0, the processor uses PS.UM to select the appropriate vector at which to transfer control. If PS.UM is one, the processor jumps to the UserExceptionVector; otherwise, it jumps to KernelExceptionVector.

Note that PS.UM is significant only when PS.EXCM is zero.

The DoubleExceptionVector handles cases where legitimate exceptions occur during exception handling while the processor is in exception mode. For example, configurations that have MMUs may have exception handlers that legitimately generate additional exceptions. Consider a virtual-memory system, an operating system that supports paging, and user tasks with their own virtual-memory spaces. For example, when a window-overflow exception occurs, the processor sets PS.EXCM to 1 and jumps to the appropriate window-overflow handler. The handler tries to dump register contents to memory, but the user task's stack has been paged out. The write operation will cause a legitimate MMU-related exception that needs to be handled. With PS.EXCM set to 1, the processor will jump to the DoubleExceptionVector, the handler will remap the user task's stack, and return to the window-overflow handler. The window-overflow handler will retry the write instruction, which now succeeds, and the window-overflow handler returns to the user task code.

3.1.4 Windowed Code and Xtensa Processor State

Any code that invokes the windowed-register mechanisms is called *Windowed Code*. This section describes the required processor state to execute windowed code. Normally, C code is windowed code, and hand-written assembly code certainly could be. Although the XCC compiler supports an alternate ABI that does not use the windowing mechanism, called the "Call0 ABI", this ABI is lower performance and should be used only when specific needs prevent using the Windowed ABI.

When the processor transfers control to a vector (for example, UserExceptionVector), the processor state is insufficient for executing windowed code. Although many exception handlers written in assembly do not use windowed code, sometimes a handler needs to call windowed code (for example, if that code is written in C). In such cases, some preliminary assembly code needs to modify the processor state to make it suitable for windowed code. The following conditions must be met.

PS.EXCM must be 0 and PS.WOE must be 1. PS.EXCM disables the current window-overflow enable of the processor (CWOE ← if PS.EXCM then 0 else PS.WOE). On Xtensa IV (T1040) processors or later, executing a windowed-register instruction (for example, callN, callxN, entry, or retw) when PS.EXCM==1 or PS.WOE==0 will result in an Illegal Instruction exception.

The WINDOWSTART and WINDOWBASE special registers must be coherent. That is, the expression ((1 << WINDOWBASE) & WINDOWSTART) must be nonzero, and WINDOWSTART must reflect contiguous window frames of appropriate size. (It is often easiest to set only one bit in WINDOWSTART and show only the current window frame.) In certain earlier processor releases, these registers are undefined on processor reset, so reset handlers must set them. On Xtensa IV (T1040) processors or later, executing a windowed-register instruction with an incoherent WINDOWSTART and WINDOWBASE may result in an Illegal Instruction exception. See Chapter 5 for more details on these special registers.

General-purpose register a1 must contain a valid stack pointer. The ABI and Software Conventions for Xtensa processors designate a1 as the stack pointer. While executing windowed code, the possibility of window-overflow and window-underflow exceptions exists. The window-exception handlers assume a valid stack pointer. Additionally, valid stack frames must exist in the stack space in memory. Section 5.2 on page 76 details the Xtensa stack layout. Debuggers for Xtensa processors consider a return value of zero (in a0) as an indicator of the last valid stack frame in a trace back of function calls.

Application code is conventionally executed at interrupt level zero, although setting PS.INTLEVEL to zero is not required. For example, the work-horse C function of a level-one interrupt handler may execute at PS.INTLEVEL==1 to mask all other level-one interrupts.

Unlike reset handlers, exception handlers must consider the context in which control arrived. Exception handlers are likely to have meaningful state in general-purpose and special registers that they must also preserve before executing windowed code, and to restore before returning from the exception. The underlying kernel, run-time, or low-level software design, particularly stack usage, usually has the greatest impact on deciding what state needs preservation. For example, if all software uses only a single stack, exception handlers likely need to preserve the current logical window and various special registers before calling windowed code. If an operating system and user tasks do not share stack space and their stacks cannot be linked somehow, exception handlers likely must save the entire physical register file to memory and save various special registers before calling windowed code.

3.2 Low-Priority Interrupts (a.k.a. Level-One Interrupts)

The Xtensa ISA packages level-one interrupts as a general exception type. Therefore, the processor routes level-one interrupts through the usual UserExceptionVector or KernelExceptionVector exception vectors (depending on the value PS.UM). The EXCCAUSE register will identify the exception as a level-one interrupt (Level1InterruptCause), and software handlers can respond accordingly.

3.3 Medium-Priority Interrupts

The Xtensa LX and Xtensa 6 processors support a configuration option that allows designers to define an exception-mode level, or `EXCMLEVEL`. This `EXCMLEVEL` defines the highest interrupt priority level that `PS.EXCM` masks.

Efficient C-callable interrupt handlers can be defined for interrupt priority levels up to EXCMLEVEL. For example, if EXCMLEVEL is 3, interrupt priority levels 2 and 3 are medium-priority interrupts; efficient C-callable interrupt handlers can be defined for interrupts at priority levels 1, 2, and 3; and interrupt priority levels 4 and above are still high-priority interrupts (see Section 3.4) and should be written in assembly.

Xtensa processors do not route medium-priority level interrupts through the usual `UserExceptionVector` or `KernelExceptionVector`. Each medium-priority interrupt level has its own vector address, exception program counter register (`EPCn`), and exception program state register (`EPSn`), where n corresponds to the priority level, 2 through 15.

3.4 High-Priority Interrupts

Some Xtensa processors also have high-priority interrupts, which are any interrupt with a priority level above `EXCMLEVEL`. Xtensa processors do not route high-priority interrupts through the usual `UserExceptionVector` or `KernelExceptionVector`. Each high-priority interrupt level has its own vector address, exception program counter register (`EPCn`), and exception program state register (`EPSn`), where *n* corresponds to the priority level, 2 through 15.

3.5 Can I Write High-Priority Interrupt Handlers In C?

Technically you can write high-priority interrupt handlers in C, but Cadence strongly discourages it. XTOS can dispatch high-priority handlers written in C. However, this convenience brings a significant performance penalty of about 100-200 cycles[4] on dispatch and nearly as much on handler exit, depending on processor configuration. This is largely due to the need to save and restore the entire address register file and various other exception related states. The impact is potentially smaller when implementing the handler using the Call0 ABI, but XTOS is not optimized for its use, and the Call0 ABI has performance and size implications when used by the main application.

4. Not counting any cache, memory busy signals, external stalls, or similar effects that can increase this number significantly.

High-priority interrupts are designed to service peripherals requiring very fast, low-latency response to an interrupt. If there is time to set up the environment to run C code, a high-priority is not needed. A medium-priority interrupt can be configured and used instead. Alternatively, a level-one interrupt can also be used instead (with software prioritization if needed).

High-priority interrupt handlers should be written in assembly language. These fast handlers should avoid instructions that invoke windowed-register mechanisms. They should generally preserve any processor state they need to destroy (for example, general-purpose registers), and should restore the state before returning.

See Section 3.1.4 or Appendix C.3.5 for a discussion of the overhead required to establish the environment for running C code.

4. Resetting the Processor

The processor begins execution at the reset vector when its hardware reset signal is released. The processor fetches code from the reset vector first. For that reason, we begin our discussion of actual runtime code with how to create a reset vector.

4.1 Reset State Requirements

At reset, only a minimum set of registers in the processor have defined values. The reset vector moves the processor from this limited known state to a more useful known state. For example, the reset vector cannot be written in C because certain registers have not been initialized, or do not have the correct value for this purpose. Most reset vectors bring the processor to a state where C code may be executed safely. They do this by initializing various registers and other processor states.

Not all registers need to be set to a known value to boot the processor and start running application software (and in fact, many are not). For example, several shift instructions use the SAR register to determine the shift amount, and SAR is undefined at reset. But the SAR does not need to be initialized by the reset code because any software that does a variable shift always sets the SAR first. This is reflected in the ABI. Many other registers are like this.

Let's examine what registers and other processor states generally need to be defined before we can start executing a program. For example, before one can call `main()`. Note that this is what most software expects to be initialized. Although related, this is different from what the processor initializes via hardware reset. The *Xtensa Instruction Set Architecture (ISA) Manual* defines the hardware reset values of architectural state. A particular Xtensa processor implementation may additionally reset registers which the ISA documents as undefined at reset; software should avoid relying on such behavior.

Most software expects the following processor registers and states to be initialized when present (according to the processor's configuration options):

- PC — The processor starts fetching from the reset vector. Always initialized by hardware reset.
- PS — Each field of `PS` is described separately. `PS.OWB` and `PS.CALLINC` do not need to be initialized to any particular value. See Chapter 3 for details about the `PS` register.

▪ PS.EXCM	To allow window exceptions, interrupts, and zero-overhead loops, this bit must be zero. Hardware resets it to one. See Chapter 3 for more details on this `PS` register bit.
▪ PS.UM	The OS or runtime determines the useful initial value of `PS.UM`, according to how it implements the user and kernel vectors. Existing runtimes typically initialize it to one. Hardware resets it to zero.
▪ PS.RING	Must be zero to allow executing privileged instructions. Protected operating systems that set it otherwise are beyond the scope of this discussion. Hardware resets it to 0.
▪ PS.WOE	This bit controls window overflow exceptions. It must be set according to which software ABI will be used by application-level code: 1 for the Windowed ABI, or 0 for the Call0 ABI. Hardware resets it to zero.
▪ PS.INTLEVEL	This field must be zero to enable interrupts of all levels. Individual interrupts are still controlled using the `INTENABLE` register. Hardware resets it to 15 (0xF) to avoid having to reset `INTENABLE`, `ICOUNTLEVEL`, and `DBREAK`*n* registers.
▪ WINDOWBASE and WINDOWSTART	These must be initialized for software configured to use the Windowed ABI. Typical values are 0 and 1 respectively, although *n* and 2^n respectively also work for valid values of *n* in `WINDOWBASE`. The ISA does not define their reset value. See Chapter 5 for more details on the Windowed ABI, and Section 4.6 for the reset sequence.
▪ a0 and a1	These must be initialized to satisfy the ABI. They are the return `PC` and stack pointer. Before calling the first function, `a0` must be zero to terminate the call stack. And `a1` must point to the base of an appropriately sized stack. Their reset value is undefined.
▪ INTENABLE	This register masks individual interrupts. Each bit must be initialized to one only if the corresponding interrupt source is initialized and a handler is ready to take the interrupt;

otherwise the bit must be zero to disable it. A typical runtime or OS initializes the entire register to zero, and the application sets individual bits as it sets up each interrupt. The ISA does not define a reset value. See also Section 4.3.

- INTERRUPT

  This register reports pending interrupts. The ISA does not define a reset value. So each bit must usually be cleared before allowing the interrupt to be taken, to avoid a spurious interrupt. This is unnecessary for a level-triggered interrupt, whose `INTERRUPT` bit simply reflects the state of the external pin.

- ICOUNTLEVEL

  This register must be initialized to zero to avoid an `ICOUNT` debug exception nearly 2^{32} instructions after zeroing `PS.INTLEVEL` (or lowering it below `DEBUGLEVEL`).

- IBREAKENABLE

  This register must be initialized to zero to avoid randomly getting a debug exception when `PC` matches undefined `IBREAKA`*n* registers while `PS.INTLEVEL` is set below `DEBUGLEVEL`. Hardware resets it to zero.

- DBREAKC*n*

  This register must be initialized to zero to avoid randomly getting a debug exception when any load or store address matches undefined `DBREAKA`*n* registers while `PS.INTLEVEL` is set below `DEBUGLEVEL`. The ISA does not define a reset value for it. See also Section 4.5.

- LCOUNT

  This register must be initialized to zero to avoid random jumps when `PC` matches an undefined `LEND` while `PC.EXCM` is zero. It is part of the zero-overhead loop mechanism. Hardware resets this register to zero.

- CCOUNT

  This register counts processor clock cycles, and is included with the timer option. It is often useful to initialize this register to zero. Some software might use it as a measure of time elapsed since startup or reset. The ISA does not define a reset value for it. See also Section 4.4.

- LITBASE — For software configured to use extended L32R mode, this register must be initialized with the location of literals (`.lit4` section). Otherwise it must be initialized to zero to select PC-relative literals. Hardware resets this register to zero.
- PTEVADDR — This MMU register points to the page table in virtual memory. If no page table is used, it must point to an unmapped area so that accesses to unmapped pages predictably return an error rather than result in random behavior. This is often done by initializing it to zero (static mappings begin at address `0xD0000000`). Its reset value is undefined.
- RASID — This MMU register must be initialized with separate values for each of its 8-bit ASID fields. This avoids multihit exceptions due to a single access matching multiple ASID fields. Hardware resets it to `0x04030201`, satisfying this requirement.
- ITLBCFG and DTLBCFG — These MMU registers must be initialized with the correct page size(s) if the variable page size MMU TLB entries are used. This is often the case in processors with a local memories and a full MMU, where variable page entries are used to map local memories at reset. In this case, hardware resets these registers appropriately. Otherwise, these registers have no defined reset value.
- CPENABLE — This register must be initialized with all bits set to allow all coprocessors to be accessed without causing exceptions. Alternatively, in a multithreaded environment, this register is usually initially cleared and subsequently dynamically managed by the OS as each thread accesses coprocessors. For more details, see Section 8.2 on page 142.
- FSR and FCR — These floating point unit registers must be initialized to zero to provide a predictable floating point rounding mode.
- Instruction and data caches — The caches must be reset, that is, all entries must be marked as invalid and unlocked, so that cached accesses don’t pull random

	contents from the cache or push random cache contents to memory. It is possible to avoid this step by accessing memory strictly in cache-bypass mode, however the performance penalty is significant. Cache initialization is unavoidable on processors with a full MMU: vectors usually point to a virtual address statically configured to cached mode. Cache state is undefined at reset. See also Section 4.7.
▪ TLB entries or CACHEATTR[5]	Memory mappings and access modes must be initialized according to the processor's system memory map and application. Hardware resets them so as to safely fetch reset vector code in bypass-cache mode. See also Section 4.7.
▪ Other registers	A specific OS or runtime design may need to initialize other registers as well. For example, some runtimes initialize `EXCSAVE`*n* registers for use by exception and interrupt handlers.

Other states outside the processor generally need to be initialized as well, such as memory in particular. For example, reset code must reside in memory pointed to by the reset vector, the program must be present or loaded where expected (possibly copied from ROM to RAM), uninitialized ("bss") data must be cleared, and so on. Such initialization is mostly system and application specific.

Given that hardware does not fully reset these registers and states, reset code must initialize some of them. And given the dependencies between them, the order in which reset code initializes these registers and states is very important. This is reflected in the example reset code given later in this chapter.

Existence of the optional non-maskable interrupt (NMI) may affect the reset sequence. NMI is unaffected by `PS.INTLEVEL`. In other words, NMI can occur anytime during the reset handler: it will always be taken if asserted. Systems using NMI must be carefully designed such that either NMI is not asserted during the reset sequence, or NMI occurrences before completion of reset work correctly. The latter is not always practical: code such as the NMI handler might not yet have been copied from ROM to RAM, caches may be uninitialized, and other required initialization might not yet have completed, when an NMI is taken too early.

5. The `CACHEATTR` register is present in processors configured with XEA1. Those configured with XEA2 use TLB entries instead. See Appendix C.

4.2 The First Instruction: Making Space for Literals

We now look at an example reset vector. For the sake of readability, this example does not cover all possible Xtensa processor configurations. Instead, the example code that follows here and in subsequent chapters targets a specific processor configuration, the DF_212GP, as noted in Section 1.2 on page 3.

Let's start from the beginning of our hypothetical reset vector source file.

Note: A variant of the complete file can be found in Appendix A.7.

Xtensa processors load large constant values (called literals) from a negative offset from the PC. Furthermore, reset vectors are often placed at the start of memory, leaving no location for the literals needed by the reset vector. Therefore, the vector jumps over a location reserved for the literals needed by the rest of the reset vector. The assembler resizes this location as needed.

```
#include <xtensa/coreasm.h>

        .section .ResetVector.text, "ax"
        .global _ResetVector
        .align  4
_ResetVector:
        j       _ResetHandler // jump around literal pool

        .literal_position  // tells assembler to place literals here

        .align  4
_ResetHandler:
```

The code begins by including files that are helpful in writing the reset vector itself. A `.section` directive locates the subsequent code at the reset vector. A label for the reset vector itself called `_ResetVector` follows and is marked as global. Next, a `j` instruction transfers control to the `Reset` label. Between the `j` and the `Reset` label is the `.literal_position` directive that tells the assembler to place literal constants in this location,[6] as the reset vector does not have a corresponding literal section.

6. The assembler has two major modes for literals. In one mode it puts literals in a separate literal section. In the other it puts them in the text section. The `literal_position` directive works only when the assembler puts literals in the text section with the `-mtext-section-literals` option. Furthermore, the `literal_position` directive is relevant only for PC-relative L32R instructions (as opposed to absolute L32R — see the Extended L32R Option in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*).

4.3 Interrupts

At this point, all interrupts are unwanted. We shut them all off by clearing INTENABLE. The reset handler regularly uses a0 as an initializer register, and a0 often contains zero throughout the reset handler.

```
Reset:
        movi    a0, 0
        wsr     a0, INTENABLE
```

4.4 Getting a True Cycle Count

The processor reset does not initialize the CCOUNT (cycle count) register. However, there is some value in setting the CCOUNT early in the reset sequence so that the CCOUNT value is predictable.

```
        wsr     a0, CCOUNT
```

4.5 Debug State

The GNU Debugger (xt-gdb) can access processors that have the Debug Option and can examine the state of the processor in almost all execution states. (The *Xtensa Instruction Set Architecture (ISA) Reference Manual* discusses debug exceptions in detail.) Most debug exceptions can only be taken if PS.INTLEVEL is less than DEBUGLEVEL. A debug interrupt be taken when PS.INTLEVEL is greater than or equal to DEBUGLEVEL, but only in OCDRunMode, that is, when debugging over OCD. For this reason, the reset code typically lowers the interrupt level of the processor to a level where full debugging can occur, very early during boot time. Assuming that the processor has been configured so that the debug level is the highest interrupt level in the system (as is recommended), this does not run the risk of additional interrupts interfering with the system. Proper initialization of the INTENABLE special register is another method of masking unwanted interrupts before lowering the interrupt level (also highly recommended).

Additionally, other processor state is undefined on reset and must be initialized before we can lower interrupts below the debug level. All DBREAKA and DBREAKC registers are undefined at reset. But because the DBREAKA registers are unused unless enabled by a DBREAKC register, we only initialize the DBREAKC registers. This particular configuration has two:

```
        wsr     a0, DBREAKC_0
        wsr     a0, DBREAKC_1
        dsync
```

On processors up to T1050, the ICOUNTLEVEL register needs to be cleared to prevent ICOUNT interrupts. However, that step is not needed on more recent hardware.

4.6 Window Registers Initialization

Before calling code that uses the Xtensa Windowed ABI, such as C code, the registers that control the windowing mechanism must be initialized. Chapter 5 describes in detail how the values in these registers affect processor execution. But for now, we simply set the first logical window of sixteen registers to cover the first physical set of sixteen registers.

```
movi    a1, 1
wsr     a1, WINDOWSTART
wsr     a0, WINDOWBASE
rsync
movi    a0, 0
```

Observe that we reset a0. Because we just modified WINDOWBASE, we may have changed which physical registers we are using, and so cannot assume that any address register has a particular value.

4.7 Region Protection and Cache Initialization

Not all Xtensa processor versions support all memory protection options. See Chapter 9 for more details. The *MMU with TLB and Autorefill* option is a general-purpose MMU. If its static memory mappings (which allow for at least 128 MB of system RAM) and cache attributes are sufficient for an application, very little or no MMU initialization is required (see Section 4.1). Any other or specific uses of MMU facilities are beyond the scope of this discussion. See Section 9.2 on page 154 for relevant information on the MMU.

Xtensa processors come out of reset such that the reset vector code is in a region that is executable and not cached.

Regardless of the option, any configured caches must be initialized. We first illustrate the more complicated example of *Region Protection with Translation*, then address how to simplify the procedure for processors with the simpler *Region Protection* option.

In this example we use zero-overhead loops for cache initialization, but we must first clear the PS.EXCM bit to enable the looping mechanisms. The following code clears PS.EXCM while maintaining the current interrupt level and keeping window overflows disabled (PS.WOE=0).

```
movi    a2, XCHAL_DEBUGLEVEL - 1
wsr     a2, PS
rsync
```

The most complex operations of the reset sequence are initializing the address translations and memory access modes. Of the following steps, the first two are not required for a hard processor reset. However, they are nice to have for a soft reset, where software jumps to the reset vector.

- Turn off caching for all regions.
- Create a virtual to physical address mapping.
- Invalidate the instruction cache.
- Invalidate the data cache.
- Set memory-access modes.

4.7.1 Disabling the Caches and Creating Address Mappings

Most Xtensa programmers need not deal with all details of enabling or disabling caches or setting memory translations and TLB entries. Cadence provides easily used macros that handle the implementation details, such as `cacheattr_set`, `icacheattr_set`, and `dcacheattr_set`. These macros offer a convenient, consolidated view of memory access modes. See the file:

```
<xtensa_tools_root>/xtensa-elf/include/xtensa/cacheattrasm.h
```

where <*xtensa_tools_root*> is the location of your Xtensa Tools installation.

For pedagogical purposes, this section describes detailed steps of these manipulations. These steps are applicable for processors using XEA2. For processors configured with XEA1, see Appendix C.3.7 for information about controlling the caches.

At reset, Xtensa processors with the *Region Protection with Translation* option have all caches disabled and all virtual to physical address translations set to an identity map. That is, all memory access is set to bypass mode and all virtual addresses are mapped to the same physical addresses.

This reset handler example illustrates how to modify address translations and caching by creating the same memory environment as established at reset. The following loop disables the caches by setting the access modes to bypass and establishes an identity-map address translation for all regions.

```
            movi    a5, 0xE0000000
            movi    a3, 0
            movi    a4, 2
            j       3f[7]
2:          sub     a3, a3, a5
```

7. Note the use of the “1f” syntax. When used with a numerical label, the trailing “f” tells the assembler to jump to the next label “1” that follows the current statement. Likewise for “2b”, except the code jumps backward to the label “2”.

```
        sub     a4, a4, a5
3:      wdtlb   a4, a3
        witlb   a4, a3
        bne     a3, a5, 2b[7]
        isync
```

Now, let us dissect this loop, first addressing the loop initialization.

```
        movi    a5, 0xE0000000
        movi    a4, 2
        movi    a3, 0
        j       3f
```

We first load a5 with the base address of the highest memory region. This value will be used in the terminating condition of the loop. It also doubles as an address increment on each iteration. Figure 2 illustrates the format of the registers for the `wdtlb` and `witlb` instructions, assuming the *Region Protection with Translation* option. We initialize a4 with the first Physical Page Number (PPN) plus Access Mode (AM) and a3 with the first Virtual Page Number (VPN). Note that we start at the first memory region at virtual address zero. Because we are establishing an identity map, the PPN is also zero. The AM value of 2 indicates bypass, so we load a4 with 2 (PPN + AM).

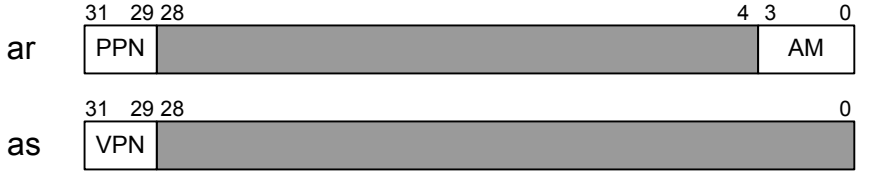

Figure 2. `WxTLB` Input Register Format for *Region Protection with Translation*

Finally, we jump into the loop at label 3.

```
2:      sub     a3, a3, a5
        sub     a4, a4, a5
3:      wdtlb   a4, a3
        witlb   a4, a3
        bne     a3, a5, 2b
        isync
```

After writing the ITLB and DTLB entries, we compare the current address against the ending address. If we have not reached the end, branch backward. At label 2, we simply increment both VPN and PPN parameters to the next memory region (subtracting `0xE0000000` is the same as adding `0x20000000`). Note that the Access Mode value is always preserved in the lower four bits of a4, so we initialize all memory regions to bypass mode.

After the loop we use an `isync` instruction to ensure all instruction and data path changes take effect (an `isync` also performs a `dsync` operation) before they may be needed.

Pitfalls

For simplicity and illustration, the previous code ignores some important, subtle points. One major assumption that makes the code work for this case is that it simply restores the processor's reset state. That is, it really is not modifying the virtual to physical address mappings.

Consider what would happen if code tried to modify an ITLB address mapping of the region in which it was currently executing. Basically, it will not work as expected. Of course, this scenario is not applicable to the DTLB mappings.

Consider what would happen if the code tried to execute load or store instructions while modifying the DTLB address mappings. (Remember that the assembler may relax a `movi` instruction into a load instruction.) The code will not work as expected.

Consider a case where you need to modify only the memory access mode without changing the address mappings. The Xtensa ISA does not provide instructions to modify only the Access Mode field of a TLB entry; the PPN field will also be written. Code needing to modify only the Access Mode field should carefully preserve the existing address mappings.

Solutions to these pitfalls exist in the macros described above and provided with all Xtensa processors. Cadence highly recommends that you use them.

Simplification for Region Protection

Xtensa processors with the *Region Protection* option (and XEA2) can change only the Access Mode field of TLB entries. The address mappings are fixed. Modifying the Access Mode fields can be done in exactly the same way as the *Region Protection with Translation* case, as the `witlb` and `wdtlb` instructions ignore the PPN field in the input register. The VPN field is still significant in identifying a specific memory region.

4.7.2 Initializing the Instruction Cache

We will now initialize the instruction cache by invalidating all of the cache entries. The first step is to unlock all cache-lines.

The following code sequence does this for an 8-KB two-way set associative cache and with 32 bytes per line:

```
        movi    a2, 8192
        movi    a3, 0
L0:
        iiu     a3, 0
        iiu     a3, 32
        iiu     a3, 64
        iiu     a3, 96
        addi    a3, a3, 128
        bne    a2, a3, L0
```

Let us first discuss how this sequence works, and then consider how the sequence can be changed to account for differences in the size and arrangement of the cache.

The instruction cache must be unlocked one index at a time, so we use an `iiu` instruction for each address. This instruction uses an index based on the virtual address to choose a line in the instruction cache and unlocks it. The instructions actually unlocking the lines are:

```
        iiu     a3, 0
        iiu     a3, 32
        iiu     a3, 64
        iiu     a3, 96
```

Each instruction unlocks a single cache line. The immediate offsets differ by 32 because the line size is 32. Consequently, a single execution of the inner loop unlocks four cache indices.

Surrounding this is the looping construct:

```
        movi    a2, 8192
        movi    a3, 0
.L0
        ...
        addi    a3, a3, 128
        bne     a2, a3, .L0
```

The a2 register gets the cache size. The a3 register holds the index. The ISA book details how the index selects the cache line. Put simply, the low bits of the index determine the line and the high-bits determine the way. Although it sounds complicated, the result is that we can loop through the entire cache simply by incrementing based on the cache line size, invalidating each index as we go.

This loop is unrolled four times, so we increment the index by 128, or four times the cache line size of 32. We then branch based on the total cache size. All indexed cache instructions work this way, and thus use the same loop control mechanism. In fact, both the unlocking loop and the invalidation loop (discussed next) could be combined into one long and harder to understand loop, but with higher performance.

We then invalidate each line in the instruction cache with a very similar loop, but one that uses `iii` instructions rather than `iiu` instructions:

```
        movi    a2, 8192
        movi    a3, 0
L1:
        iii     a3, 0
        iii     a3, 32
        iii     a3, 64
        iii     a3, 96
        addi    a3, a3, 128
        bne     a2, a3, L1
```

4.7.3 Initializing the Data Cache

The data cache is also an 8-KB two-way set-associative cache and supports cache line locking. Thus, we initialize the data cache exactly the same way, only substituting corresponding data-cache instructions for the instruction-cache instructions. The invalidate instructions do not affect locked lines, so we unlock them all first, as follows:

```
        movi    a2, 8192
        movi    a3, 0
L2:
        diu     a3, 0
        diu     a3, 32
        diu     a3, 64
        diu     a3, 96
        addi    a3, a3, 128
        bne    a2, a3, L2

        movi    a2, 8192
        movi    a3, 0
L3:
        dii     a3, 0
        dii     a3, 32
        dii     a3, 64
        dii     a3, 96
        addi    a3, a3, 128
        bne     a2, a3, L3

        dsync
```

The trailing `dsync` instruction ensures that the processor waits for the invalidates to complete before continuing.

4.7.4 Setting Memory-Access Modes

With the caches invalidated, we are ready to “turn on” the caches (that is, to set access modes to enable caching in the specific memory regions of interest). This section addresses XEA2; for information on how to handle XEA1, see Appendix C.

This operation is handled the same way as turning off the caches, with essentially the same code sequence. However, for pedagogical purposes we will add a twist and say that we now want to modify only the Access Mode field. This means we must first read the existing mapping so we can preserve it.

Let us start by defining the following assembler macro:

```
.macro  set_access_mode am
rdtlb1  a4, a3
ritlb1  a5, a3
srli    a4, a4, 4
slli    a4, a4, 4
srli    a5, a5, 4
slli    a5, a5, 4
addi    a4, a4, \am
addi    a5, a5, \am
wdtlb   a4, a3
witlb   a5, a3
.endm
```

This macro takes one parameter, the Access Mode value, to write into the TLB. It also has three implicit parameters. Register a3 points to the memory region, and a4 and a5 are working registers for the `DTLB` and `ITLB` instructions, respectively. (A more robust version of this macro would make all three registers formal parameters to the macro, but we hard-code them here for clarity.)

Now, let us dissect this macro.

```
rdtlb1  a4, a3
ritlb1  a5, a3
```

The `RDTLB1` and `RITLB1` instructions extract the PPN+AM combination from the corresponding TLB entry. The macro uses a4 and a5 to hold PPN+AM values from the DTLB and ITLB, respectively. See the *Xtensa Microprocessor Data Book* and the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for specifics on the `RITLB1` and `RDTLB1` instructions.

```
srli    a4, a4, 4
slli    a4, a4, 4
srli    a5, a5, 4
slli    a5, a5, 4
addi    a4, a4, \am
addi    a5, a5, \am
```

Shifting the PPN+AM values right by 4 bits then left by 4 bits effectively clears the AM portion. Then we add the desired Access Mode value to both, creating a new PPN+AM combination while preserving the original PPN mapping. Note that this macro applies the same Access Mode value to both TLBs.

```
wdtlb   a4, a3
witlb   a5, a3
```

Finally, the macro writes the new PPN+AM value to the DTLB and ITLB entries.

Now, suppose we want to disable all accesses to memory regions `0x00000000`, `0xC0000000`, and `0xE0000000`; enable caches for regions `0x40000000` and `0x80000000`; and leave all others unchanged (that is, uncached, in bypass mode). We could invoke the macro several times in the following code sequence to accomplish this design.

```
movi    a2, 0x20000000  /* region address increment */
movi    a3, 0           /* start at region 0 */
set_access_mode 15
add     a3, a3, a2      /* a3 <-- 0x20000000 */
add     a3, a3, a2      /* a3 <-- 0x40000000 */
set_access_mode 1
add     a3, a3, a2      /* a3 <-- 0x60000000 */
add     a3, a3, a2      /* a3 <-- 0x80000000 */
set_access_mode 1
add     a3, a3, a2      /* a3 <-- 0xA0000000 */
add     a3, a3, a2      /* a3 <-- 0xC0000000 */
set_access_mode 15
add     a3, a3, a2      /* a3 <-- 0xE0000000 */
set_access_mode 15
```

In this case we are using three different Access Mode values:

- A nibble of value ‘0xF’ specifies a memory range as invalid. Memory references to addresses that are in an invalid range will cause an exception.
- A nibble of value ‘1’ specifies a region whose accesses are cached.
- A nibble of value ‘2’ specifies a bypass range.

- The same pitfalls apply to this example as to turning off the caches previously. The same recommendations also apply. You will find some useful macros in our source-code distribution that provide a clean interface and results in smaller code. Again, see

  `<xtensa_tools_root>/xtensa-elf/include/xtensa/cacheattrasm.h`

Simplification for Region Protection

Xtensa processors with the *Region Protection* option (and XEA2) may change only the Access Mode field of TLB entries. Using the `tlb1` instructions will return only the Access Mode field, since the address mappings are fixed.

The variation applied to turning on the caches is meaningless and makes no difference for processors with the *Region Protection* option.

4.8 ROM Unpacking

The processor's performance will be much higher now that the caches have been turned on. We are now ready to unpack sections from ROM. This operation is best performed after caching is turned on, as ROM unpacking can be a time consuming operation. Regardless, the ROM unpacking must be performed before C code can be run. There are a couple of reasons for following this order. The first reason is Xtensa-specific: the Xtensa ABI requires window overflow and underflow handlers installed before code that uses it can run. The second reason is generic: C code often depends on initialized values in RAM and these values need to be unpacked out of the ROM and placed in RAM before code can be run.

We use an unpacking table as a convention to simplify the process of unpacking sections from ROM. Other architectures use explicit copying of data sections and initial vectors in other ways. However, unlike other architectures, Xtensa processors support local memories at arbitrary addresses that may contain initialized data or code. Vectors are also completely configurable. An unpack table supports all this functionality in one simple construct.

The linker scripts in the `nort-rom` linker support package prepare an executable for execution in systems whose RAM needs to be initialized. Sections that are located in RAM are archived in ROM, and the linker constructs a table of the information necessary to unpack these sections. As the final state of the reset vector, we want to make sure that we unpack all of the sections archived in ROM to their correct RAM locations.

Consider the code that does this unpacking:

```
          movi    a2, _rom_store_table
          beqz    a2, unpackdone

unpack:
          l32i    a3, a2, 0 // start vaddr
          l32i    a4, a2, 4 // end vaddr
          l32i    a5, a2, 8 // store vaddr
          bltu    a3, a4, unpack1
          bnez    a3, unpacknext
          bnez    a5, unpacknext
          j       unpackdone

unpack1:
          l32i    a6, a5, 0
          s32i    a6, a3, 0
          addi    a3, a3, 4
          addi    a5, a5, 4
          bltu    a3, a4, unpack1

unpacknext:
          addi    a2, a2, 12
          j       unpack

unpackdone:
```

This code has several implications. First, there is a global variable called `_rom_store_table`. The linker script creates this variable so that it points to a table that describes the sections to unpack. Second, each entry in this table consists of three entries with four bytes each. Were this table to be accessed in C, the structure for it would look like this:

```
struct rom_table {
    void *unpack_start;
    void *unpack_end;
    void *archive_start;
};
```

The first entry in the table, the one that is of zero offset from the base, holds the starting destination address of the data. The second entry, at an offset of 4 from the base, holds the ending destination address. The third entry, at an offset of 8 from the base, holds the starting source address.

This code works as follows:

```
          movi    a2, _rom_store_table
          beqz    a2, unpackdone
```

First, the address of the `_rom_store_table` is loaded. If this address is zero, nothing is archived in ROM (allowing a single reset vector to support many LSPs). The unpack is done and control is transferred to the end of the loop. Otherwise, the following section of code reads the first table entry:

```
unpack:
        l32i    a3, a2, 0 // start vaddr
        l32i    a4, a2, 4 // end vaddr
        l32i    a5, a2, 8 // store vaddr
        bltu    a3, a4, unpack1
        bnez    a3, unpacknext
        bnez    a5, unpacknext
        j       unpackdone
```

We load the destination start address into a3, the destination end address into a4, and the source address into a5. If the destination source is less than the destination end, we transfer control to `unpack1` to unpack the section. However, if either the destination start or the source start is not zero, then the next entry in the table is read by transferring control to `unpacknext`. If none of these conditions is true, the unpacking is done, as we have reached the terminating entry of all zeros.

The following code actually performs the unpacking:

```
unpack1:
        l32i    a6, a5, 0
        addi    a5, a5, 4
        s32i    a6, a3, 0
        addi    a3, a3, 4
        bltu    a3, a4, unpack1
```

This code is straightforward. The data is loaded from the store address in the archive, and then written to the destination address. Both the archive address and the destination address are then incremented. Finally, the unpacking continues as long as the destination address remains less than the destination ending address.

Now, we must prepare to read the next entry in the table. The a2 register holds the current address of the table pointer.

```
unpacknext:
        addi    a2, a2, 12
        j       unpack
```

This code adds 12 to the table pointer (the size of one table entry) and then jumps back to the code that performs the unpacking.

4.9 Calling the First C Function

Three final steps that must occur before the first C function can be invoked. The stack must be set up, the PS register must be set correctly, and the bss section must be cleared.

In an environment with multiple tasks there will be multiple stacks. Usually, quite a bit of operating system initialization is required before the OS is ready to begin starting tasks. There must be an initial stack for this operating system initialization to be done in C. The allocation of this initial stack is operating system-specific and may vary depending on the policy that the operating system chooses to use for its initialization stack.

The linker scripts declare a label __stack to be at the top of RAM. In the following code, we set the stack to begin at this label.

```
unpackdone:
        movi sp, __stack
```

Next, the PS register must be set to a correct initial value. This means that the interrupt level should be zero, PS.EXCM should be zero, PS.RING should be zero (the examples all run in privileged mode), PS.WOE should be set, and we choose to set PS.UM. This is done within the following statements:

```
        movi    a2, PS_WOE_MASK | PS_UM_MASK
        wsr     a2, PS
        rsync
```

After the rsync instruction, the changes to PS are effective. See Chapter 3 for details about the various PS fields.

The bss section can be cleared either before or after unpacking. We clear bss now. Generally, clearing bss is a simple operation that can be performed with the following:

```
        movi    a8, _bss_start
        movi    a10, _bss_end
        sub     a11, a10, a8
        srli    a11, a11, 2
```

This prolog to the clearing operation computes the number of words to clear, then places that result in a11. The linker script defines _bss_start and _bss_end and aligns them to a 4-byte boundary. This alignment permits the loop to use an s32i instruction. We now set up a zero-overhead loop to clear bss.

```
        movi    a9, 0
        loopnez a11, zerodone
        s32i    a9, a8, 0
        addi    a8, a8, 4
zerodone:
```

This can be written more efficiently. Even using the zero-overhead loop, there is one addition per written zero. Another, more efficient approach would be to clear one word at a time until the size of the `bss` section is a multiple of 16, then clear in larger blocks. The code would look like this:

```
        movi    a8, _bss_start
        movi    a10, _bss_end
        sub     a11, a10, a8
        slli    a12, a11, 28
        srli    a12, a12, 30

        movi    a9, 0
        loopnez a12, zerodone_fourbyte
        s32i    a9, a8, 0
        addi    a8, a8, 4
zerodone_fourbyte:

        srli    a11, a11, 4
        loopnez a11, zerodone_16byte
        s32i    a9, a8, 0
        s32i    a9, a8, 4
        s32i    a9, a8, 8
        s32i    a9, a8, 12
        addi    a8, a8, 16
zerodone_16byte:
```

This code is identical to the first code, except that it first clears out to `bss` so that the size is a multiple of 16. It then clears the rest of `bss` by writing in sets of four words.

With `bss` clear, window overflows enabled, and the processor set into user vector mode, we are now ready to call the first C function. a0 is set to zero to mark the first stack frame. We call a routine, `main`, that has the usual prototype.

```
callmain:
        movi    a0, 0
        movi    a6, 0   // clear argc
        movi    a7, 0   // clear argv
        movi    a8, 0   // clear envp
        movi    a4, main
        callx4  a4
```

We have just transferred control to `main` and are done with the reset sequence.

4.10 What Happens if the First C Function Returns?

The first C function may return. Even if the operating system programmer states that it is not valid to return from `main`, it is possible to make an error in the application and return anyway.

If this happens, the correct behavior often depends on the target that is running the code. In a deployed application, some sort of fatal-error function could be called that either reboots the system or displays the error that prompts a call to technical support. In a development system, executing a `break` instruction to drop into the debugger may be desirable. When running on the simulator, it makes sense to simply shut the simulator down and exit. The program is done.

The code to shutdown the simulator uses the simulator's semi-hosted environment. As the following code does this, it is what we will use through the rest of this discussion.

```
reset_exit:
        movi    a2, SYS_exit
        simcall
```

The structure of this exit code suggests the way in which the semi-hosted environment works. The simulator intercepts the `simcall` instruction and treats it as a special command. It examines the register values to determine what to do. An a2 value of `SYS_exit` instructs the simulator to exit.

4.11 Building the Reset Vector

There are some special considerations when building the reset vector. The reset vector, of all the Xtensa processor vectors, is the most likely to be set at the beginning of a memory device. For this reason, it is assumed that the reset vector does not have a corresponding literal section. It is assumed that the literal section for the reset vector will be in the text section for the reset vector. (Recall the discussion of the assembler directive `.literal_position` in Section 4.2.)

The assembler will, by default, place literals in a separate literal section. Consequently, the assembler must be explicitly instructed with the `-mtext-section-literals` directive to place literals in the text section for building the reset vector. The previous code can be built with the following command:

```
$ xt-xcc -g -c -mtext-section-literals reset.S
```

To ensure that the code that we wanted in the `.ResetVector.text` section is actually there, and to ensure that the object file did not end up with a literal section, we can examine the contents of the object file in the following way:

```
$ xt-objdump --section-headers reset.o

reset.o:     file format elf32-xtensa-be

Sections:
Idx Name                 Size      VMA       LMA       File off  Algn
  0 .literal             00000000  00000000  00000000  00000034  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text                00000000  00000000  00000000  00000034  2**0
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data                00000000  00000000  00000000  00000034  2**0
                   CONTENTS, ALLOC, LOAD, DATA
  3 .bss                 00000000  00000000  00000000  00000034  2**0
                   ALLOC
  4 .ResetVector.text    0000022b  00000000  00000000  00000034  2**2
                   CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .xt.prop             000000c0  00000000  00000000  0000025f  2**0
                   CONTENTS, RELOC, READONLY
```

Note that only two sections have a size greater than zero, `.ResetVector.text` and `.xt.prop`. Section `.xt.prop` contains only helper information for disassembly tools; it contains no object bytes generated from the source code. Indeed, you can verify this assertion with the `xt-size` tool, noting that the size of the object code is equal to the size of the `.ResetVector.text` section.

```
$ xt-size --format=xtensa reset.o

TYPE       text  rodata    data     bss     dec     hex  filename
code        517       0       0       0     517     205 reset.o
literal      36       0       0       0      36      24 reset.o
other         1       0       0       0       1       1 reset.o
Total       554       0       0       0     554     22a reset.o
```

5. Window Exception Handlers

The window exception handlers for the Xtensa processor are self-contained and need not be modified. The handlers were designed at the same time as the hardware that requires them and are optimal for the architecture and ABI. They are included and explained here because understanding the window exception handlers is a good way to understand the Xtensa Windowed ABI stack layout. Actual source code for the handlers is included in the <*xtensa_tools_root*>`/xtensa-elf/src/xtos/WindowVectors.S` file in the installed Xtensa Tools.

5.1 Architectural Overview of Windowed Registers

The Xtensa ISA offers a Windowed Register Option that improves both performance and code density by eliminating register saves and restores at procedure entry and exit, and argument-shuffling at the call. It allows more local variables to live permanently in registers, reducing the need for stack-frame maintenance in non-leaf routines.

Some registers are shared between the callee and the caller, and some registers are automatically preserved for the caller. As a result, the caller does not have to save and restore live register values to the stack by hand, as long as the saved registers fit into the number of registers saved by the call. This improves the compiled code's performance and code density.

The call instructions are named for the number of registers that they preserve. `Call4` preserves registers a0 through a3. `Call8` preserves registers a0 through a7. `Call12` preserves registers a0 through a11.

We need some terminology to discuss windowed registers. A *Window Frame* is the register group that is saved by a call instruction. A `call4` instruction creates a window frame of four registers. A `call8` creates a window frame of 8 registers. A `call12` creates a window frame of 12 registers. A window frame is always one of these three sizes. A `call0` instruction does not create a window frame; it is used in the context of a completely different ABI convention. See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details.

A *Window Pane* is a group of four registers, always on a modulo 4 boundary. That is, register groups a0..a3, a4..a7, a8..a11, a12..a15 are window panes. Register groups a0..a4, a1..a4, and a7..a10 are not window panes.

The processor has an array, or file, of physical registers and a *Logical Window* into that file. The processor maps a contiguous group of sixteen physical registers into sixteen logical registers. The logical registers are referred to as registers a0 through a15, and

they comprise a logical window. The physical registers are named ar0 through ar31, with additional registers ar32 through ar63 on configurations with 64 address registers. A windowed call rotates the mapping of logical registers to physical registers, but there is always some overlap between frames, so the caller and callee can use registers for passing arguments and returning values.

Consider the following example that calls `printf`. We want to perform the following C function:

```
main()
{
        printf( "%x\n", 1 );
}
```

For illustration, we want to select the type of call used, so we must write in assembly. This is our function:

```
        .data
        .align  4
.str:
        .string "%x\n"

        .text
        .align  4
        .global main
main:
        entry   sp, 48
        movi    a6, .str
        movi    a7, 1
        call4   printf
        retw
```

Running it gives the expected result.

```
$ xt-xcc func4.S
$ xt-run a.out
1
```

Note that we used a `call4` to invoke `printf`.

Now we examine what it would take to invoke `printf` with a `call8`. The `call4` rotated the logical window by four registers. This made `main`'s a6 and a7 registers visible to `printf` as a2 and a3. The `call4` completely hid `main`'s registers a0..a3 from `printf`. Using a `call8` will rotate the logical window by 8 registers instead of by 4. This will hide registers a0..a7 from `printf` and make registers a10 and a11 visible to `printf` as registers a2 and a3.

Performing this same invocation with a `call8` instruction requires the following changes to `main`:

```
main:
        entry   sp, 48
        movi    a10, .str
        movi    a11, 1
        call8   printf
        retw
```

Because we used a `call8`, the outgoing argument registers changed to a10 and a11. Observe that the called function always receives its arguments starting with a2.

Switching to `call12` is similar. A `call12` preserves registers a0..a11 and makes a14 and a15 visible to the callee as registers a2 and a3. Rewriting `main` to use a `call12` we see:

```
main:
        entry   sp, 48
        movi    a14, .str
        movi    a15, 1
        call12  printf
        retw
```

From the application programmer's perspective, this is the whole story and is really just a further discussion of the calling convention. However, the operating system programmer must confront additional issues surrounding windowed registers.

After some time, the physical register file will fill up, and rotating the logical window will wrap around and expose physical registers that are already in use by a function higher in the call chain. If this overflow condition occurs, the processor will cause a window overflow exception. The processor uses two special registers, `WINDOWSTART` and `WINDOWBASE`, to track logical windows and live window frames in the physical register file.

Recall that the reset code initially set `WINDOWBASE` to 0 and `WINDOWSTART` to 1. The `WINDOWBASE` register indicates the position of the current logical window in the physical register file. When we set `WINDOWBASE` to 0, we are mapping physical register 0 as logical register 0 and physical register 15 as logical register 15. That is, a0..a15 are mapped to ar0..ar15, and the other ar registers are not accessible directly.

`WINDOWSTART` contains one bit per window pane in the physical register file. It tracks the location of window frames in the physical register file by placing a 1 in the `WINDOWSTART` register at the base of each window frame (that is, the a0..a3 window pane of the window frame).

For further examination, we add the following `main` routine to the reset handler developed in Chapter 4.

```
void func(void)
{
}

int main(void)
{
    func();
}
```

We compile this system in the following way:

```
$ xt-xcc -g -c -mtext-section-literals reset.S
$ xt-xcc -g -c -O2 main.c
$ xt-xcc -mlsp=nort reset.o main.o
```

Start gdb with the resulting file using the command:

```
$ xt-gdb a.out
```

Recall that the label `callmain` is in the reset vector right before we call `main`. Set breakpoints at `callmain`, `main`, and `func`. The first breakpoint triggered is at callmain. In `xt-gdb`, we can examine processor registers.

Table 3 shows a snapshot of two register groups at `callmain`. The first group consists of two special registers, `WINDOWBASE` and `WINDOWSTART`. At `callmain`, they still have the same values that the reset code assigned to them.

The second register group consists of the physical register file, and Table 3 offers three different views. The simplest view is the entire physical register file. For our Xtensa processor configuration, the physical register file consists of 32 address registers. Another view is the logical window, the base of which is indicated by `WINDOWBASE`. The final view is the name of the window frame. The window-frame name associates the name of a function to the position of the logical window upon first entering the function.

If we continue the execution in `xt-gdb`, the next breakpoint will trigger at `main`. Note that the call to `main` is a `call4` and rotates the logical window by four registers. Table 4 shows a snapshot of the registers at this point.

Table 3. Register State at callmain

Window Register	Value		
WINDOWSTART	'8b00000001		
WINDOWBASE	0		
Window Frame Name	**Logical Register**	**Physical Register**	**Value**
callmain	a0	ar0	0
	a1	ar1	0x6ff7ffc0
	a2	ar2	0x40020
	a3	ar3	0xe0000000
	a4	ar4	0xe000000f
	a5	ar5	0xe000000f
	a6	ar6	0
	a7	ar7	0
	a8	ar8	0x600003c8
	a9	ar9	0
	a10	ar10	0x600003c8
	a11	ar11	0
	a12	ar12	0
	a13	ar13	0
	a14	ar14	0
	a15	ar15	0
		ar16	0
		ar17	0
		ar18	0
		ar19	0
		ar20	0
		ar21	0
		ar22	0
		ar23	0
		ar24	0
		ar25	0
		ar26	0
		ar27	0
		ar28	0
		ar29	0
		ar30	0
		ar31	0

Table 4. Register State at main

Window Register	Value		
WINDOWSTART	'b00000011		
WINDOWBASE	1		
Window Frame Name	**Logical Register**	**Physical Register**	**Value**
callmain		ar0	0
		ar1	0x6ff7ffc0
		ar2	0x40020
		ar3	0xe0000000
main	a0	ar4	0x4000023a
	a1	ar5	0x6ff7ffa0
	a2	ar6	0
	a3	ar7	0
	a4	ar8	0x600003c8
	a5	ar9	0
	a6	ar10	0x600003c8
	a7	ar11	0
	a8	ar12	0
	a9	ar13	0
	a10	ar14	0
	a11	ar15	0
	a12	ar16	0
	a13	ar17	0
	a14	ar18	0
	a15	ar19	0
		ar20	0
		ar21	0
		ar22	0
		ar23	0
		ar24	0
		ar25	0
		ar26	0
		ar27	0
		ar28	0
		ar29	0
		ar30	0
		ar31	0

Consider the changes to the logical window. Registers a0 through a15 now refer to physical registers ar4 through ar19. WINDOWBASE has been incremented from 0 to 1 and WINDOWSTART has another bit set, this one in the second bit position.

Each increment in WINDOWBASE moves the logical window forward by four registers, or one window pane. If WINDOWBASE is 1, a0 corresponds to ar4. If WINDOWBASE is 4, a0 corresponds to ar16. Each bit in WINDOWSTART refers to one window pane. Each bit set to 1 in WINDOWSTART indicates that a window frame starts at that point in the physical register file. In other words, a window frame's base window pane, a0..a3, starts at that position in the physical register file.

The WINDOWBASE was 0 before main was called, and only the callmain window frame was in the register file. This window frame (starting at WINDOWBASE of 0) was represented by '8b0000001 in the WINDOWSTART register. The name main represents a second window frame in the physical register file. The call4 incremented the WINDOWBASE to 1 and changed the WINDOWSTART to '8b0000011. This WINDOWSTART value indicates that one window frame starts at ar0 and another window frame starts at ar4. The WINDOWBASE of 1 indicates that the current logical window starts at ar4 and that logical register a0 is physical register ar4.

Note that the call sequence also changed the a0 and a1 registers. These are the return address and the new stack pointer for main. See the call4 and entry instruction descriptions in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details on these values.

Before continuing execution, we want to examine the assembly for main to see what type of call is used to invoke func. The disassemble command tells xt-gdb to disassemble a given function.

```
(xt-gdb) disassemble main
Dump of assembler code for function main:
0x600003b4 <main>:      entry   a1, 32
0x600003b7 <main+3>:    call8   0x600003ac <func>
0x600003ba <main+6>:    retw.n
End of assembler dump.
```

Note that the function uses a call8. Now let us continue execution to the breakpoint at func. Table 5 shows a snapshot of the registers when stopped at func.

Table 5. Register State at func

Window Register	Value		
WINDOWSTART	'b00001011		
WINDOWBASE	3		
Window Frame Name	**Logical Register**	**Physical Register**	**Value**
callmain		ar0	0
		ar1	0x6ff7ffc0
		ar2	0x40020
		ar3	0xe0000000
main		ar4	0x4000023a
		ar5	0x6ff7ffa0
		ar6	0
		ar7	0
		ar8	0x600003c8
		ar9	0
		ar10	0x600003c8
		ar11	0
func	a0	ar12	0xa00003ba
	a1	ar13	0x6ff7ff80
	a2	ar14	0
	a3	ar15	0
	a4	ar16	0
	a5	ar17	0
	a6	ar18	0
	a7	ar19	0
	a8	ar20	0
	a9	ar21	0
	a10	ar22	0
	a11	ar23	0
	a12	ar24	0
	a13	ar25	0
	a14	ar26	0
	a15	ar27	0
		ar28	0
		ar29	0
		ar30	0
		ar31	0

At this point, a new window frame starts at ar12 (WINDOWBASE of 3), which means that the fourth bit in the WINDOWSTART register is set. Registers a0 and a1 (ar12 and ar13) receive the return address and the new stack pointer.

With an understanding of the WINDOWBASE and WINDOWSTART registers and how the processor tracks the state of the window frames, we are now prepared to discuss the conditions that trigger a window overflow.

On every address register reference, the processor verifies that the referenced register does not belong to another window frame. It does this by checking whether any of the WINDOWSTART bits between the referenced register and WINDOWBASE are set. If a bit is set, the register belongs to another window frame, and the processor causes a window overflow exception.

To illustrate an overflow, change the example so func calls yet another function.

```
void func1(void)
{
}
void func(void)
{
    func1();
}
```

The compiler will generate a call8 instruction inside func to call func1.

After setting a breakpoint at func1 and restarting the program in xt-gdb, Table 6 shows the resulting register state at func1.

If func1 references any of the registers in the first physical window pane (physical ar0..ar3 or current logical a12..a15), it is accessing registers in the callmain window frame. At this point, bit position 0 in WINDOWSTART is set. The processor detects this condition and causes a window overflow exception. The overflow exception handler copies the callmain window frame from the physical register file into memory (on the stack). Table 7 on page 75 shows the state of the register file after returning from the overflow exception handler. Note that if func1 references only a0..a11, a window overflow is not triggered.

Table 6. Register State at func1

Window Register	Value		
WINDOWSTART	`8b00101011		
WINDOWBASE	5		
Window Frame Name	**Logical Register**	**Physical Register**	**Value**
callmain	a12	ar0	0
	a13	ar1	0x6ff7ffc0
	a14	ar2	0x40020
	a15	ar3	0xe0000000
main		ar4	0x4000023a
		ar5	0x6ff7ffa0
		ar6	0
		ar7	0
		ar8	0x600003c8
		ar9	0
		ar10	0x600003c8
		ar11	0
func		ar12	0xa00003ba
		ar13	0x6ff7ff80
		ar14	0
		ar15	0
		ar16	0
		ar17	0
		ar18	0
		ar19	0
func1	a0	ar20	0xa00003b2
	a1	ar21	0x6ff7ff60
	a2	ar22	0
	a3	ar23	0
	a4	ar24	0
	a5	ar25	0
	a6	ar26	0
	a7	ar27	0
	a8	ar28	0
	a9	ar29	0
	a10	ar30	0
	a11	ar31	0

Table 7. Register State Following Overflow

Window Register	Value		
WINDOWSTART	'8b00101010		
WINDOWBASE	5		
Window Frame Name	**Logical Register**	**Physical Register**	**Value**
	a12	ar0	0
	a13	ar1	0x6ff7ffc0
	a14	ar2	0x40020
	a15	ar3	0xe0000000
main		ar4	0x4000023a
		ar5	0x6ff7ffa0
		ar6	0
		ar7	0
		ar8	0x600003c8
		ar9	0
		ar10	0x600003c8
		ar11	0
func		ar12	0xa00003ba
		ar13	0x6ff7ff80
		ar14	0
		ar15	0
		ar16	0
		ar17	0
		ar18	0
		ar19	0
func1	a0	ar20	0xa00003b2
	a1	ar21	0x6ff7ff60
	a2	ar22	0
	a3	ar23	0
	a4	ar24	0
	a5	ar25	0
	a6	ar26	0
	a7	ar27	0
	a8	ar28	0
	a9	ar29	0
	a10	ar30	0
	a11	ar31	0

Note that the only register that has changed is the WINDOWSTART register. The callmain window frame has been saved to memory, so Table 7 no longer displays the window frame in the register file.

While the processor checks for window overflow on every register reference, the processor checks for window underflows only on retw and retw.n instructions. The underflow check is quite simple. Recall that the top bits of the return address correspond to the amount that the logical window was incremented on the call instruction. When returning, the processor decrements WINDOWBASE by this amount. If the bit in the WINDOWSTART register that corresponds to the new WINDOWBASE is not set, the window frame being referenced is not in the physical register file, but is instead in memory. The processor causes a window underflow exception to restore the register contents from memory.

5.2 Stack Frame Layout

An understanding of stack frames is important for understanding how to handle window exceptions, that is, where to save window frames. First, here are some more definitions.

The *Window Save Area* consists of portions of the stack frame into which the window overflow handlers save register contents. It is comprised of two parts: the Base Save Area and the Extra Save Area.

The *Base Save Area* is the part of the Window Save Area designated to hold the values of a window frame's a0 through a3. This area is a fixed size of 16 bytes and is located on the stack immediately below the stack-frame pointer (toward smaller addresses).

The *Extra Save Area* is the part of the Window Save Area designated to hold the values of registers a4 through a11. This area is of variable size, namely zero, 16, or 32 bytes large. This area is located on the stack within the highest addresses of a stack frame (immediately below the previous frame's base save area).

Figure 3 illustrates the layout of two adjacent stack frames. Stacks on Xtensa processors grow toward smaller addresses. The function's stack frame may contain local variables and the extra save area of the function. Function arguments passed in memory to a callee are part of the space designated for local variables. A function's stack frame always contains a base save area, but not from its function; instead, the base save area is for the a0..a3 registers of the caller's context. In other words, the base save area for a caller's a0..a3 is considered part of the callee's stack frame. This foreign base save area enables stack backtracing and simplifies other window exception handling functions. For a full discussion of these issues, refer to the ABI and Software Conventions for Xtensa chapter in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

Figure 3 also illustrates that a function's context is not completely contained within a function's stack frame. Indeed, a function's context may be pieced across a window frame in the physical register file, the base save area of a callee's stack frame, and the function's stack frame (which includes the extra save area).

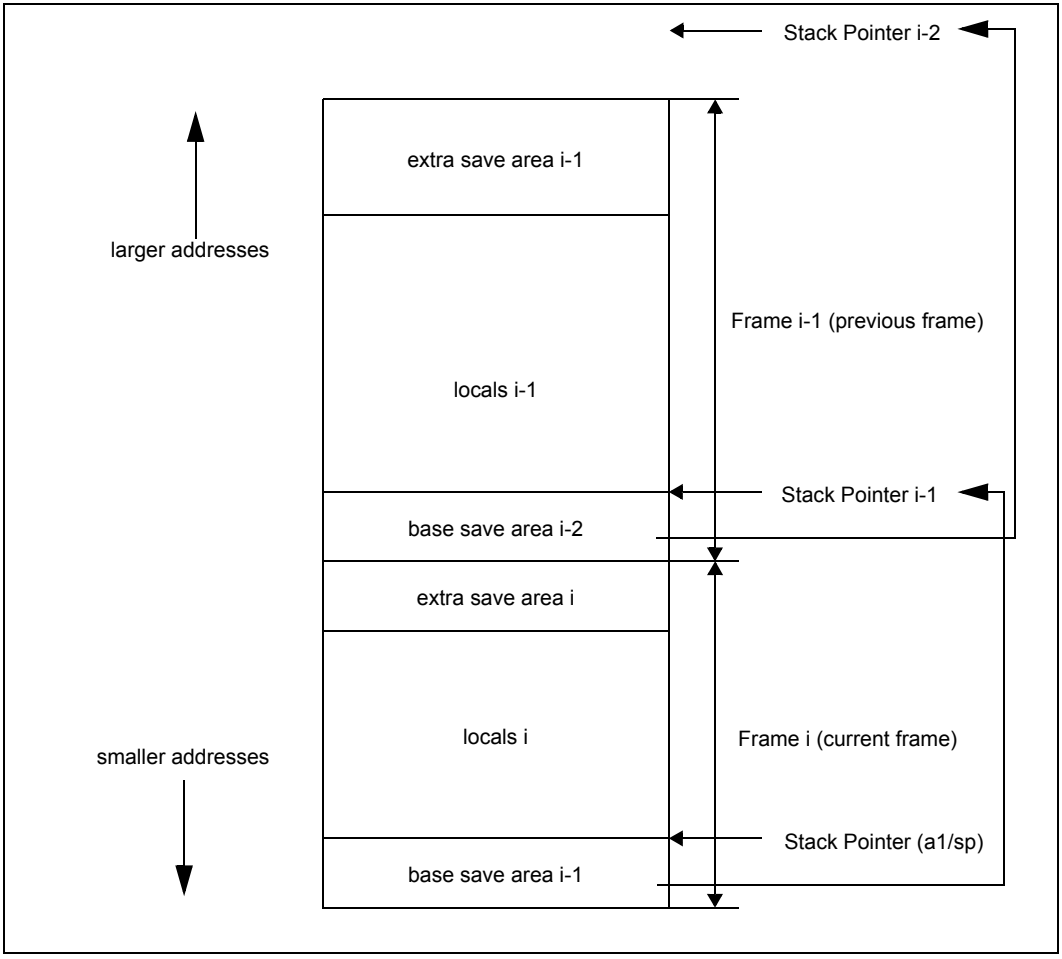

Figure 3. Xtensa Stack Frame Layout

5.3 Window Exception Handlers

There is one overflow and one underflow handler for each size of the window frame. There are three different window frame sizes (produced by `call4`, `call8`, and `call12` instructions) so there are a total of six different window handlers.

```
// Exports
.global _WindowOverflow4
.global _WindowUnderflow4
.global _WindowOverflow8
.global _WindowUnderflow8
.global _WindowOverflow12
.global _WindowUnderflow12
```

After declaring the global symbols for these six different handlers, we make sure that the vectors are placed in the proper section:

```
.section .WindowVectors.text, "ax"
```

Here is a case where we deviate somewhat from the guidelines that have been given previously. With six different window vectors, one would expect to have six different window vector sections. However, the processor knows only one window vector base and expects the vectors to be located at specific offsets from that base. Table 8 lists the offset values. There are 64 bytes reserved for each vector.

Table 8. Vector Offsets

Vector	Offset From Base
WindowOverflow4	0
WindowUnderflow4	64
WindowOverflow8	128
WindowUnderflow8	192
WindowOverflow12	256
WindowUnderflow12	320

Now consider the overflow handler for window frames that hold four registers. We will use a single section and use the `.align` directive to locate code, rather than using six different sections.

When the processor detects the overflow of a four-register window frame, the processor aborts the instruction that caused the overflow (it will re-issue the instruction after the overflow is completed). The processor then transfers control to the appropriate overflow handler. It sets `WINDOWBASE` to map the window frame being overflowed so that the first register in the window frame being overflowed is in logical register a0.

Understanding this invocation convention, consider the overflow handler for a four-register window frame:

```
    .align 64
_WindowOverflow4:
    s32e    a0, a5, -16 // save a0 to call[j+1]'s stack frame
    s32e    a1, a5, -12 // save a1 to call[j+1]'s stack frame
    s32e    a2, a5,  -8 // save a2 to call[j+1]'s stack frame
    s32e    a3, a5,  -4 // save a3 to call[j+1]'s stack frame
    rfwo                // rotates back to call[i] position
```

We are saving a caller's four-register window frame to memory. a0 is the base of this window frame, and a5 is the stack-pointer register (a1) of the callee's window frame. If we are saving the caller's window frame, the callee's window frame still resides in the physical register file. That is how we can use a5.

Remember the discussion of required arguments to the `entry` instruction: functions must include at least an additional 16 bytes in the stack frame for the caller's base save area. The four registers for this window frame are stored in the base save area, which is immediately below the callee's stack pointer (see Figure 3 on page 77 for a diagram). Then the handler returns from the overflow exception.

The corresponding underflow routine follows the same idea.

```
    .align 64
_WindowUnderflow4:
    l32e    a0, a5, -16 // restore a0 from call[i+1]'s stack frame
    l32e    a1, a5, -12 // restore a1 from call[i+1]'s stack frame
    l32e    a2, a5,  -8 // restore a2 from call[i+1]'s stack frame
    l32e    a3, a5,  -4 // restore a3 from call[i+1]'s stack frame
    rfwu
```

First, recall that the processor is going to give control to the window vector base, plus 64, when it wants to perform an underflow of a four-register window frame. We move this code to a location 64 bytes after the vector by aligning the code to a 64-byte boundary. We know that the code for the `_WindowOverflow4` is less than 64 bytes. Obviously, this technique would not work correctly if it was more than 64 bytes.

When invoking the underflow handler, the processor sets the `WINDOWBASE` so that a0 is the a0 of the caller's window frame to be restored. We are returning from the callee and need to restore the caller's window frame.

Recall that underflows occur only on a windowed return instruction. A `retw` instruction knows the size of the window frame by the top two window-increment bits of the return address in a0.

At `_WindowUnderflow4`, we have a four-register window frame, and the stack pointer of the callee's window frame is in a5. That is, it is in the a1 register of the callee's window frame that is four registers away. We also know that the caller's a0..a3 are stored in the base save area immediately below the callee's stack-frame pointer.

This handler simply loads the registers with their correct values from memory, and returns.

The basics of handling the other window overflows and underflows are similar. The only difference is that the window frames are larger, and we must store the extra data somewhere. Consider the first part of the overflow handler for eight-register window frames.

```
    .align 64
_WindowOverflow8:
    s32e    a0, a9, -16 // save a0 to call[j+1]'s stack frame
    l32e    a0, a1, -12 // a0 <- call[j-1]'s sp
    s32e    a1, a9, -12 // save a1 to call[j+1]'s stack frame
    s32e    a2, a9,  -8 // save a2 to call[j+1]'s stack frame
    s32e    a3, a9,  -4 // save a3 to call[j+1]'s stack frame
```

Except for the `l32e` instruction, this code looks quite similar to the overflow handler for four-register window frames. The only difference is that the callee's stack pointer is in a9 rather than in a5. We start by saving the caller's a0..a3 in the base save area in the callee's stack frame. The window frame is eight registers in size, so the callee's a1 is in the caller's a9.

Next we must save the caller's a4..a7 in the extra save area. The extra save area is found at the highest addresses in the caller's stack frame. We can find the highest addresses in the caller's stack frame by skipping the base save area just below the caller of the caller's stack-frame pointer.

```
    s32e    a4, a0, -32 // save a4 to call[j]'s stack frame
    s32e    a5, a0, -28 // save a5 to call[j]'s stack frame
    s32e    a6, a0, -24 // save a6 to call[j]'s stack frame
    s32e    a7, a0, -20 // save a7 to call[j]'s stack frame
    rfwo                // rotates back to call[i] position
```

The registers for the caller's window frame are stored just below the stack pointer for the callee's stack frame. To perform an overflow or an underflow, there must be an easy way to find the adjacent window frame's stack pointer. This method is that easy way. If a window frame is in memory, the first four registers (including the stack pointer) of the caller's window frame are in the base save area. In other words, the stack pointer of the caller's window frame is stored immediately beneath the callee's stack pointer in memory (see Figure 3 on page 77).

The `l32e` instruction (that we previously ignored) loaded a0 with the caller of the caller's stack-frame pointer. We know that this stack pointer is in memory because the caller window frame is in the process of being saved to memory. That means that all ancestor window frames have already been saved to memory.

We store a4..a7 in the extra save area portion of the caller's stack frame. This stack frame is accessible via the caller of the caller's stack pointer in a0. Using negative offsets (remembering to skip the 16 bytes of a base save area), we can save a4..a7 in the appropriate place. Figure 3 on page 77 may help the reader visualize this step.

Now consider the eight-register window frame underflow handler. The start of this handler looks quite a bit like the start of the underflow handler for four-register window frames:

```
    .align 64
_WindowUnderflow8:
    l32e    a0, a9, -16 //restore a0 from call[i+1]'s stack frame
    l32e    a1, a9, -12 //restore a1 from call[i+1]'s stack frame
    l32e    a2, a9,  -8 //restore a2 from call[i+1]'s stack frame
    l32e    a7, a1, -12 // a7 <- call[i-1]'s sp
    l32e    a3, a9,  -4 // restore a3 from call[i+1]'s stack frame
```

The only differences come from the size of the window frame. We are returning from the callee and need to restore the caller's window frame. The callee's stack pointer is in a9. The caller's a0..a3 are in the base save area, just below the callee's stack pointer. We use negative offsets from the callee's stack pointer, and restore the caller's a0..a3.

We also use a7 as temporary variable while restoring a4..a7. The extra save area containing the caller's a4..a7 is located in the caller's stack frame. We access this extra save area via the caller of the caller's stack pointer, which we load into a7.

```
    l32e    a4, a7, -32 // restore a4 from call[i]'s stack frame
    l32e    a5, a7, -28 // restore a5 from call[i]'s stack frame
    l32e    a6, a7, -24 // restore a6 from call[i]'s stack frame
    l32e    a7, a7, -20 // restore a7 from call[i]'s stack frame
    rfwu
```

We use negative offsets to skip over the base save area and restore a4..a7 from the extra save area. Note that the caller of the caller's stack pointer must be in memory because we are now restoring the caller's window frame from memory. This code section simply restores a4..a7 and returns from the underflow.

The underflow and overflow handlers for 12-register window frames are identical to the handlers for eight-register window frames, except that 12-register window handlers must manage a window frame that has 12 registers.

```
    .align 64
_WindowOverflow12:
    s32e    a0,  a13, -16 // save a0 to call[j+1]'s stack frame
    l32e    a0,  a1,  -12 // a0 <- call[j-1]'s sp
    s32e    a1,  a13, -12 // save a1 to call[j+1]'s stack frame
    s32e    a2,  a13,  -8 // save a2 to call[j+1]'s stack frame
    s32e    a3,  a13,  -4 // save a3 to call[j+1]'s stack frame
    s32e    a4,  a0,  -48 // save a4 to end of call[j]'s stack frame
    s32e    a5,  a0,  -44 // save a5 to end of call[j]'s stack frame
    s32e    a6,  a0,  -40 // save a6 to end of call[j]'s stack frame
    s32e    a7,  a0,  -36 // save a7 to end of call[j]'s stack frame
    s32e    a8,  a0,  -32 // save a8 to end of call[j]'s stack frame
    s32e    a9,  a0,  -28 // save a9 to end of call[j]'s stack frame
    s32e    a10, a0,  -24 //save a10 to end of call[j]'s stack frame
    s32e    a11, a0,  -20 //save a11 to end of call[j]'s stack frame
    rfwo                  // rotates back to call[i] position
```

This handler is very similar to the case of eight-register window frames with two main differences. First, the callee's stack pointer is in a13 because we are saving a 12-register window frame. Second, the caller's extra save area is 32 bytes large (instead of 16 bytes) because we are saving a4..a11. We first save a0..a3 in the base save area, followed by a4..a11 in the extra save area.

The underflow is also similar.

```
    .align 64
_WindowUnderflow12:
    l32e    a0,  a13, -16 // restore a0 from call[i+1]'s stack frame
    l32e    a1,  a13, -12 // restore a1 from call[i+1]'s stack frame
    l32e    a2,  a13,  -8 // restore a2 from call[i+1]'s stack frame
    l32e    a11, a1,  -12 // a11 <- call[i-1]'s sp
    l32e    a3,  a13,  -4 // restore a3 from call[i+1]'s stack frame
    l32e    a4,  a11, -48 // restore a4 through a11 from the end
    l32e    a5,  a11, -44 //   of call[i]'s stack frame
    l32e    a6,  a11, -40
    l32e    a7,  a11, -36
    l32e    a8,  a11, -32
    l32e    a9,  a11, -28
    l32e    a10, a11, -24
    l32e    a11, a11, -20
    rfwu
```

Again, there are two main differences between this case and the case of eight-register window frames. First, the callee's stack pointer is in a13 due to the 12-register frame. Second, the extra save area is 32 bytes large to hold a4..a11. We first restore a0..a3 from the base save area, and then we restore a4..a11 from the extra save area.

Note: For XEA1, certain details of the window handling mechanism are slightly different. For more information, see Appendix C.3.5.

6. Timer Interrupts and Interrupt Handling

Most real-time operating systems use a regularly occurring timer interrupt as a system heartbeat or tick. The operating system takes control and does certain housekeeping tasks on every tick. Getting the tick working requires that timers be set, interrupts be enabled, and interrupts be processed. This chapter addresses these topics.

6.1 Setting the Timers

The Xtensa architecture provides for up to four timers (though current implementations are limited to three). Processor configurations that will run classic real-time operating systems require at least one timer.

Each timer has a CCOMPARE register. These timer registers are named CCOMPARE_0 through CCOMPARE_*N-1*, where N is the number of timers. The CCOMPARE registers are continuously compared against the CCOUNT register, which increments every cycle. An interrupt is posted when the CCOMPARE and the CCOUNT registers are equal.

Setting the timer to go off at a particular time requires setting the CCOMPARE register to the cycle count at which the interrupt should occur. Because this is a special register, it is not directly accessible from C. Assembly must be used to access the timer registers.

6.1.1 C-Callable Assembly to Read CCOUNT

Updating the CCOMPARE register often requires access to the CCOUNT register. Essentially, we want a function that matches the following prototype:

```
unsigned int read_ccount();
```

The assembly version of this function is:

```
#include <xtensa/config/specreg.h>
        .text
        .align  4
        .global read_ccount
        .type   read_ccount,@function
read_ccount:
        entry   a1, 16
        rsr     a2, CCOUNT
        retw
```

This simple function uses rsr to read CCOUNT into a2, and then returns.

Note that the Xtensa HAL includes many useful utilities, such as the one above. The equivalent HAL function is called `xthal_get_ccount`, with the same prototype as `read_ccount`. While `xthal_get_count` is a suitable drop-in replacement for `read_ccount`, we continue with our contrived examples for their pedagogical benefits. See the *Xtensa System Software Reference Manual* for other useful library functions in the Xtensa HAL.

6.1.2 Reading CCOUNT with Inline Assembly

Consider the following code that uses the previous function to assign the current `CCOUNT` to global variable `a`:

```
int a;

main()
{
    a = read_ccount();
}
```

The compiler produces the following assembly for this function:

```
a.o:     file format elf32-xtensa-le

Disassembly of section .literal:

00000000 <.literal>:
   0:   00000000                                        ....
                        0: R_XTENSA_32  .bss
Disassembly of section .text:

00000000 <main>:
   0:   004136          entry   a1, 32
   3:   000025          call8   4 <main+0x4>
                        3: R_XTENSA_SLOT0_OP    read_ccount
   6:   000081          l32r    a8, fffc0008 <main+0xfffc0008>
                        6: R_XTENSA_SLOT0_OP    .literal
   9:   08a9            s32i.n  a10, a8, 0
   b:   f01d            retw.n
```

Note the use of the `call8` instruction to invoke `read_ccount`. This is an expensive way to read `CCOUNT`, as the value of the `CCOUNT` register could be read into a2 where the `call8` is presently generated by the compiler.

Another way to write assembly for use in C code is to use the inline assembly features of the `xt-xcc` compiler along with the inline function features.

Let us review the results before discussing the mechanisms. Consider the following change to our `main` function, where we have inlined the `read_ccount()` function:

```
#include <xtensa/config/specreg.h>
#include "timer.h"
int a;

main()
{
    a = read_ccount();
}
```

Compilation of this `main` function gives the following result:

```
c.o:     file format elf32-xtensa-le

Disassembly of section .literal:

00000000 <.literal>:
   0:   00000000                                        ....
                        0: R_XTENSA_32  .bss
Disassembly of section .text:

00000000 <main>:
   0:   004136          entry   a1, 32
   3:   000081          l32r    a8, fffc0004 <main+0xfffc0004>
                        3: R_XTENSA_SLOT0_OP    .literal
   6:   03ea30          rsr.ccount      a3
   9:   0839            s32i.n  a3, a8, 0
   b:   f01d            retw.n
```

This result is a much more efficient routine. Here is the inline version from <*timer.h*> that gives the improved result:

```
static __inline__ int read_ccount()
{
    unsigned int ccount;

    __asm__ __volatile__ (
        "rsr     %0, ccount"
        : "=a" (ccount) : : "memory"
        );
    return ccount;
}
```

At the heart of the inline function stands the use of an `__asm__` statement. The full details of the `__asm__` statement is documented in the *Xtensa C and C++ Compiler User's Guide* and is beyond the scope of this guide. However, the use of this directive is quite

simple here. The __asm__ sets the local variable ccount. The local variable ccount is described to the __asm__ as its one output. The assembly statement that we wish to use to compute ccount is included in the __asm__ statement itself. Note that the register used for ccount is referred to as %0. One of the benefits of inline assembly is that the compiler still allocates registers and the compiler will substitute the appropriate register for %0 in the __asm__ directive.

As a side note, a common reason to read CCOUNT is to measure the execution time of a code block. However, when using __asm__ statements, the C compiler may shuffle instructions into or out of the logical code block that the programmer wishes to measure. Adding memory to the clobber list of the __asm__ statement acts as a barrier over which the compiler will not move memory instructions (other ALU instructions may still move). In any case, verifying the disassembly (that is, where the reads to CCOUNT are placed) is the best way to verify tight timing requirements around narrow code blocks.

6.1.3 Reading CCOMPARE

The code to read CCOMPARE is almost identical to the code that reads CCOUNT. The following two inline functions read CCOMPARE0 and CCOMPARE1:

```
static __inline__ int read_ccompare0()
{
    unsigned int ccount;

    __asm__ __volatile__ (
        "rsr     %0, ccompare0"
        : "=a" (ccount)
        );
    return ccount;
}

static __inline__ int read_ccompare1()
{
    unsigned int ccount;

    __asm__ __volatile__ (
        "rsr     %0, ccompare1"
        : "=a" (ccount)
        );
    return ccount;
}
```

The functions read_ccompare0 and read_ccompare1 work identically to read_ccount. To ensure that they are working correctly, we use the following main test routine:

```
#include <xtensa/config/specreg.h>
#include "timer.h"
int a[2];

main()
{
    a[0] = read_ccompare0();
    a[1] = read_ccompare1();
}
```

The disassembly shows that these functions achieve the desired result:

```
t.o:     file format elf32-xtensa-le

Disassembly of section .literal:

00000000 <.literal>:
   0:   00000000                                        ....
                        0: R_XTENSA_32  .bss
Disassembly of section .text:

00000000 <main>:
   0:   004136          entry   a1, 32
   3:   000081          l32r    a8, fffc0004 <main+0xfffc0004>
                        3: R_XTENSA_SLOT0_OP    .literal
   6:   03f090          rsr.ccompare0   a9
   9:   0899            s32i.n  a9, a8, 0
   b:   03f130          rsr.ccompare1   a3
   e:   1839            s32i.n  a3, a8, 4
  10:   f01d            retw.n
```

The base address of the `a` array is loaded into a8 with the `l32r`. The `CCOMPARE` values are loaded into `a9` and `a3` with `rsr` instructions. The `CCOMPARE` values are then stored into the appropriate array offsets with `s32i` instructions.

6.1.4 Setting CCOMPARE

Setting the `CCOMPARE` values requires more expansive use of `__asm__`. Previous routines have only read values and have not taken values as input. Setting `CCOUNT` requires a C variable as an input parameter to the `__asm__`. The inline function to set `CCOMPARE_0` is as follows:

```
static __inline__ void set_ccompare0(int val)
{
    __asm__ __volatile__ (
        "wsr     %0, ccompare0\n"
        "isync\n"
```

```
        :
        : "a" (val)
        );
}
```

Inputs to __asm__ statements are optionally declared immediately after outputs. In this case, the register that the compiler has allocated for `val` is written to the `ccompare0` special register with the `wsr` instruction.

6.1.5 Setting the First Timer Value

We have produced the basic building blocks that will set up the first timer. The code to set the timer is now amazingly simple:

```
#include <xtensa/config/specreg.h>
#include "timer.h"

#define TIMER_INTERVAL 0x1000
main()
{
        set_ccompare0( read_ccount() + TIMER_INTERVAL );
}
```

This statement sets the `CCOMPARE` register to be the currently read `CCOUNT`, plus the `TIMER_INTERVAL`. The code produced by the compiler is as follows:

```
d.o:     file format elf32-xtensa-le

Disassembly of section .text:

00000000 <main>:
   0:   004136          entry   a1, 32
   3:   03ea80          rsr.ccount      a8
   6:   10d882          addmi   a8, a8, 0x1000
   9:   13f080          wsr.ccompare0   a8
   c:   002000          isync
   f:   f01d            retw.n
```

The generated code is simple and efficient. `CCOUNT` is read into a8, 0x1000 is added to a8, and the result is written into `ccompare0`.

6.2 Enabling Interrupts

Setting the CCOMPARE register only sets the time when an interrupt becomes pending, not when it is taken. The INTENABLE register determines whether or not the pending interrupt will affect the processor execution. The bit in the INTENABLE register that represents timer 0 must be set to 1 for the interrupt to affect the processor, so the INTENABLE register must be set before a timer interrupt can occur.

For the example processor, timer 0 is tied to interrupt 6, which means that we want to set bit 6 in INTENABLE. In the end, we want our main routine to look like this:

```
#include <xtensa/config/core.h>
#include <xtensa/config/specreg.h>
#include "timer.h"

#define TIMER_INTERVAL 0x1000
#define TIMER_INT_MASK (1 << 6)
main()
{
        set_ccompare0( read_ccount() + TIMER_INTERVAL );
        enable_ints( TIMER_INT_MASK );
}
```

Main is straightforward. Enable_ints is more difficult.

If the INTENABLE register may be modified by interrupt handlers, we must make sure that changes to INTENABLE are guarded against interrupts. The following inline function sets bits in the INTENABLE register atomically with respect to level-one interrupts.

```
static __inline__ unsigned int enable_ints(unsigned int mask)
{
    unsigned int ret;
    unsigned int new_intenable;

    __asm__ __volatile__(
        "rsil    a15, 1                \n\t"
        "rsr     %0, intenable         \n\t"
        "or      %1, %0, %2            \n\t"
        "wsr     %1, intenable         \n\t"
        "wsr     a15, PS               \n\t"
        "rsync                         \n\t"
        : "=&a" (ret), "=&a" (new_intenable)
        : "a" (mask)
        : "a15"
        );
    return ret;
}
```

This routine is using new syntax for the inline assembly definition. The routine produces two outputs, the old INTENABLE value ret and the new INTENABLE value new_intenable. There is one input variable mask. We intentionally use a15 to provoke a window overflow in case the compiler selects a register that would otherwise cause a window overflow while interrupts are disabled.[8] Refer to the *Xtensa C and C++ Compiler User's Guide* for the details of __asm__ syntax.

This routine first takes the processor to interrupt level one. It then reads the INTENABLE special register and sets the bits that are in mask. It writes the new value back into INTENABLE and restores the previous processor interrupt level.[9]

Note: In some operating systems, such as the XTOS basic runtime, care must be taken to never modify the INTENABLE register directly, and instead use the operating system's facilities for enabling and disabling interrupts. The INTENABLE register sometimes plays two roles: that of enabling and disabling individual interrupts, and that of masking interrupts by level. This generally occurs in two cases. First, with processors configured with XEA1, where windowed ABI code (e.g. C code) can only run at interrupt priority level zero due to the windowed exception mechanism and must use another mechanism than PS.INTLEVEL to mask interrupts while running C code. Second, where the OS implements software prioritization of interrupts within an interrupt priority level, i.e. with virtual priority levels that are finer grained than what is represented in PS.INTLEVEL. In these cases, the operating system atomically updates INTENABLE whenever individual interrupts are enabled or disabled, and whenever the virtual interrupt level mask changes.

6.3 Handling the First Interrupt

We add an infinite loop at the end of the main routine. This loop prevents the program from exiting, allowing us to examine timer interrupts in the simulator. Without this loop, the program would finish and exit before the first timer interrupt ever occurred.

```
#include <xtensa/config/core.h>
#include <xtensa/config/specreg.h>
#include "timer.h"

#define TIMER_INTERVAL 0x1000
#define TIMER_INT_MASK (1 << 6)
main()
{
```

8. The selection of a15 is critically important for Xtensa processors configured with XEA1. On these processors, the processor lowers PS.INTLEVEL to zero on a rfwo instruction. The effect would be to re-enable interrupts and destroy the intended atomicity of the function. While the selection of a15 is not important for processors configured with XEA2, the function is portable across XEA1 and XEA2.

9. This routine expressly sets the processor interrupt level to one, so it is not safely callable at interrupt levels higher than one. Also, it is not safe for handlers of interrupts at levels higher than one to manipulate INTENABLE, except temporarily. The runtime designer must decide at which interrupt level the INTENABLE register will be atomically manipulated. In this example, that interrupt level is one. This has the least interrupt latency impact on higher level interrupt handlers. This routine can be changed to support higher level atomiticity by changing the initial rsil instruction.

```
        set_ccompare0( read_ccount() + TIMER_INTERVAL );
        enable_ints( TIMER_INT_MASK );
        while(1);
    }
```

When the interrupt occurs, the processor will begin executing at the `UserExceptionVector`.

6.3.1 The Interrupt Handling Plan

Before examining the details of the interrupt handling, let us briefly consider the goals and mechanisms of interrupt handling. A primary goal of interrupt handling is speed. The time between the recognition of the interrupt and the invocation of the user's interrupt handler should be as low as possible. Another goal is memory efficiency. We do not want to require large amounts of extra memory in the system to deal with interrupts. In this first example, we will assume that only a single task is being serviced. At the same time, we will consider the implications of supporting multiple tasks.

Interrupt handlers require a stack. If it is the user's stack, then in a multiple-task system every user stack must have sufficient space for the worst case interrupt-stack usage. This extra stack size can become costly. For this reason, we will use a separate interrupt stack for all interrupts, regardless of the user task that was interrupted.

There are two ways to address speed: algorithms and implementation. The following examples will deal only with algorithmic speed. These examples are functionally correct, but perhaps lack an optimal set of instructions. The approach provides clarity to the example.

At the time an interrupt occurs, the physical register file contains live window frames of the executing code. The code could have been either user or kernel code, but we assume user code was interrupted, for now.

There are several ways to transfer to the interrupt stack. One way is to save all the window frames to memory and then begin the interrupt stack again. While this is a simple solution to the problem, saving window frames to the stack can be costly and time consuming.

Instead, we will insert a *linking stack frame* so the window overflow exception handler will still function properly. We achieve much lower interrupt latencies by leaving the live window frames in the physical register file (that is, we do not need to save them to memory).

6.3.2 The UserExceptionVector

The processor transfers control to the UserExceptionVector when all of the following conditions are true:

- The processor is in the user vector mode, that is, PS.UM is set.
- A general exception occurs (including a level-one interrupt).
- On XEA2, PS.EXCM is clear.

To invoke an interrupt handler, we must first place code at the vector location that jumps to a handler. Generally speaking, code for the UserExceptionVector will be larger than the space allocated for the vector, so we use a simple stub to jump from the user exception vector to its handler located elsewhere.

```
/*
   UserExceptionVector

   This implements the user exception vector and transfers
   control to the UserExceptionHandler.
*/
#include <xtensa/config/specreg.h>

        .section .UserExceptionVector.text, "ax"
        .begin literal_prefix .UserExceptionVector
        .align 4

_UserExceptionVector:
        wsr     a3, EXCSAVE_1
        movi    a3, _UserExceptionHandler
        jx      a3

        .end    literal_prefix
```

The first thing that this code does is prepare a3 for use by saving it into the EXCSAVE_1 special register.[10] Control is then transferred to the _UserExceptionHandler. As the j instruction has a limited offset, control is transferred with the jx instruction. The distance from the UserExceptionVector to the UserExceptionHandler is not known, so use of the jx instruction guarantees that the destination address can be reached.

10. EXCSAVE_1 is a special register included in the Xtensa processor for use in the UserException and KernelException vectors. It solves the problem of how to get a working register by allowing the user to save an address register with a wsr instruction. In this example, the address register a3, which contains information that must be preserved, is saved into EXCSAVE_1. EXCSAVE_1 is conventionally safe to use only when the processor is at interrupt level one (or, more precisely, at interrupt level EXCM_LEVEL) while handling general exceptions.

6.3.3 The UserExceptionHandler Prolog

In this example, we will assume that the `UserExceptionHandler` has a hardcoded dispatch for every type of exception. We will not be producing a generic exception dispatching mechanism.

```
/* LevelOneHandlers.S

  Handle the user and kernel exceptions.
*/
#include <xtensa/coreasm.h>

        .data
        .align  4
        .global userStackPtr
userStackPtr:
        .word 0

        .align  16
        .global intStackBase
        .global intStackEnd
intStackBase:
        .space  8192, 0[11]
intStackEnd:
        .space  16, 0
intStackStart:
```

We declare an interrupt stack between `intStackBase` and `intStackEnd`.[12] The Xtensa ABI requires a stack alignment of 16 bytes, and we use an align directive to meet this requirement. We also declare a location labeled `userStackPtr` for saving the current stack pointer. Interrupt handlers will use the interrupt stack rather than the user stack. The user's stack pointer will be saved in `userStackPtr` for later restoration.

The first thing that we should do is to recognize that an interrupt has occurred. Recall that `a3` has already been saved away by the vector code into `EXCSAVE_1`.

Therefore, we start the User Exception Handler with the following sequence:

```
        .text
        .align  4
        .global _UserExceptionHandler
_UserExceptionHandler:
        rsr     a3, EXCCAUSE
```

11. The use of the `.space` directive here is not optimal. This directive allocates space in the `.data` section because `.space` can specify any value for initialization. Because this memory is being initialized to zero, the objects and resulting executables will be smaller if a `.comm` directive is used instead.

12. Note that 16 bytes are reserved beyond the interrupt stack to leave space for a copy of the user's base save area. See Chapter 5 for details on the base save area and Section 6.3.6 on page 99 for details on copying the base save area to the interrupt stack.

```
        beqi    a3, EXCCAUSE_LEVEL1INTERRUPT, handleInt
        rsr     a3, EXCSAVE_1
        break   1, 1

handleInt:
```

The first instructions of the handler follow the declaration of the interrupt stack, and the vector transfers control to the handler at `_UserExceptionHandler`. Although the code looks first to decode the cause of the exception, we are only going to handle interrupts at this point. Loading the `EXCCAUSE` into `a3`, we test `a3` against the `LEVEL1INTERRUPT` cause value (that is, 4) —jumping to `handleInt` when the test is true. If the test is not true, we restore `a3`[13] and execute a `break` instruction. Later this `break` instruction can be replaced with the appropriate code to handle the other exceptions. Until then, the break supplies a good guard against different types of exceptions, in case an error is introduced into the developing runtime.

6.3.4 Saving the Current State

Now that we are certain that the `UserExceptionVector` was reached via an interrupt, we begin preparing the processor to handle the interrupt. We must save address registers `a0` through `a15`—and special registers `EPC1`, `SAR`, `LBEGIN`, `LEND`, `LCOUNT`, and `PS`—when an interrupt is taken. These values are part of the interrupted code's context, and we must later restore them before returning to the interrupted code. We save these registers onto the top of the user's stack.

The memory layout in Figure 4 represents the state of the user stack after an interrupt has been received and the saving mechanisms run. The user's code was interrupted in stack frame `i`.

13. We restore a3 to adhere to `break` conventions. Refer to the ABI chapter of the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details on `break` instruction conventions.

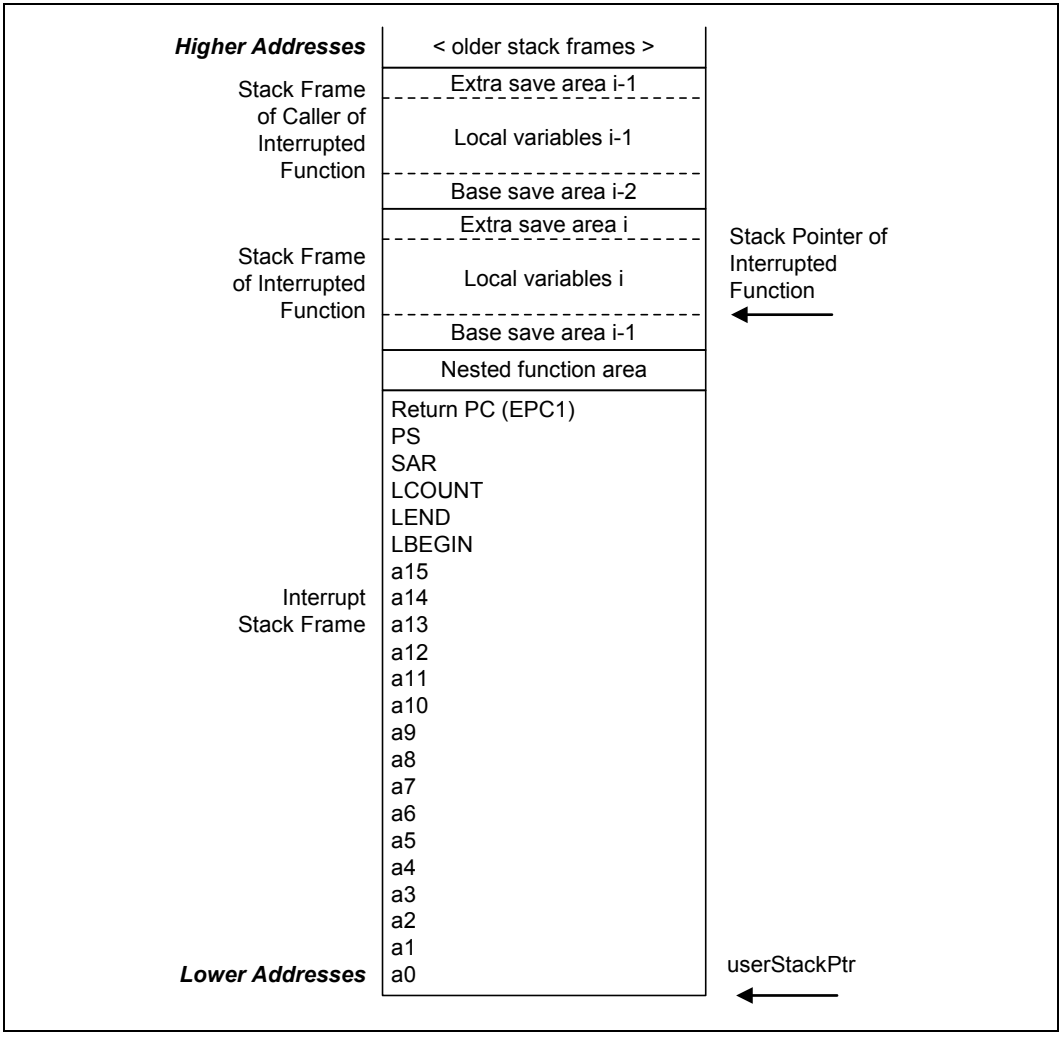

Figure 4. User Stack Layout

Interrupt and exception handlers must honor the base save area (16 bytes) and a nested function area (another 16 bytes) immediately below the current stack pointer. All stack frames have a base save area. Only the current function needs a nested function area; ancestor stack frames do not have a nested function area. See the ABI chapter in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for details on nested functions.

The following code saves the current logical window and certain special registers to the interrupt stack frame on the user stack, according to the order reflected in Figure 4. Each register is four bytes in size.

```
handleInt:
        addi    a3, a1, -FRAME_SIZE

saveIntRegs:
        s32i    a0, a3, 0
        s32i    a1, a3, 4
        s32i    a2, a3, 8
        s32i    a4, a3, 16
        s32i    a5, a3, 20
        s32i    a6, a3, 24
        s32i    a7, a3, 28
        s32i    a8, a3, 32
        s32i    a9, a3, 36
        s32i    a10, a3, 40
        s32i    a11, a3, 44
        s32i    a12, a3, 48
        s32i    a13, a3, 52
        s32i    a14, a3, 56
        s32i    a15, a3, 60
        rsr     a2, EXCSAVE_1
        s32i    a2, a3, 12
        rsr     a2, LBEG
        s32i    a2, a3, 64
        rsr     a2, LEND
        s32i    a2, a3, 68
        rsr     a2, LCOUNT
        s32i    a2, a3, 72
        rsr     a2, SAR
        s32i    a2, a3, 76
        rsr     a2, PS
        s32i    a2, a3, 80
        rsr     a2, EPC_1
        s32i    a2, a3, 84
        movi    a2, 0
        wsr     a2, LCOUNT
        isync

moveToStack:
```

Note that these registers are saved in the same format as described in Figure 4 on page 97. The interrupted code was, potentially, using all of the address registers as well as the looping special registers (`LBEG`, `LEND`, `LCOUNT`), the shift amount register (`SAR`), and `PS.CALLINC` (if the interrupt occurred between a `callN` and `entry` instruction). The address at which the processor was interrupted is saved in `EPC_1`. All of these registers are saved into the user's stack because all of their values are required to restore the processor to its state before the interrupt occurred.

The last three instructions clear LCOUNT. If application code and interrupt handlers call common functions, LCOUNT may otherwise cause problems when the program counter matches LEND.

6.3.5 Saving the User Stack Pointer

Because the interrupt stack will be used for servicing interrupts, the value of the user stack pointer will soon be overwritten. This stack pointer must be saved so that it may later be restored:

```
moveToStack:
        movi    a2, userStackPtr
        s32i    a3, a2, 0
```

The previously declared memory at userStackPtr is used for this purpose.

6.3.6 Moving the Base Save Area

We do not know if the window frame of the caller of the interrupted function is saved in memory or still located in a live window frame in the physical register file. Decoding this information from the current WINDOWSTART value is a complex operation, but we can avoid the whole chore.

We wish to use the interrupted function's stack frame as a dummy stack frame to link the user stack and the interrupt stack. We will change the stack pointer to address the interrupt stack frame.

The interrupted function's window frame may still be in the physical register file. Invocation of the interrupt handler may eventually cause a window overflow exception on the interrupted function's window frame. This saved information, which is a part of the user stack, will be saved into the interrupt stack because we changed the stack pointer to address the interrupt stack. Consequently, the interrupt return sequence must restore this base save area back into the user stack where it belongs. Failure to handle this situation will result in intermittent corruption of the first window pane of the interrupted function's window frame. Copying the base save area from the user stack to the interrupt stack guarantees that the base save area in the interrupt stack is always known to be valid.

There are four cases that may result from this situation:

1. If the window frame is in registers when the interrupt is taken—and remains in registers when the interrupt service routine completes—the user's data remains valid. The copying of the base save area is unnecessary, but harmless.
2. If the window frame is in registers when the interrupt is taken—and is in memory when the interrupt service routine completes—the copy of the base save area from the interrupt stack to the user stack keeps all of the user data intact.

3. If the window frame is in memory when the interrupt is taken—and is in memory when the interrupt service routine completes—copying the base save area from the user stack to the interrupt stack validates the copying of the base save area from the interrupt stack to the user stack. The user data remains intact.
4. The fourth case should never occur. If the window frame is in memory when the interrupt is taken, the window frame should never be in registers when the interrupt completes. The mechanism that transfers data from memory to registers is a window underflow. The interrupt service routine should never cause an underflow of this data because the interrupt service routine's first stack frame is deeper in the stack than this data.

The following code moves the base save area to the interrupt stack:

```
addi    a2, a1, -16
movi    a1, intStackEnd
l32i    a0, a2, 0
s32i    a0, a1, 0
l32i    a0, a2, 4
s32i    a0, a1, 4
l32i    a0, a2, 8
s32i    a0, a1, 8
l32i    a0, a2, 12
s32i    a0, a1, 12
```

We also load `a1` with the lowest address in the interrupt stack. This is the base save area in the interrupt stack.

Having set up the pointers, we copy the base save area from the user stack to the interrupt stack. Note that interrupts are disabled during this sequence.

6.3.7 Setting the Stack Pointer

Now that the base save area has been copied, the stack pointer can be set to the interrupt stack, as follows:

```
addi    a1, a1, 16
```

Note that the stack pointer now points to the base of the interrupt stack at `intStackStart`. Immediately below (toward lower addresses) are the 16 bytes copied from the base save area. After the interrupt service routine has been invoked, the layout of the interrupt stack will be as shown in Figure 5.

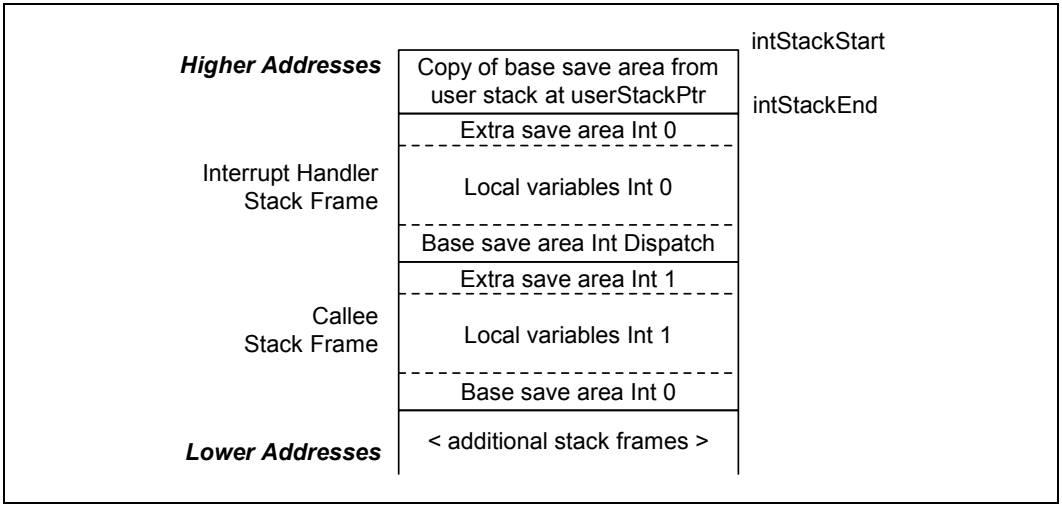

Figure 5. Interrupt Stack Layout

6.3.8 Calling the C Handler

A few PS fields need to change before we can call the C interrupt service routine. For context, Figure 6 repeats the format of the PS register.

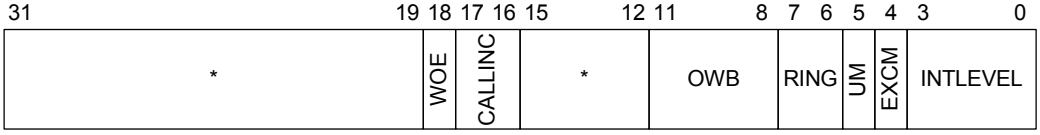

Figure 6. PS Register Format (XEA2)

We set PS.INTLEVEL to 1 to disable all other level-one interrupts. Note that in our handler, PS.INTLEVEL is now 0. When taking a level-one interrupt, the processor does not set PS.INTLEVEL to 1; instead, it remains at zero and the processor sets PS.EXCM to 1, which raises the effective interrupt mask to 1 (or more generally to EXCMLEVEL).

Nested level-one interrupts are possible, and we discuss them later in Section 6.5.7. For this example, all level-one interrupts will be disabled, and the interrupt handler will scan for active interrupts.

We also clear PS.EXCM to 0 to exit exception mode. (Section 3.1 on page 35 explains various PS fields that PS.EXCM overrides). We set PS.WOE to 1 to enable window overflows. PS.RING is 0 for all our examples. Finally, we clear PS.UM to send any exceptions that occur during interrupt handling to the KernelExceptionVector vector. PS.OWB and PS.CALLINC are unused in this context.

Note the following important point. We saved but did not corrupt a4..a15. This is critical because these registers may belong to other window frames that need to be saved by a window overflow.

The following code makes all necessary changes to PS.

```
invokeIntHandler:
        movi    a3, PS_WOE_MASK + 1
        wsr     a3, PS
        rsync
```

Register references following the rsync instruction may cause a window overflow exception now that PS.WOE is set. Recall that all of our stack manipulations are complete, and that a1 once again points to a valid stack. This is important because overflows will cause memory corruption if they occur when the stack is not valid.

The C routine to invoke, intDispatch, has the following prototype:

```
void intDispatch( void );
```

Because there are no arguments to pass, we just call the function.

```
        movi    a4, intDispatch
        callx4  a4

returnFromInterrupt:
```

The calling sequence is quite simple. Register a4 receives the address of the intDispatch routine, and we call the function. Note that using a call8 or call12 here is unsafe because the interrupted routine may not have allocated the required Extra Save Area in its stack frame.

6.3.9 Return Overview

With the interrupt handler invoked and completed, we must now resume execution of the processor in the state before the interrupt occurred. There are four steps to returning:

1. Return the processor to exception mode.
2. Copy the base save area back to the user's stack.
3. Restore the state of the processor.
4. Return from the interrupt.

Processor Exception Mode

Setting `PS.EXCM` to 1 puts the processor in exception mode and disables both level-one interrupts and window-overflow exceptions. The saved `PS` on the interrupt stack has `PS.EXCM` set, so we simply restore this value first. In exception mode, other processor registers are safe to manipulate while restoring the user context.

```
returnFromInterrupt:

        movi    a3, userStackPtr
        l32i    a3, a3, 0
        l32i    a4, a3, 80
        wsr     a4, PS
        rsync
```

The restoration of the `PS` register requires that we reload the user stack pointer, since all information related to the user's context was saved in that stack. With the stack pointer in `a3`, we can load the original value and restore it to `PS`. A trailing `rsync` instruction ensures that subsequent instructions occur in the context of the new `PS` value.

Moving the Base Save Area

Recall that the base save area was copied to the interrupt stack to handle the case where the registers of the interrupted function's caller were in memory when the interrupt occurred. The other possibility is that this window frame was not in memory, but that the execution of the interrupt handling caused a window overflow that saved it to memory. Consequently, we now copy this back into the user stack.[14]

```
        movi    a2, intStackEnd
        l32i    a0, a2, 0
        s32i    a0, a3, FRAME_SIZE - 16
        l32i    a0, a2, 4
        s32i    a0, a3, FRAME_SIZE - 12
        l32i    a0, a2, 8
        s32i    a0, a3, FRAME_SIZE - 8
        l32i    a0, a2, 12
        s32i    a0, a3, FRAME_SIZE - 4
```

Restoring the Processor State

The state of the processor must now be restored from the user stack.

14. This version of the handler does not yet support context switching.

This is done in a code sequence that is reminiscent of the saving code sequence:

```
l32i    a0, a3, 84
wsr     a0, EPC_1
l32i    a0, a3, 76
wsr     a0, SAR
l32i    a0, a3, 64
wsr     a0, LBEG
l32i    a0, a3, 68
wsr     a0, LEND
l32i    a0, a3, 72
wsr     a0, LCOUNT
isync

l32i    a0, a3, 0
l32i    a1, a3, 4
l32i    a2, a3, 8
l32i    a4, a3, 16
l32i    a5, a3, 20
l32i    a6, a3, 24
l32i    a7, a3, 28
l32i    a8, a3, 32
l32i    a9, a3, 36
l32i    a10, a3, 40
l32i    a11, a3, 44
l32i    a12, a3, 48
l32i    a13, a3, 52
l32i    a14, a3, 56
l32i    a15, a3, 60
l32i    a3, a3, 12
```

Note: For XEA1, the order of restoring the zero-overhead loop registers is significant. Restoring `LCOUNT` after restoring `LBEG` and `LEND` is safer. Otherwise, `LBEG` and `LEND` may become active before `LCOUNT` can be set. This issue does not exist for XEA2 because `PS.EXCM` is 1 at this point, disabling loops. This code is appropriate for both XEA1 and XEA2.

Returning from the Interrupt

With the return address having been placed in the `EPC_1` special register, returning from the interrupt is only a matter of executing the appropriate instruction. This instruction clears `PS.EXCM`, and jumps to the address in `EPC_1`.

```
rfe
```

6.4 The C Interrupt Handler

The C interrupt handler is somewhat anticlimactic compared with the assembly code required to invoke it. The handling code needs to find out which interrupt has been posted, and then take the appropriate action.

Before this can be done, there is one additional assembly function required. Currently, we have not yet provided a way to read the `INTERRUPT` special register from C. `INTERRUPT` tracks the currently pending processor interrupts. The inline function to read `INTERRUPT` is similar to the ones that read `CCOMPARE` and the other timer registers.

```
static __inline__ unsigned int read_interrupt()
{
    unsigned int interrupt;

    __asm__ __volatile__ (
        "rsr     %0, interrupt"
        : "=a" (interrupt)
        );
    return interrupt;
}
```

With this function in place, the C timer-interrupt handler appears as follows:

```
#include <xtensa/config/specreg.h>
#include <xtensa/config/core.h>
#include "interrupts.h"
#include "timer.h"

#define TIMER_INTERVAL 0x1000
#define TIMER_INT_MASK (1 << 6)

int system_ticks;

void intDispatch( )
{
    unsigned int ints = read_interrupt() & read_intenable();
    if( ints & TIMER_INT_MASK ) {
        unsigned long old_ccompare;
        unsigned long diff;

        do {
            system_ticks++;
            old_ccompare = read_ccompare0();
            set_ccompare0( old_ccompare + TIMER_INTERVAL );
            diff = read_ccount() - old_ccompare;
        } while ( diff > TIMER_INTERVAL );
    }
}
```

Let us consider this handler in detail (we list only pieces of the full function).

```
unsigned int ints = read_interrupt() & read_intenable();
```

The handler must determine which interrupts to process. The set of pending and enabled interrupts is determined by anding together the current contents of the INTERRUPT register with the INTENABLE value that was active at the time of the interrupt.

Three issues complicate the handling of the interrupt.

- First, interrupt processing could have been held off beyond the TIMER_INTERVAL. Simply adding TIMER_INTERVAL to the last CCOMPARE_0 value does not guarantee that the new timer is set correctly, if the code that disables interrupts takes more than TIMER_INTERVAL cycles. Ignoring this case can result in a timer-compare value that falls short of CCOUNT and thus requires far too long to reach (on the order of 4 billion cycles).
- Second, the timer continues to tick while the interrupt is being handled. Consequently, the CCOUNT value may pass the CCOMPARE_0 value between the time that the value to place in CCOMPARE_0 is computed, and the time that this value is set into the register.
- Third, the CCOUNT register may wrap at any time. This complicates the testing of whether CCOMPARE_0 is beyond CCOUNT.

All of these issues may be ignored if the software designer can guarantee that interrupt latency is less than the TIMER_INTERVAL. In the simple case, the processing of this interrupt consists of only a single C statement:

```
set_ccompare0( read_ccompare0() + TIMER_INTERVAL );
```

The same simplifying assumption was made earlier when first enabling the timer. In the general case, enabling the timer also requires code similar to the following, except that the initial CCOMPARE_0 value is set relative to CCOUNT. The following code deals with the more complex case:

```
unsigned long old_ccompare;
unsigned long diff;

do {
    system_ticks++;
    old_ccompare = read_ccompare0();
    set_ccompare0( old_ccompare + TIMER_INTERVAL );
    diff = read_ccount() - old_ccompare;
} while ( diff > TIMER_INTERVAL );
```

Systems that respond slowly to timer interrupts run the risk of missing one or more system ticks entirely. This handler uses a loop to detect delays in interrupt handling, account for any missed ticks, and select an appropriate new CCOMPARE_0 register value.

The number of system ticks is equal to the number of TIMER_INTERVAL cycles that have elapsed. The loop increments the system-tick counter, and then reads the previous CCOMPARE_0 register. The next CCOMPARE_0 value is the old one plus the interval. Writing to the CCOMPARE_0 register also clears the interrupt signal.

To detect missed system ticks, the loop computes the difference between the current CCOUNT register (which is always running) and the previous CCOMPARE_0 value. If the difference (variable diff) is greater than the constant timer interval, the system has missed a system tick. In this case, the loop executes again, accounting for another system tick and adjusting the CCOMPARE_0 register again.

Failure to detect missed system ticks and adjust CCOMPARE_0 accordingly would result in a CCOMPARE_0 value that CCOUNT has already passed. The next timer interrupt would not occur until the 32-bit CCOUNT wraps around again, on the order of 4 billion cycles.

Note that the loop computes the next timer interval based on the previous CCOMPARE_0 value, not the CCOUNT value. The CCOUNT register is always running; its value at interrupt-handling time is unpredictable due to system delays. Adding a timer interval to CCOUNT would introduce timer drift and variations in the delay between timer interrupts.

6.5 Nested Interrupt Handling

Nested interrupt handling is desirable when one interrupt is more important (or requires lower latency response) than another. This implies that nesting interrupts is useful only when combined with some type of interrupt prioritization. For example, strict prioritization among interrupts provides determinism, making it possible to analyze worst-case response using methods such as rate-monotonic analysis, i.e. where lower priority interrupts have no (or small and bounded) impact on the latency of higher priority interrupts.

6.5.1 Interrupt Prioritization

The first question involves the prioritization scheme: which interrupts pre-empt (nest over) which ones?

Normally, interrupts pre-empt those of a lower (but not same) priority level. This generally works as long as interrupt handlers run at the priority level (PS.INTLEVEL) of the corresponding interrupt. The interrupt dispatch code must ensure this. This implies that multiple interrupts configured at the same priority level cannot pre-empt, i.e. nest, each other. It is possible to work around this in software by manipulating the INTENABLE register to create a software defined prioritization among such interrupts.

The interrupt dispatch example of the previous section only considers low priority (level-one) interrupts. By itself, it has no provisions for nesting interrupts. However, medium-priority interrupts (see Section 3.3 on page 41) and high-priority interrupts (see Section 3.4 on page 41) can nest over level-one interrupts dispatched using the above code.

Medium-priority interrupts can be efficiently dispatched to a handler written in C. They can (and typically do) share the level-one interrupts' stack, or use their own. For the time being, this document provides no example of a medium-priority interrupt dispatcher[15].

High-priority interrupts cannot be efficiently dispatched to handlers written in C[16]. Each high-priority interrupt level's vector is almost always written in assembly. To ensure clean nestability, each level's handler must save and restore any state it touches to its own dedicated save area (and/or dedicated stack, if any). Thus, high-priority interrupt handling is independent of the code presented here, and is not discussed further.

6.5.2 Interrupt Nesting Example

In order to provide an example of interrupt nesting, the following sections modify the level-one interrupt handler presented previously. They implement a software prioritization.

The interrupt handler presented previously masks all level-one interrupts and thereby precludes one level-one interrupt from interrupting another. This interrupt handler also dispatches all interrupts to the same handling routine. A mechanism that allows separate handlers for each interrupt is easier to use when interrupt handling is interruptible.

Support for nested exception handling requires the following additions and changes to the interrupt handling mechanism:

- A means of disabling interrupts of lower or same priority than the one being dispatched, e.g. using PS.INTLEVEL if possible, or INTENABLE if necessary.
- Addition of an interrupt dispatching mechanism.
- Adding a mechanism to distinguish nested from non-nested interrupts. Our examples do this by switching to kernel vector mode when handling interrupts, so we now need to handle kernel exceptions.
- Dispatching of the nested interrupt.

15. The sources for XTOS can be examined instead. See the *Xtensa System Software Reference Manual* for details.

16. With the Call0 ABI, it is possible to dispatch high-priority interrupts to C code without excessive overhead. This is because the Call0 ABI involves less processor state and does not impose the possibility of exceptions, as opposed to the Windowed ABI where window, alloca, and syscall exceptions may occur. It is even possible, in a Windowed ABI application, to handle high-priority interrupts using the Call0 ABI, as long as the interrupt handler and application do not share any code. However, existing runtimes (such as XTOS) do not support such scenarios without modifications.

6.5.3 Interrupt Stack Use

When handling nested interrupts, we do not switch to a new stack. We continue using the interrupt stack, add interrupt stack frames, copy base save areas, and so forth. This is why we need to detect nested vs. non-nested interrupts.

This results in a slightly different stack layout for nested interrupts, as shown in Figure 7, compared with non-nested interrupts shown previously in Figure 4 on page 97.

This new layout also shows a new field named Software-priority Interrupt Mask. It is used for software prioritization of interrupts, discussed in the next section.

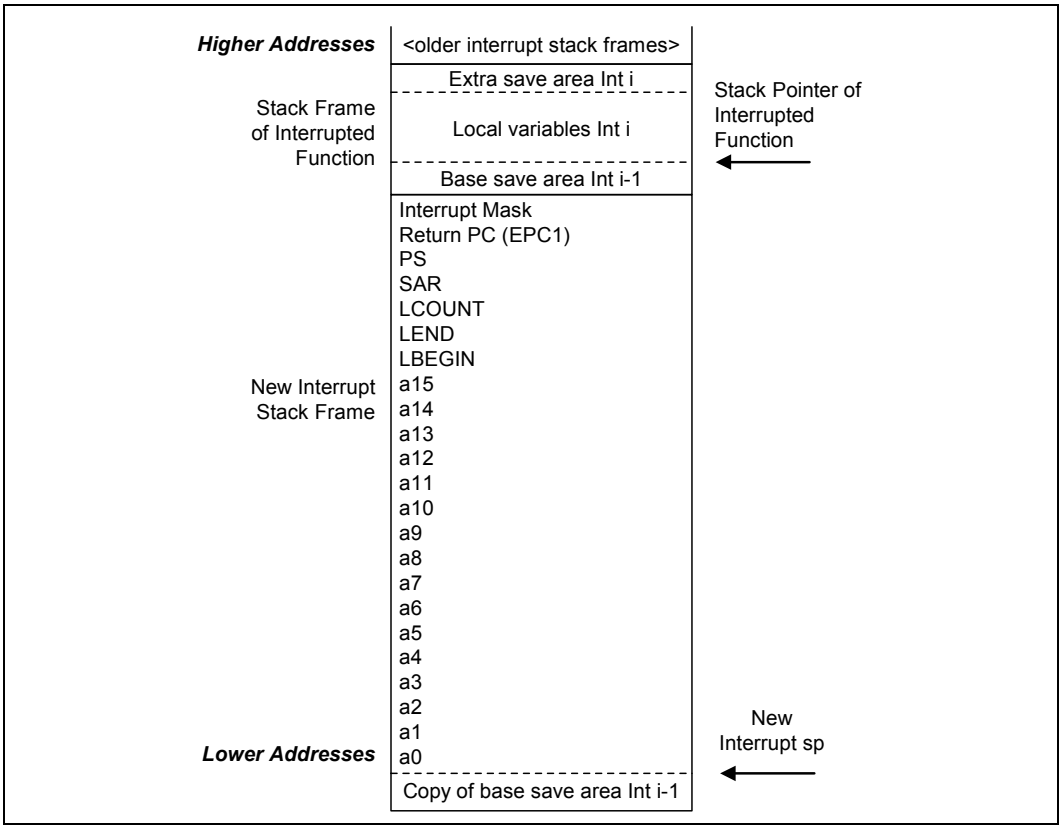

Figure 7. Nested Interrupt Stack Layout

6.5.4 Managing Software Prioritized Interrupts

Hardware prioritizes interrupts by priority level. Multiple interrupts can be configured to the same hardware priority level. Hardware does not prioritize among interrupts configured to the same priority level. They all go to the same interrupt vector, and it is up to

software to look at what interrupts are pending at that level and make a decision on which one(s) to handle. For example, it can just process the first one, process them all sequentially, or process interrupts using some other scheme.

If it choose to strictly prioritize among interrupts of the same hardware priority level, software must select the interrupt it considers to be of highest priority. It also needs a mechanism to enable interrupts within that level that are of higher priority and disable the rest. `PS.INTLEVEL`is insufficient for this task, because it only distinguishes hardware priority levels. The only alternative mechanism available to mask specific interrupts based on a software-defined priority is the `INTENABLE` register. However, the `INTENABLE` register normally plays the role of enabling and disabling individual interrupts independently of the processor's current priority level (`PS.INTLEVEL`). This conflicts with the role of a (software-defined) priority level.

To manage software prioritized interrupts, the `INTENABLE` register must play two roles.

- the current interrupt level: masking interrupts by software-defined priority level (like a simulated `PS.INTLEVEL`, but finer-grained), and
- the enable mask: enabling and disabling individual interrupts.

This is not possible using the contents of INTENABLE alone. For example, when an interrupt is masked because of the current interrupt level, we still need to track whether it is enabled based on the enable mask. The only way to implement these two roles is to maintain their two states separately, and to compute the correct value of `INTENABLE` based on these two independent values.

Note: Operating systems or kernels running on a processor configured with XEA1 must use a similar mechanism regardless of whether they prioritize interrupts in software.[17]

For the purpose of this example, we'll maintain these two states in two global variables or memory locations, declared as follows:

```
        .data
        .global intMasking
        .align  4
intMasking:
        .word   0xFFFFFFFF       // interrupt level mask
        .word   0x00000000       // individual enable mask
#define INT_LEVEL_MASK   0     // offset to the level mask
#define INT_ENABLE_MASK  4     // offset to the enable mask
```

17. In XEA1, `PS.INTLEVEL` cannot be used to control the processor interrupt level while running windowed C code, because window exceptions clear `PS.INTLEVEL` on return. Hence in practice, `PS.INTLEVEL` must remain zero in C code, and level-masking is implemented by virtualizing the `INTENABLE` register as described in this section.

Rather than define a different symbol for each variable, here we define only one symbol that can be used to access both variables. It turns out that this simplifies the code (and improves performance), because it allows us to load a single address in a register rather than two to access both variables. This reduces the number of registers, and thus register save and restore overhead, needed in the code that follows. If we needed to access these variables frequently from C, we might give each variable its own symbol as well.

Note also that the interrupt level is defined as a mask of which interrupts are enabled by the current level. This keeps the implementation simple. The `INTENABLE` register is kept equal to the intersection (bitwise `and`) of the two masks: the interrupt level mask, and the individual interrupt enable mask. Both masks behave identically in this respect. They are simply used differently.

For this method to work correctly, i.e. to ensure `INTENABLE` always reflects these two states, and for updates to these states to operate independently, any change to either state (and thus to `INTENABLE`) must be made *atomically*. That is, any interrupt whose handler might also modify these states must be disabled during any such update (until `INTENABLE` is written).

Atomicity ensures correctness. Otherwise, if some code reads/updates the two states and is interrupted just before writing `INTENABLE`, and the interrupt's handler also updates the states (e.g.. the enable mask), upon return from the handler the interrupted code will write its already-computed but now invalid value to the `INTENABLE` register.

This technique has a few implications. All accesses previously made to `INTENABLE` and `PS.INTLEVEL` must now be made to the corresponding global variables instead. The INTENABLE register must be recomputed anytime any of the two variables changes.

For example, where previously we saved and restored the interrupt level as part of the `PS` register, we must now also save and restore the software-defined current interrupt level mask variable which is now separate from `PS`. More on that later.

Also, where we previously manipulated `INTENABLE` directly, we must now manipulate the individual interrupt enable mask variable instead, and update `INTENABLE` accordingly. Thus, we rewrite the `enable_ints()` function as follows:

```
static __inline__ unsigned int enable_ints(unsigned int mask)
{
    extern unsigned int intMasking[2];
    unsigned int ret;
    unsigned int new_intenable;

    __asm__ __volatile__(
        "rsil    a15, 1        \n\t"
        "l32i    %0, %3, %5    \n\t"  /* INT_ENABLE_MASK */
        "or      %1, %0, %2    \n\t"
        "s32i    %1, %3, %5    \n\t"     /* INT_ENABLE_MASK */
```

```
        "l32i    %2, %3, %4    \n\t"     /* INT_LEVEL_MASK */
        "and     %1, %1, %2    \n\t"
        "wsr     %1, intenable \n\t"
        "wsr     a15, ps       \n\t"
        "rsync                 \n\t"
        : "=&a" (ret), "=&a" (new_intenable)
        : "a" (mask), "a" (intMasking),
          "i" (INT_LEVEL_MASK), "i" (INT_ENABLE_MASK)
        : "a15"
        );
    return ret;
}
```

6.5.5 Allowing Nested Interrupts via INTENABLE

The section of the original handler that prohibited nested level-one interrupts follows:

```
invokeIntHandler:
        movi    a3, PS_WOE_MASK + 1
        wsr     a3, PS
        rsync
```

The above code raises PS.INTLEVEL to 1, effectively disabling level-one interrupts.

To disable only the necessary subset of level-one interrupts, i.e. to prioritize among level-one interrupts in software, the previous code is replaced with the following new code:

```
invokeIntHandler:
        movi    a0, intMasking
        l32i    a2, a0, INT_LEVEL_MASK
        s32i    a2, a3, 88

        rsr     a2, INTERRUPT
        rsr     a3, INTENABLE
        and     a2, a2, a3
        beqz    a2, spurious
        neg     a3, a2
        and     a2, a3, a2

        //rsil  a3, LOCKOUTLEVEL
        movi    a3, ~XCHAL_INTLEVEL1_MASK
        addi    a2, a2, -1
        or      a3, a3, a2
        addi    a2, a2, 1
        s32i    a3, a0, INT_LEVEL_MASK

        l32i    a0, a0, INT_ENABLE_MASK
        and     a3, a3, a0
        wsr     a3, INTENABLE
```

```
movi    a0, 0

movi    a3, PS_WOE_MASK
wsr     a3, PS
rsync
```

This code is composed of six sections. The first section saves the previous interrupt level. The second computes the mask of the interrupt to be handled. The third computes a new fine-grained interrupt level based on that interrupt selection. The fourth updates `INTENABLE`. The fifth marks the base of the call stack, for debugging purposes. And the sixth is discussed in the next section.

Before we compute a new interrupt level mask, we have to save the previous one:

```
movi    a0, intMasking
l32i    a2, a0, INT_LEVEL_MASK
s32i    a2, a3, 88
```

Note that we associate the current interrupt level mask with the interrupt task's state. In other words, we save this mask in the same stack frame as the corresponding PS.

Now consider this code:

```
rsr     a2, INTERRUPT
rsr     a3, INTENABLE
and     a2, a2, a3
beqz    a2, spurious
neg     a3, a2
and     a2, a3, a2
```

The first three instructions compute a word that contains all unmasked pending interrupts. This mask is in `a2`. It may be zero, because external hardware can assert and deassert level-triggered interrupts at any time: a level-triggered interrupt may have deasserted after triggering the interrupt exception and before we read `INTERRUPT`. This is called a spurious interrupt, and is detected by the branch instruction.

The next two instructions change the word in `a2` into one that preserves only the least-significant set bit while clearing all others. This bit is the interrupt to be handled.[18]

18. The code sequence that turns a word into another word, that has only the least significant bit of the original word set, is somewhat cryptic. In the following discussion the least significant set bit of the original word is referred to as the LSB.

    To see how this code sequence works, consider the `neg` instruction as two steps. First, as subtracting one, and second, as taking the one's compliment. Consider the following example. Starting with `0x08000800`, we subtract one and get `0x080007ff`. Note that subtracting one carried out of the least significant bit. This leaves all bits below the LSB as one, clears the LSB, and leaves all the bits above the LSB intact. Negating this, we get the word `0xf7fff800`. Now all the bits above the LSB are negated, the LSB is set, and all the bits below are zero. This masking word is the result of the single `neg` instruction.

    Consider the bits in sets. All the bits above the LSB are negated in the masking word. Consequently, when the masking word is anded with the original word, zeroes result in all of these bits. The bits below the LSB in the original word are zero and so all of these bits in the resulting word will be zero. The LSB itself is set in both words and will therefore be set in the resulting word.

Note that the use of registers a4 through a15 is restricted in the code where window overflows are not enabled. Registers a4 through a15 may well be a part of one or more ancestor window frames that need to be saved to the stack, e.g. by window overflow exception(s) normally triggered by accessing them (when window overflows are enabled). Because the window frames have not been saved to memory, they must either be kept intact or restored prior to turning on window overflows (and interrupts).

Thus, registers a0..a3 are used, and we do not want to touch a4..a15.

The code above determined which interrupt to handle. As described in the last section, software assigns a priority to this interrupt.

We then compute the new interrupt priority level for the interrupt being handled:

```
//rsil  a3, LOCKOUTLEVEL[19]
movi    a3, ~XCHAL_INTLEVEL1_MASK
addi    a2, a2, -1
or      a3, a3, a2
addi    a2, a2, 1
s32i    a3, a0, INT_LEVEL_MASK
```

The software-defined current interrupt priority is represented as a bitmask of the interrupts that remain enabled at the current priority. In other words, it is a bitmask of interrupts of strictly higher priority than the current priority. The new priority is that of the interrupt we selected as the highest-priority level-one interrupt, whose corresponding bit is now stored in register a2. So our new interrupt priority mask needs to represent all interrupts of higher priority than the level-one interrupt whose bit is in a2.

Among level-one interrupts, we decided that lower numbered interrupts have higher priority (recall that we selected the highest priority interrupt as the least significant bit set). It turns out that by simply decrementing the single bit in register a2 (first addi instruction), we obtain a mask of all bits lower than that in a2, i.e. a mask of all higher priority level-one interrupts and potentially a few higher priority interrupts. The or instruction completes the mask by setting all bits that correspond to higher priority interrupts. This is done inverting the compile-time HAL macro XCHAL_INTLEVEL1_MASK, a mask of all level-one interrupts.

The second addi instruction restores a2. There were not enough registers to store the decremented value in another register, without having to save and restore that register.

19. The PS.EXCM bit is still set at this point, so level-one and medium-priority level interrupts are already disabled. So the subsequent sequence that modifies the interrupt level mask remains atomic with respect to these interrupts. This RSIL instruction is only needed if handlers for high-priority interrupts (of priority greater than EXCMLEVEL, up to LOCKOUTLEVEL) are allowed to modify the individual interrupt enable mask (e.g. to disable their own interrupt) or the current software-defined interrupt level mask (which must be saved and restored if modified).

Now that we've computed a new priority level mask in register a3, we can reflect this change in the INTENABLE register. As discussed in the previous section, INTENABLE is always the intersection (bitwise and) of the level mask and the individual enable mask:

```
l32i    a0, a0, INT_ENABLE_MASK
and     a3, a3, a0
wsr     a3, INTENABLE
```

Consider this next instruction:

```
movi    a0, 0
```

This instruction clears register a0, normally used to store function call return addresses, thus marking the base of the call stack as seen by a debugger. That is, if you debug an interrupt handler and request a stack traceback, you'll see a call stack that starts at the interrupt dispatcher, which calls the handler, and any other functions called by the handler at that point. You will not see the call stack of the code that was running just before the interrupt occurred. To see those, an alternative is to replace the movi with code that restores the interrupted code's a0 as follows:

```
movi    a0, userStackPtr
l32i    a0, a0, 0
l32i    a0, a0, 0
```

However, the above sequence still has the disadvantage that in the debugger's view of an interrupt handler's call stack, the interrupted function call is replaced by the interrupt dispatcher. To remedy this requires a longer sequence, not shown here, that creates a new call frame and computes that frame's a0 by adjusting the upper two bits of EPC1.

The atomic update of INTENABLE is now complete, and we can now enable the relevant interrupts. The following sequence does this and is discussed further in the next section:

```
movi    a3, PS_WOE_MASK
wsr     a3, PS
rsync
```

Note: It is tempting, for simplicity, to just mask (using INTENABLE, saving its original value and restoring it later) the one interrupt to handle rather than prioritize them as shown here. This allows interrupts to nest, without any specific priority among interrupts. One might either leave PS.INTLEVEL set to zero to immediately allow any other interrupt to nest (case A), or leave PS.INTLEVEL set to one and let the handler clear PS.INTLEVEL for lower priority interrupts (case B, effectively a primitive two-level software priority scheme). However, there are a number of caveats with this approach, in particular among all interrupts in case A, and among interrupt handlers that enable other interrupts in case B. Hence this approach is generally discouraged. The caveats include:

- Interrupts are not taken in a deterministic manner amongst each other. For example, if two such interrupts are triggered at about the same time, the first to be taken is often the last to complete, which can be undesirable and unexpected.
- Much fewer guarantees can be made about the time it takes to process an interrupt.
- Overall interrupt handling performance is worse. Nesting among interrupts dispatched this way increases the total interrupt processing time (by increasing the number of interrupt context saves and restores), and increases the average interrupt completion time, relative to simple round-robin non-nested scheduling.
- Interrupt handlers cannot easily disable their own interrupt, unless they modify the saved copy of `INTENABLE`. For example, a UART interrupt handler might disable itself after emptying its queue of characters to transmit.

6.5.6 Addition of an Interrupt Dispatching Mechanism

The previous interrupt dispatching mechanism was quite simple; a single function, `intDispatch`, was called with the following code:

```
        movi    a4, intDispatch
        callx4  a4
```

Having a single function that handles all interrupts is not a convenient way to structure code. Nested interrupt handling exacerbates this inconvenience. Adding an interrupt dispatching mechanism that can invoke a different interrupt handler for each interrupt eases the development of application software.

Such a dispatching mechanism is composed of two parts. The first is the dispatch table, and the second is the dispatch mechanism. Adding a dispatch table to the interrupt handling code is quite simple.

```
        .data
        .global intHandlers
intHandlers:
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
```

```
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
.word   intUnhandled
```

The dispatch table contains one word for each of 32 possible interrupts (although real code would use the `.rept` directive to define the same table more compactly). Each word holds the address of the function that is to handle the interrupt. Interrupts that are not handled have the address `intUnhandled` in their corresponding word.

With the table in place, the dispatching mechanism can be written.

```
movi    a3, PS_WOE_MASK
wsr     a3, PS
rsync

find_ls_one a4, a2

movi    a3, intHandlers
addx4   a4, a4, a3
l32i    a4, a4, 0
callx4  a4
```

There are several sections to the above code. First, we examine the restoring of the processor to interrupt level zero.

```
movi    a3, PS_WOE_MASK
wsr     a3, PS
rsync
```

The interrupt to be processed was masked using the INTENABLE register in a previous sequence. Here, the processor returns to interrupt level zero with window overflows enabled. Afterward, an interrupt can occur. We can now use registers a4 through a15 safely: not only have they been saved, but window overflow exceptions will now spill their preserved contents correctly to the stack if needed.

The work to dispatch the interrupt begins converting to an index the selected interrupt bit that remains in the a2 register. This word has a single bit set in the position of the interrupt to be processed.

```
        find_ls_one a4, a2
```

The <xtensa/coreasm.h> header file defines the find_ls_one macro to find the least-significant bit that is set. Using the Xtensa CHAL, this macro compiles into different code depending on the configuration of the processor. Using this macro, the number of the interrupt to be processed is put into the a4 register. This value is then used as an offset into the table.

Finally, we load the handler from the table and call it.

```
        movi    a3, intHandlers
        addx4   a4, a4, a3
        l32i    a4, a4, 0
        callx4  a4
```

Upon return from the interrupt handler, we load the stack pointer, retrieve the user's PS value, and write it to the PS register. This step restores the processor to exception mode, since PS.EXCM is set. As discussed in the last section, we must now also restore the software-defined interrupt priority level and update INTENABLE, effectively enabling the interrupt again.

```
returnFromInterrupt:
        movi    a3, userStackPtr
        l32i    a3, a3, 0
        l32i    a4, a3, 80
        wsr     a4, PS
        rsync

        movi    a5, intMasking
        l32i    a2, a3, 88
        //rsil  a4, LOCKOUTLEVEL
        s32i    a2, a5, INT_LEVEL_MASK
        l32i    a5, a5, INT_ENABLE_MASK
        and     a2, a2, a5
        wsr     a2, INTENABLE
```

Note that before this example of software-prioritized interrupts, we did not need to restore the interrupt level mask and update INTENABLE because we did not modify them. Previously, we simply raised PS.INTLEVEL to 1 to mask all other level-one interrupts.

Also note that the order of restoring PS and restoring INTENABLE is significant. When PS.EXCM is 1, all level-one (and medium priority level) interrupts are masked, and this needs to occur before re-enabling interrupts via INTENABLE. Conversely, if the RSIL instruction were needed (see the footnote in the previous occurrence of RSIL for details), it would take care of masking interrupts initially, and restoring PS would need to move *after* the update to INTENABLE.

Restoring all other state follows the same steps as before.

6.5.7 Handling Kernel Exceptions

The example code distinguishes nested and non-nested interrupts using the PS.UM bit. Another common alternative, not shown here but recommended for processors configured with XEA1[20], is to use a global variable.

At application startup, PS.UM is set. Thus, level-one interrupts and general exceptions that occur while the application is running go to the user exception vector. On dispatching the first timer interrupt handler, we clear PS.UM. Any level-one interrupt (or other general exception) that occurs while PS.UM is zero causes the processor to transfer control to the KernelExceptionVector. Thus, all nested level-one interrupts will go to the KernelExceptionVector.

Handling nested interrupts differs from the handling of non-nested interrupts. Using the two user and kernel vectors in the way we have shown conveniently separates these two cases without requiring software to distinguish between them and the action of branching from a common vector handler.

Handling kernel exceptions is analogous to handling user exceptions with few differences. Explaining the kernel exception handler here would be redundant. Refer to Appendix A for a listing of the kernel exception handler code. Note particularly that the code prepends the word “nested” to labels so they can be readily identified in the debugger examples that follow.

In some kernels, handling of nested and non-nested interrupts differ in another way: non-nested interrupts may invoke a task scheduler upon returning, in case that interrupt (or any nested interrupt during its execution) caused a change in task states. We’ll examine that case in the next chapter.

---

20. There exists an erratum for T1050.0 and T1040.2 and earlier processors configured with XEA1 that affects code running in kernel mode (PS.UM clear) with level-one interrupts enabled. The conditions that trigger this erratum can be avoided by always switching the processor back to user mode (PS.UM set) before or when enabling level-one interrupts. See the T1040 and T1050 Xtensa processor errata for details.

6.5.8 Examples of Nested Interrupts

With the run time written, the next step is writing a simple test. The Xtensa processor configuration being used here has a second timer, also on a level-one interrupt. This second timer can serve as an ideal mechanism with which to schedule and test a nested interrupt. Consider the following `main` routine:

```
#define TIMER_INTERVAL 0x1000
#define TIMER_INT_MASK  (1 << 6)
#define TIMER2_INT_MASK (1 << 10)
#define TWO_TIMERS_INT_MASK ( TIMER_INT_MASK + TIMER2_INT_MASK )

main()
{
        unsigned int ccnt = read_ccount();
        set_ccompare0( ccnt + TIMER_INTERVAL );
        set_ccompare1( ccnt + TIMER_INTERVAL + 200);
        enable_ints( TWO_TIMERS_INT_MASK );
        wait_func();
}
```

Here one timer is set to go off about 200 cycles after the other timer. Chances are that this will generate a nested interrupt. The first and second timer are associated with interrupts numbered 6 and 10, at priority levels 1 and 3, respectively. The second timer's interrupt has higher priority and can nest over the first.

The `wait_func` is a simple spinning routine that is here to generate another stack frame on the stack. It provides a good demonstration of how the interrupted call stack appears.

```
wait_func()
{
    while(1);
}
```

To actually handle these interrupts, the dispatching table must be changed to point to the routines that handle the timers. The first two timers use interrupts numbered 6 and 10 on this Xtensa processor configuration. As a consequence, we change table entries 6 and 10 to the handlers for these timers. The state of the table is as follows:

```
.global intHandlers
intHandlers:
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
```

```
        .word   timer0
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   timer1
        .word   intUnhandled
        .word   intUnhandled
```

Note the changes to the timer interrupt handler for timer 0:

```
void timer0( )
{
    unsigned long old_ccompare;
    unsigned long diff;

    do {
        system_ticks++;
        old_ccompare = read_ccompare0();
        set_ccompare0( old_ccompare + TIMER_INTERVAL );
        diff = read_ccount() - old_ccompare;
    } while ( diff > TIMER_INTERVAL );
}
```

There is one main difference from the previous `intDispatch` routine. The `timer0` handler is only invoked for interrupt 6 and does not need to test to see which interrupt should be processed.

The `timer1` routine has two differences from `timer0` above. First, it manipulates `CCOMPARE_1`. Second, it does not increment the `system_ticks` global variable. The changes are shown below:

```
void timer1( )
{
    unsigned long old_ccompare;
    unsigned long diff;

    do {
        old_ccompare = read_ccompare1();
        set_ccompare1( old_ccompare + TIMER_INTERVAL );
        diff = read_ccount() - old_ccompare;
    } while ( diff > TIMER_INTERVAL );
}
```

We can observe the results by using `xt-gdb` to connect to the simulator.

The timer interrupts are the interesting events here, so we set breakpoints on their handlers. Using `xt-gdb`, and we set breakpoints at `timer0` and `timer1`, with the following commands:

```
(xt-gdb) break timer1
Breakpoint 1 at 0x60000665: file timer.h, line 40.
(xt-gdb) break timer0
Breakpoint 2 at 0x6000063c: file intdisp.c, line 14.
```

Running to the first break is shown below.

```
(xt-gdb) run
Starting program: exec
Switching to remote protocol
Remote debugging using localhost:55225

Breakpoint 2, timer0 () at intdisp.c:14
14              system_ticks++;
```

The first interrupt to occur is the `timer0` interrupt. (Note that in the `main` routine, the `timer0` interrupt was set to occur several cycles earlier than the `timer1` interrupt.)

The `main` routine was running in the `wait_func` function when an interrupt occurred. The interrupt handler uses the stack frame for the `wait_func` function to process the interrupt dispatch.

```
(xt-gdb) where
#0  timer0 () at intdisp.c:14
#1  0x600004a0 in returnFromInterrupt ()
#2  0x600003d2 in main () at main.c:11
#3  0x40000246 in reset_exit ()
```

The call stack reflects this operation. After `main`, we see that the interrupt handler has used the `wait_func` stack frame to link to the interrupt stack. As a consequence, `returnFromInterrupt` is listed in its place.

Continuing `xt-gdb`, the second interrupt, now from the second timer, occurs:

```
Breakpoint 1, timer1 () at timer.h:40
40              );
```

`xt-gdb` breaks at the interrupt handler for the second timer interrupt. The call stack is as follows:

```
(xt-gdb) where
#0  timer1 () at timer.h:40
#1  0x600005c1 in nestedReturnFromInterrupt ()
#2  0x600004a0 in returnFromInterrupt ()
#3  0x600003d2 in main () at main.c:11
#4  0x40000246 in reset_exit ()
```

Note that the stack frame for `timer0` has been interrupted and has been used as the linking stack frame to dispatch `timer1`, as shown by the presence of `nestedReturnFromInterrupt` in the stack frame of `timer0`.

7. Context Switching

Discussion of context switching for Xtensa processors is inseparably intertwined with discussions of context switching in general. This chapter presents a context switching framework to highlight the particulars of Xtensa processors. However, a context switching framework for illustrative purposes need not be complex. The only features necessary are creation of tasks and simple, round-robin scheduling. We present such a scheduler in the following sections. This runtime is not representative of a useful real-time operating system, but does have the same requirements on the processor support software. Also note that the details are applicable only for Xtensa processors with Xtensa Exception Architecture 2 (XEA2).

7.1 Task Creation

The stack layout specified in Chapter 6 for interrupts on the user stack contains all of the information included in the context of a particular task. This includes all of the address registers, the relevant special registers, and the return address. In defining the interrupt stack, the specifics of the context have been defined.

You must be able to create tasks. For the operating system, this is equivalent to the initial creation of a context where the user supplies the stack and the entry point. In the following example code, you may also supply an argument value. Consider the C types involved here:

```
typedef void (TaskFunc)(void *arg);
```

An entry point for the simple task is a function that takes a void * argument and returns nothing. This function is not allowed to return. This is the type of function that you will pass to the task creation call.

```
void add_task( TaskFunc *entry, void *stack,
               unsigned int stack_size, void *arg );
```

The `add_task` function adds a task to the system. You must supply this function with the entry point, the pointer to the stack,[21] the size of the stack, and the argument to pass to the task. The `add_task` function must do two things. It must prepare the stack for the task and it must register the task with the runtime.

21. The pointer to the stack is the lowest memory address of the stack memory region. It is not the initial stack pointer.

The runtime keeps information about the tasks that are active in the system. In a standard operating system or kernel, tasks are represented by structures called task control blocks (TCBs). This simple runtime does not require any information about a task except the stack pointer. Due to this simplicity, the runtime can track the active tasks with the following global variables:

```
unsigned int *task_stacks[MAXTASKS];
unsigned int allocated_tasks = 0;
int current_task = -1;
```

The `task_stacks` array contains the stack pointers of all tasks that are active in the system. The `allocated_tasks` global variable tracks the total number of tasks active in the system. The `task_stacks` array has a valid stack pointer for entries 0 through `allocated_tasks-1`. The `current_task` variable is the index of the task that is currently running.

The `add_task` function performs three basic functions: computation of the initial stack pointer, initialization of the stack, and registration of the stack pointer with the global data structures. (It could also check against `MAXTASKS` and verify a minimum stack size, returning error codes, as appropriate.)

```
void add_task( TaskFunc *entry,
               void *stack,
               unsigned int stack_size,
               void *arg )
{
    unsigned int *sp;
    unsigned int stack_words;
    int i;

    sp = (unsigned int *)((int)(stack + stack_size) & 0xfffffff0);
    sp = sp - FRAME_SIZE / 4;

    for( i = 0 ; i < FRAME_SIZE / 4; i++ )
        sp[i] = 0;

    sp[ FRAME_PC/4 ] = (unsigned int)entry;
    sp[ FRAME_PS/4 ] = PS_WOE_MASK  | PS_UM_MASK |
                       PS_EXCM_MASK | (1 << PS_CALLINC_SHIFT);
    sp[ FRAME_AR(6)/4 ] = (int)arg;
    sp[ FRAME_AR(1)/4 ] = (int)(sp + FRAME_SIZE / 4);

    task_stacks[allocated_tasks++] = sp;
}
```

The add_task function addresses each of these in turn. Consider first the manipulation of the stack pointer. The alignment of the stack pointer passed and the size of the stack is not restricted. The Xtensa ABI requires a 16-byte alignment on the stack. The following excerpt of add_task computes the starting stack pointer:

```
sp = (unsigned int *)((int)(stack + stack_size) & 0xfffffff0);
sp = sp - FRAME_SIZE / 4;
```

In the first statement, the stack pointer is set to the highest 16-byte boundary that is within the block of memory passed to add_task. The highest section of this memory is used because Xtensa stacks grow down toward lower addressed memory. Aligning the stack pointer also aligns the stack to a 16-byte boundary. The stack pointer, as expected by the runtime, has been decremented from the top to add space for the context information. For this reason, the FRAME_SIZE is subtracted from the stack pointer.[22]

Now that the top of the stack has been established, the stack must be initialized for use. This initialization is done in two steps. First, the area of the stack that holds the context information is completely cleared. Most of the values in this area will find zero to be a valid value. Next, values are set into the context area for those portions of the context for which zero is not a valid value.

```
for( i = 0 ; i < FRAME_SIZE / 4 ; i++ )
    sp[i] = 0;
```

Clearing is simple. Setting the valid values is more complex. Consider these statements, one at a time:

```
sp[ FRAME_PC/4 ] = (unsigned int)entry;
```

The task will begin execution at the entry point that you have specified. This entry point is the starting PC of the context.

Now we must set the PS register fields to the values we want the user task to have when it regains control. The only exception is PS.EXCM, which must be set to 1, because that is what the context restoration code requires. When the exception handler executes an rfe instruction, it clears PS.EXCM and leaves all other PS fields unmodified.

Every function compiled for the windowed register calling convention will start with an entry instruction. This entry instruction expects that PS.CALLINC has been initialized by the appropriate call instruction. Because there is no actual information to be saved in the base registers, and we do not want to allocate space for an extra save area, we set PS.CALLINC to 1, which indicates the fewest number of registers in the window frame. Also, the PS.WOE field must be set to enable window overflows, the PS.UM field must be set to send general exceptions to the UserExceptionVector, and the

22. Keep in mind that sp is an unsigned int pointer while FRAME_SIZE is in bytes. As a consequence, all indices of byte offsets must be divided by sizeof[unsigned int] to make them word offsets.

PS.INTLEVEL field must be zero to enable all interrupts. Note that tasks run in privileged mode (PS.RING==0). Additionally, PS.EXCM must be set to override PS.WOE and PS.INTLEVEL until the rfe instruction. All of this is done in the following statement:

```
sp[ FRAME_PS/4 ] = PS_WOE_MASK  | PS_UM_MASK |
                   PS_EXCM_MASK | (1 << PS_CALLINC_SHIFT);
```

The function will act as if it were called by a CALL4 because PS.CALLINC was set 1. Since we did not supply a return address in a4, the function cannot return. Register a6 will be a2 in the called function. Consequently, the argument value must be set into a6:

```
sp[ FRAME_AR(6)/4 ] = (int)arg;
```

Finally, the stack pointer must be saved:

```
sp[ FRAME_AR(1)/4 ] = (int)(sp + FRAME_SIZE / 4);
```

7.2 Handling the Tick

Operating systems that implement preemptive time slicing run a small set of code on a regular interval that determines if the active task should be put to sleep and another task run in its place. This code is run on a regular interval by installing it as the interrupt service routine for a timer interrupt. Timer0 will be used in the example system to execute this code. This interrupt is called the tick; the code that is called when the interrupt lapses is called the tick handler.

The tick handler must reset the timer just as in the previous interrupt handling examples. Consequently the interrupt handler remains largely unchanged.

```
void timer0( )
{
    unsigned long old_ccompare;
    unsigned long diff;

    do {
        system_ticks++;
        old_ccompare = read_ccompare0();
        set_ccompare0( old_ccompare + TIMER_INTERVAL );
        diff = read_ccount() - old_ccompare;
    } while ( diff > TIMER_INTERVAL );

    task_tick();
}
```

The only change is the invocation of the task_tick routine at the end of the handler.

The job of the tick handler is to determine the next task to run. In the example system, the tick handler (which is serving as the scheduler) will use a simple round robin policy. The current task will cease to execute and the next task will begin to execute.

The tick handler must communicate information to the switching mechanism about which task is to stop running and which task is to begin running. The existing interrupt handling mechanism has a `userStackPtr` global variable. This variable holds the stack pointer of the task that was interrupted. This is the stack of the task that will cease to run.

Communication of the new task to run will be done with a new global variable called the `newUserStackPtr`. The tick handler will set this to the context of the new task to run. (This is the same stack pointer that was created by the `add_task` function.) We declare `newUserStackPtr` in C near the tick handler. Also note that we moved the declaration of `userStackPtr` from an assembly file to the same C file. The tick handler begins with the following extern declarations:

```
int *newUserStackPtr;
int *userStackPtr;
```

We also need to initialize the value of `newUserStackPtr`. If an interrupt handler doesn't select a new task to run, we need to resume the current task. At the point where `userStackPtr` is set to the current task's stack, we simply set `newUserStackPtr` to the same value, as follows.

```
moveToStack:
        movi    a2, userStackPtr
        s32i    a3, a2, 0
        movi    a2, newUserStackPtr
        s32i    a3, a2, 0
```

Now back to the tick handler. Recall that three global variables were declared to track the state of the tasks active in the system. The `task_stacks` array contains all of the stack pointers. The `allocated_tasks` array has the number of tasks active in the system. Finally, the `current_task` index contains the index of the task to run. Note that the declaration of the `current_task` index initializes its value to –1.

This is the tick handler:

```
void task_tick( void )
{
    if( current_task >= 0 )
        task_stacks[current_task] = userStackPtr;

    if( ++current_task == allocated_tasks )
        current_task = 0;
    newUserStackPtr = task_stacks[current_task];
}
```

This handler is really just a simple scheduler. In a prioritized, preemptive operating system, the scheduler would be examining the priorities of tasks that are ready to run. Depending on the inter-process communication mechanisms available, the tick handler would also be examining internal operating system data structures to determine if tasks that are asleep have become ready to run.

In this case, the scheduling policy is quite simple: the current task is put to sleep and the next task is awakened. Doing this requires three simple steps.

```
if( current_task >= 0 )
    task_stacks[current_task] = userStackPtr;
```

1. First, the current stack pointer is archived. This is done only when the task being swapped out is a “real” task for the system. The code run coming into the reset vector is running on a booting task. This is not a “real” task for the operating system. Different operating systems will deal with this in different ways. In this case, once the task starts, the booting code ceases to run and is never run again.

   This is the reason for the guard condition on the assignment of `task_stacks[current_task]`. The current stack pointer of a task must be archived when that task ceases to run. The assignment statement performs that archive operation. However, the very first time the tick routine is invoked, the currently executing code is the booting code rather than a task. By initializing `current_task` to –1, and by guarding this assignment, the operating system ignores the archiving step for this first task. Note that the tick interrupt should be neither taken, nor handled until the system is booted. Because the reset sequence exits with interrupts disabled, choice of when to start the tick is naturally left to the OS or user code.

2. Second, the tick handler must determine the next task to run after the current task’s state has been archived.

   ```
   if( ++current_task == allocated_tasks )
       current_task = 0;
   ```

   The simple task scheduling policy to be used allows the simple modulo increment of the `current_task` index, which is exactly what the code above does.

3. Third, the tick handler communicates the new task to run to the switching mechanism.

   ```
   newUserStackPtr = task_stacks[current_task];
   ```

   In this case, the switching mechanism will examine the `newUserStackPtr` variable to determine which context to switch.

7.3 Switching the Context

Preemptive context switching is somewhat anticlimactic, once basic and nested interrupt handling is properly done. There is no need to revisit the entire interrupt handler, only to review a few things and highlight the changes that turn the non-nested interrupt handler into a context switching mechanism.

First, recall that the `timer0` interrupt is registered as the interrupt handler for the first timer. The code for this handler was discussed previously. Next, remember that the interrupt handler worked in three basic sections: saving context, invoking the interrupt handler, and restoring context.

Changing the interrupt handler to support context switching involves changing the context restoration mechanism to examine the `newUserStackPtr` variable and to switch contexts when the variable value has changed.

```
returnFromInterrupt:

        movi    a4, userStackPtr
        l32i    a3, a4, 0

        l32i    a2, a3, 80
        wsr     a2, PS
        rsync
```

The first change is made to the register allocation in the code that returns the processor to a protected state. Note that the code sequence exits with `userStackPtr` in a3 for use in subsequent code.

Also note that the restore begins as if it was returning to the same task. The contents of the registers and the base save area that were moved to the interrupt stack are a part of the context for the task that is being put to sleep. (This is the task that is represented by the `userStackPtr`.) This information must be placed in the stack for the task that is being put to sleep. To be able to perform this operation, the base save area must be moved back (same code as before; not shown here) and the physical register file must be saved to the stack. Before saving the physical register file to the stack, we must first restore a4..a15.

```
arRestore:
        l32i    a4, a3, 16
        l32i    a5, a3, 20
        l32i    a6, a3, 24
        l32i    a7, a3, 28
        l32i    a8, a3, 32
        l32i    a9, a3, 36
        l32i    a10, a3, 40
        l32i    a11, a3, 44
```

```
        l32i    a12, a3, 48
        l32i    a13, a3, 52
        l32i    a14, a3, 56
        l32i    a15, a3, 60
```

The code to handle nested interrupts restored the special registers before restoring the address registers. Because there is no reason that the special registers must be restored first, implementation of the context switch is streamlined—in this case—by restoring `ar` registers first.

Note that a label has been added to the start of this sequence. This foreshadows the restoration of the address registers for both the task that is going to sleep and the task that is beginning execution. Later code may jump back to this point.

```
        movi    a2, userStackPtr
        movi    a1, newUserStackPtr
        l32i    a0, a2, 0
        l32i    a1, a1, 0
        beq     a0, a1, noSwitch
```

With a4 through a15 restored, the code is now ready to determine if a switch is to occur. If the contents of the `userStackPtr` and the `newUserStackPtr` are different, the context is switched. If they are equal, the context is not switched.

```
doSwitch:
        s32i    a1, a2, 0 // userStackPtr = newUserStackPtr

        l32i    a1, a3, 4 // restore current task's sp
        call0   xthal_window_spill_nw

        movi    a3, userStackPtr
        l32i    a3, a3, 0
        j       arRestore
```

Four steps occur. First, the contents of `newUserStackPtr` (in a1) are stored into the `userStackPtr` variable. This updates the state of the system and represents that the context has been switched. (Actually, the context is not completely switched until the `rfe` instruction at the end of exception handling. However, interrupts are disabled until then, so we are okay.)

The steps that follow destroy the contents of the working registers. Consequently, this step is performed first. Since interrupts are masked, this inverted order of operation is technically correct.[23]

23. High-priority interrupt handlers generally do not interact directly with user tasks and should avoid doing anything that requires knowledge of this inversion order and the window of uncertainty.

```
doSwitch:
        s32i    a1, a2, 0
```

Second, we save the physical register file to memory.

```
        l32i    a1, a3, 4
        call0   xthal_window_spill_nw
```

The HAL routine `xthal_window_spill_nw` saves the live window frames in the physical register file to the stack, and requires a valid stack pointer. The value of the stack pointer of the task going to sleep has been destroyed by the previous code sequence. The load instruction restores the stack pointer. See the *Xtensa System Software Reference Manual* for details on `xthal_window_spill_nw`.

Third, after all of the data for the task going to sleep is saved into its stack, the context switcher sets the stack pointer to the new task's stack, and jumps to `arRestore` to begin the restoration of the new task's state.

```
        movi    a3, userStackPtr
        l32i    a3, a3, 0
        j       arRestore
```

A second invocation of the code to detect a switch will determine that no switch has occurred.

```
        movi    a2, userStackPtr
        movi    a1, newUserStackPtr
        l32i    a0, a2, 0
        l32i    a1, a1, 0
        beq     a0, a1, noSwitch
```

This time through the code, the `noSwitch` branch will be taken.

```
noSwitch:
        l32i    a2, a3, 80
        wsr     a2, PS
        rsync
```

Fourth, the `PS` of the returning task is restored. All `PS` fields are set properly for running the user code, except for `PS.EXCM`, which is set to 1 to keep the processor in exception mode. The ending `rfe` instruction will clear `PS.EXCM`.

```
        l32i    a2, a3, 84
        wsr     a2, EPC_1
        l32i    a2, a3, 76
        wsr     a2, SAR
        l32i    a2, a3, 72
```

```
wsr     a2, LCOUNT
l32i    a2, a3, 68
wsr     a2, LEND
l32i    a2, a3, 64
wsr     a2, LBEG
isync
```

Restoration of the special registers is followed by restoration of the remaining `ar` registers and the return from exception.

```
l32i    a0, a3, 0
l32i    a1, a3, 4
l32i    a2, a3, 8
l32i    a3, a3, 12

rfe
```

7.3.1 Notes on Context-Switching Address Registers

In general, when context-switching address registers in the windowed ABI, you need to consider not only registers `a0` thru `a15`, but also all live caller frames that are still in physical address registers (in `ar0..ar63`[24]). Most or all of these caller frames are outside the current subset of 16 registers (indexed by `WindowBase`) accessible as `a0` thru `a15`. Rather than save and restore all 32 or 64 physical address registers, it may be more efficient to write out the caller frame registers to where they belong on the stack. This is done above by calling the HAL window spill routine (`xthal_window_spill()`, or in this case its `*_nw` variant from assembler code). What remains to save and restore are `a0` thru `a15`. Later, after restoring `a0` thru `a15`, any caller frame gets restored as needed by the usual window underflow mechanism when returning from a function.

So what about `WindowBase` and `WindowStart`? Note that after a window spill, that is, once any live callers are pushed out to the stack, `WindowStart` has just one bit set (the one indexed by `WindowBase`) because the current frame is the only live frame. Also note that, knowing there are no live caller frames, we can restore `a0` thru `a15` at any other value of `WindowBase` (in some other part of the physical register file `ar0..ar63`), as long as the corresponding `WindowStart` bit is set. That is, when only the current frame is live, we have: `WindowStart` = `1` << `WindowBase`. For example, we could just set `WindowBase` = 0 and `WindowStart` = `1`; or, if we've just done a window spill, and know that (`WindowStart` == `1` << `WindowBase`), we can leave them as is; and then just restore `a0` thru `a15`.

In other words, there is generally no need to save and restore the `WindowBase` and `WindowStart` registers in a normal context-switch.

24. Assuming 64 address registers; or, `ar0..ar31` on Xtensa processors configured with 32 address registers.

There are exceptions to this, although usually avoided. In particular, the situation is different with high priority interrupts, which can interrupt exception handlers and spill routines, such that caller frames cannot be reliably spilled from the values of `WindowBase` and other registers at hand. In such cases, it may be necessary to save and restore the entire physical address register file, as well as `WindowBase` and `WindowStart`: not just for context-switch, but also to be able to call any windowed ABI compliant functions such as C code. Note that in general, OS ported to Xtensa do not support the use of high priority interrupts, either at all or at least not to directly cause a context-switch, for reasons such as this one.

7.4 Using and Testing the Switch

Consider the use of this code. The following `main` routine performs two steps. First, it creates two tasks. Next it sets and enables the `timer0` interrupt. When this interrupt is taken, the task tick routine gets control and `main` will never again run.

```
main()
{
    add_task( task1, task1_stack, STACK_SIZE, (void *)10 );
    add_task( task2, task2_stack, STACK_SIZE, (void *)20 );

    set_ccompare0( read_ccount() + TIMER_INTERVAL );
    enable_ints( 0x00000001 );
    wait_func();
}
```

There are several ways that code performing a context switch can be flawed; testing all of these ways is a long and laborious process. Stack construction is prone to errors. The following test performs a series of deep calls. The calls go deep, so that the window frames are forced to overflow and underflow.

This code tests that the stack was laid out correctly. It also tests that the local registers used by the compiler for this function are saved and restored correctly. While this does not guarantee that all context is saved and restored correctly, it does suggest that most remaining errors will be in subsets of the context rather than the whole context.

```
int global_fool_the_optimizer = 0;
int failed = 0;

void deep_call( int depth )
{
    int a = depth;
    int b = depth + 1;
    int c = depth + 2;
    int d = depth + 3;
```

```
    if( global_fool_the_optimizer )
    {
        a = b = c = d = 10;
    }

    if( depth > 0 )
        deep_call( depth - 1 );

    if( a != depth ||
        b != depth + 1 ||
        c != depth + 2 ||
        d != depth + 3)
    {
        failed++;
    }
}
```

This test routine calls itself recursively to a maximum call depth of `depth`. Before each call, it sets four local variables; after each call it checks that those variables have the correct values. This routine is testing for errors in the stack or in the context switching.

The use of the `global_fool_the_optimizer` variable in this test code prevents the compiler from optimizing away the checks of the local variables. Otherwise, the compiler will recognize that variables a, b, c, and d have invariant values, and it will remove the check of the values after the call.

Construction of the tasks themselves is quite simple.

```
unsigned char task1_stack[STACK_SIZE];
void task1( void *arg )
{
    while( 1 )
        deep_call( (int) arg );
}

unsigned char task2_stack[STACK_SIZE];
void task2( void *arg )
{
    while( 1 )
        deep_call( (int) arg );
}
```

Each task is identical.[25] Each continually calls `deep_call` with the argument that was passed to it. Recall from the `main` routine that `task1` was passed an argument of 10 and `task2` was passed an argument of 20.

25. The same entry point could have been used for both tasks. The different entry points were constructed only to aid in clarity in the running examples. The tasks must have different stacks.

There are three interesting points in the execution of this test: the invocation of `task1`, the invocation of `task2`, and the deepest point in `deep_call`. The following `xt-gdb` commands set breakpoints at these conditions:

```
(xt-gdb) break task1
Breakpoint 1 at 0x6000040f: file main.c, line 41.
(xt-gdb) break task2
Breakpoint 2 at 0x6000041b: file main.c, line 48.
(xt-gdb) break deep_call
Breakpoint 3 at 0x600003cb: file main.c, line 13.
(xt-gdb) condition 3 depth == 0
(xt-gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x6000040f in task1 at main.c:41
2   breakpoint     keep y   0x6000041b in task2 at main.c:48
3   breakpoint     keep y   0x600003cb in deep_call at main.c:13
        stop only if depth == 0x0
```

Running the executable results in a break at `task1`:

```
Breakpoint 1, task1 (arg=0xa) at main.c:41
41      deep_call( (int) arg );
(xt-gdb) where
#0  task1 (arg=0xa) at main.c:41
```

Unless the timer interrupts occurred at exceedingly small intervals, one would expect the next break to occur at the deepest point in the `deep_call` function:

```
(xt-gdb) c
Continuing.

Breakpoint 3, deep_call (depth=0x0) at main.c:13
13      int a = depth;
(xt-gdb) where
#0  deep_call (depth=0x0) at main.c:13
#1  0x600003ec in deep_call (depth=0x1) at main.c:24
#2  0x600003ec in deep_call (depth=0x2) at main.c:24
#3  0x600003ec in deep_call (depth=0x3) at main.c:24
#4  0x600003ec in deep_call (depth=0x4) at main.c:24
#5  0x600003ec in deep_call (depth=0x5) at main.c:24
#6  0x600003ec in deep_call (depth=0x6) at main.c:24
#7  0x600003ec in deep_call (depth=0x7) at main.c:24
#8  0x600003ec in deep_call (depth=0x8) at main.c:24
#9  0x600003ec in deep_call (depth=0x9) at main.c:24
#10 0x600003ec in deep_call (depth=0xa) at main.c:24
#11 0x60000415 in task1 (arg=0xa) at main.c:41
```

Note that this was the next break. Also note that all is as expected here. The call has gone ten deep. Because this is the first execution, the failed count could not have incremented yet, since failed is checked upon return. Continuing brings us back to `deep_call` and shows that, without an intervening context switch, the failed count does not increment:

```
Breakpoint 3, deep_call (depth=0x0) at main.c:13
13      int a = depth;
(xt-gdb) where
#0  deep_call (depth=0x0) at main.c:13
#1  0x600003ec in deep_call (depth=0x1) at main.c:24
#2  0x600003ec in deep_call (depth=0x2) at main.c:24
#3  0x600003ec in deep_call (depth=0x3) at main.c:24
#4  0x600003ec in deep_call (depth=0x4) at main.c:24
#5  0x600003ec in deep_call (depth=0x5) at main.c:24
#6  0x600003ec in deep_call (depth=0x6) at main.c:24
#7  0x600003ec in deep_call (depth=0x7) at main.c:24
#8  0x600003ec in deep_call (depth=0x8) at main.c:24
#9  0x600003ec in deep_call (depth=0x9) at main.c:24
#10 0x600003ec in deep_call (depth=0xa) at main.c:24
#11 0x60000415 in task1 (arg=0xa) at main.c:41
(xt-gdb) print failed
$1 = 0x0
```

There are two remaining cases of interest: What does the `deep_call` trace look like under `task2`, and what is the failed count after several context switches? Efficient examination of the first case is done in the following way:

```
(xt-gdb) disable 3
(xt-gdb) disable 1
(xt-gdb) c
Continuing.

Breakpoint 2, task2 (arg=0x14) at main.c:48
48      deep_call( (int) arg );
(xt-gdb) enable 3
(xt-gdb) c
Continuing.

Breakpoint 3, deep_call (depth=0x0) at main.c:13
13      int a = depth;
(xt-gdb) where
#0  deep_call (depth=0x0) at main.c:13
#1  0x600003ec in deep_call (depth=0x1) at main.c:24
#2  0x600003ec in deep_call (depth=0x2) at main.c:24
#3  0x600003ec in deep_call (depth=0x3) at main.c:24
#4  0x600003ec in deep_call (depth=0x4) at main.c:24
#5  0x600003ec in deep_call (depth=0x5) at main.c:24
```

```
#6  0x600003ec in deep_call (depth=0x6) at main.c:24
#7  0x600003ec in deep_call (depth=0x7) at main.c:24
#8  0x600003ec in deep_call (depth=0x8) at main.c:24
#9  0x600003ec in deep_call (depth=0x9) at main.c:24
#10 0x600003ec in deep_call (depth=0xa) at main.c:24
#11 0x600003ec in deep_call (depth=0xb) at main.c:24
#12 0x600003ec in deep_call (depth=0xc) at main.c:24
#13 0x600003ec in deep_call (depth=0xd) at main.c:24
#14 0x600003ec in deep_call (depth=0xe) at main.c:24
#15 0x600003ec in deep_call (depth=0xf) at main.c:24
#16 0x600003ec in deep_call (depth=0x10) at main.c:24
#17 0x600003ec in deep_call (depth=0x11) at main.c:24
#18 0x600003ec in deep_call (depth=0x12) at main.c:24
#19 0x600003ec in deep_call (depth=0x13) at main.c:24
#20 0x600003ec in deep_call (depth=0x14) at main.c:24
#21 0x60000421 in task2 (arg=0x14) at main.c:48
```

This is what we expected.

Examination of the second case, an error in the context switching, requires examining the stack after the tasks have switched several times and the test code has been run. We can break after several context switches by manipulating the breakpoints.

```
(xt-gdb) disable 2
(xt-gdb) enable 1
(xt-gdb) c
Continuing.

Breakpoint 1, task1 (arg=0xa) at main.c:41
41      deep_call( (int) arg );
(xt-gdb) print failed
$4 = 0x0
```

Variable `failed` remains zero, suggesting that the context switch is performing as expected.

8. Supporting Configurable Processors

Xtensa processors are both configurable and extensible. Configuring a processor means adding predefined features to a base processor core. Developers can select various processor options when configuring a processor in Xtensa Xplorer. Configuration options, such as coprocessors and multiply-accumulate functions, add processor state.

Extending a processor means adding developer-defined features to a processor core. Developers use the Tensilica Instruction Extension (TIE) language to define new processor functions and resources. Developers may very well define new processor state of various kinds.

Custom State is all new processor state added to the processor by configuration or extension.

Depending on software design, custom state may introduce additional details for the runtime or operating system. Custom state may be part of the context of user tasks, and the Xtensa Coprocessor Option affects context switching:

- Custom state associated with a coprocessor need not be saved and restored on every context switch, but can be switched only when necessary. This is called Lazy Context Switching (see Section 8.2). We occasionally refer to this type of custom state as coprocessor state.
- Custom state that is not associated with a coprocessor needs to be saved and restored on every context switch. We occasionally refer to this type of custom state as noncoprocessor state.
- Both kinds of custom state may coexist in a processor.
- The TIE language considers register files and user state to be distinct and separate. However, from the OS point of view, they are both custom state, and they can be either coprocessor state or noncoprocessor state.

Developers must make two policy decisions regarding the use of custom state in interrupt and exception handling. Will the system use custom state in interrupt handlers? In exception handlers? The following examples assume that custom state will not be used in interrupt or exception handlers, or that if it is used, the specific handlers expressly manage the custom state as a special case. In fact, this is the policy adopted for all OS ports so far to the Xtensa architecture. This management can occur in one of two ways:

1. The developer guarantees that custom state is used only in interrupt handlers, so it does not need to be saved and restored.
2. The developer expressly saves and restores the custom state surrounding its use.

8.1 HAL Support

The Xtensa HAL automatically provides the necessary support for the operating system functions that must be performed during a context switch. In particular, the HAL provides a series of state manipulation routines that simplify the task of saving, restoring, and determining the size of custom state. These routines work for any configurations and extensions of Xtensa processors. The *Xtensa System Software Reference Manual* contains complete documentation for these functions.

8.2 Lazy Context Switching

We will first discuss lazy context switching before discussing the impacts of custom state on an operating system. This discussion applies only to custom state associated with a coprocessor and implies the processor is configured with the Coprocessor Option.

Access to custom state associated with a coprocessor can be enabled or disabled with the `CPENABLE` special register. When disabled, references to the coprocessor's state cause a processor exception. This is a powerful tool for the operating system.

The following example illustrates how an operating system uses `CPENABLE` to implement lazy context switching. Assume that:

- There are two tasks, task A and task B.
- There are two coprocessors, `cp_0` and `cp_1`.
- The operating system keeps track of the "owner" of each coprocessor in global state.
- The custom state is a custom register file associated with a coprocessor.

The state of the system consists of the valid bits in `CPENABLE`, and the operating system assigns an owner to each coprocessor. Consider Table 9, which contains an example sequence of events.

Table 9. Lazy Context Switching Example Sequence

O Valid	1 Valid	O Owner	1 Owner	Event	Comment
x	x	x	x		Core came up in unknown state.
				initialization	
1	1	none	none		At first, the system enabled both coprocessors. It then called a sequence of two HAL functions to set state for each register file to ensure that the coprocessors are in a known state.

Table 9. Lazy Context Switching Example Sequence (continued)

O Valid	1 Valid	O Owner	1 Owner	Event	Comment
0	0	none	none		At the end of initialization, the system disabled all coprocessors and assigned no owners.
				Task A is created	
0	0	none	none		During the creation of Task A, the OS makes sure to use the init_mem calls to initialize Task A's coprocessor save area to "safe" values for initial restoration of the coprocessor state. Coprocessor state is not affected, only the save area in memory is affected.
				Task B is created	
0	0	none	none		Task B's creation is just like Task A's creation.
				Task A uses cp_0	
1	0	A	None		The use of the register file caused an exception. The OS exception handler set the enable bit. Because there was not a previous owner of cp_0, the OS does not save any data. Because Task A accessed the coprocessor, the OS loads Task A's data for this coprocessor into cp_0 with the xthal_restore_cpregs(0) call. The OS sets the owner of cp_0 to A.
				Task B swaps in	
0	0	A	none		At the swap, the OS left A's state in the cp_0 register file. The register file was disabled but A was left as the owner.
				Task B uses cp_1	
0	1	A	B		As before, when A first used cp_0, the OS exception handler set the enable bit. The OS noted that cp_1 had not previously been used and therefore did not save anything. The OS loaded B's state into cp_1 and set the owner of cp_1 to B.
				Task A swaps in	

Table 9. Lazy Context Switching Example Sequence (continued)

O Valid	1 Valid	O Owner	1 Owner	Event	Comment
1	0	A	B		On this swap, the OS cleared the enable bit for cp_1 because B was swapping out and was the owner of cp_1. Seeing that A was swapping in, it set the enable bit for cp_0. Task A can use cp_0 without causing exception. Note that this is merely one implementation. All enable bits could be turned off, and if A touches the coprocessor, the OS could, in the exception, recognize that A's state is already loaded into cp_0 and avoid the restore at that point. The exception would set the enable bit.
				Task A uses cp_0	
1	0	A	B		Because A's state was already in cp_0, the OS had already set the enable bit on the context switch. Since the enable bit is set, no exception occurs and the OS does nothing here.
				Task A uses cp_1	
1	1	A	A		Task A's use of cp_1 caused an exception. The OS exception handler set the enable bit for cp_1. The handler, seeing that Task B owned cp_1, saved the contents of cp_1 to Task B's coprocessor save area with the xthal_save_cpreg(1) call. It then restored Task A's state to cp_1 with xthal_restore_cpreg(1), and Task A is made its owner.
				Task B swaps in	
0	0	A	A		All of the enable bits owned by Task A were turned off. Task B owned no coprocessors so no enable bits were turned on.

Table 9. Lazy Context Switching Example Sequence (continued)

O Valid	1 Valid	O Owner	1 Owner	Event	Comment
				Task B uses cp_1	
0	1	A	B		Task B's use of cp_1 caused an exception. This exception turned on the enable bit for cp_1. The OS saw that Task A currently owned cp_1 and used the xthal_save_cpreg(1) call to save the current state to Task A's coprocessor save area. The OS then used xthal_restore_cpreg(1) to restore Task B's state to cp_1 and made Task B the owner once again.
				The system continues....	

8.3 Custom State Save Area

Each user task must have space allocated to save and restore its custom state. There are two types of custom state: coprocessor state and noncoprocessor state. Each type imposes different requirements, so we differentiate between *coprocessor save areas* and *noncoprocessor save areas*. Collectively, we call these save areas *custom state save area*.

An OS can allocate these custom state save areas in various ways:

- Place them in the task control block either directly or via pointers.
- Extend the size of each user's stack to make space for these save areas. Save custom state to either the exception stack frames or to specifically reserved areas of the stack.
- If exception handlers are able to allocate memory dynamically, an OS might even allocate the coprocessor save areas dynamically for each task, piggy-backing on the lazy switch mechanism. Dynamic allocation would occur when a task first uses a given coprocessor. Of course, system designers must carefully consider the case of running out of memory.

Our examples in this chapter use the second method, extending the stack size.

Note: it is important to align state save areas according to the alignment requirements reported by the HAL. This alignment can be greater than the stack's minimum 16-byte alignment requirement.

Certain precautions must be taken when allocating the custom state save area on the stack. To fully understand the requirements, consider Figure 8, which shows an erroneous stack layout for a swapped-out user task. The core register state is allocated first, followed by the noncoprocessor save area, which is followed by the coprocessor save area.

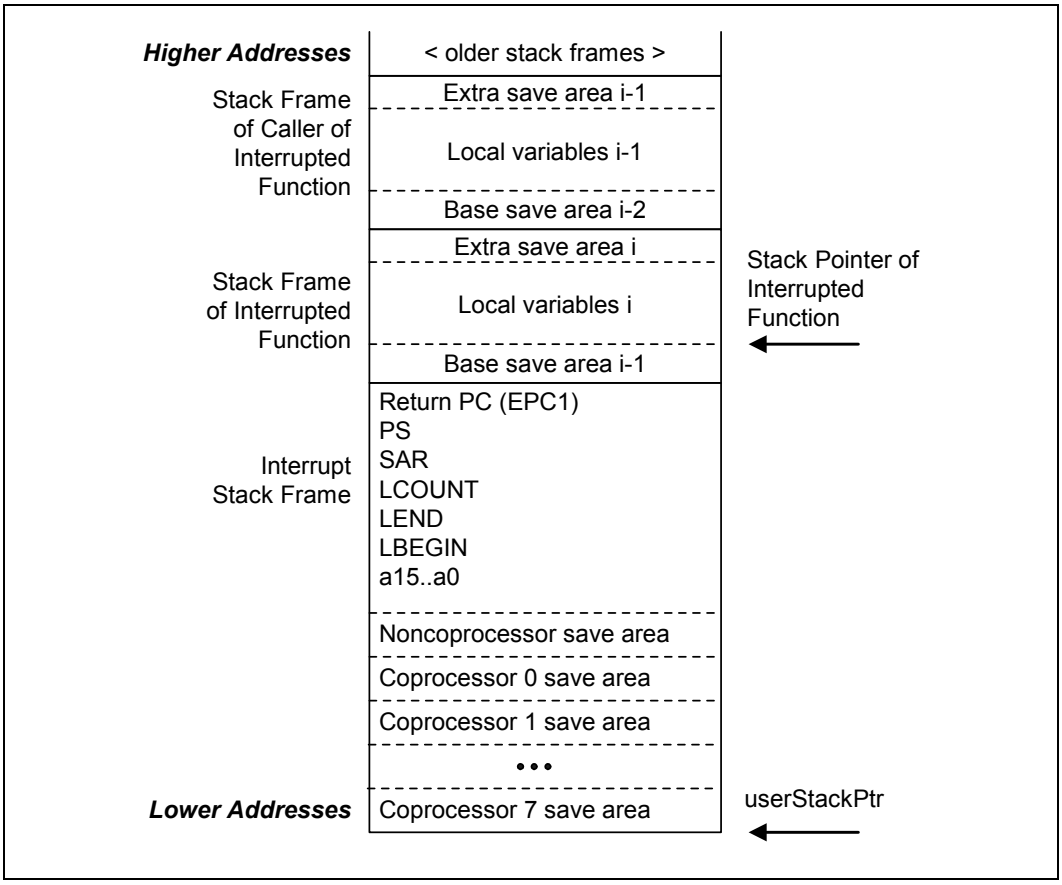

Figure 8. BAD User Stack Layout for Custom State

Consider what happens on a context switch. The core processor state is restored to the processor, so this portion of the stack becomes available. The noncoprocessor state is restored to the processor, so this memory is also available. But the coprocessor state is restored lazily and thus needs to remain on the stack.

The stack layout in Figure 8 will result in coprocessor state corruption when switching to a user task that immediately starts going deeper into the stack, and does not first reference all coprocessor state that it has previously touched. Because coprocessor state is switched lazily, the coprocessor save area should not exist in the low addresses of the stack frame where it will be overwritten when the user task again begins execution.

Figure 9 shows a correct stack layout for custom state save areas.

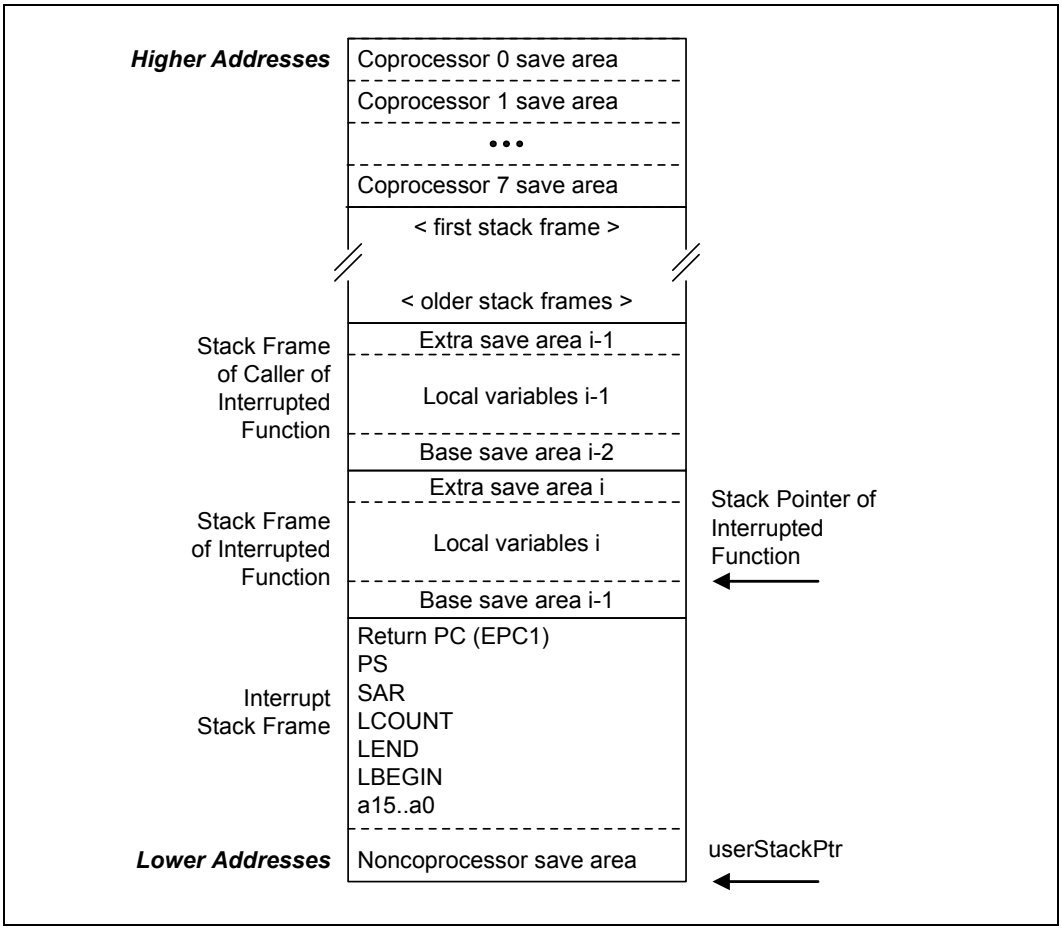

Figure 9. User Stack Layout with Custom State

The interrupt stack frame is appropriate only for processor state that is saved and restored on every context switch. Noncoprocessor state is analogous to the core processor state, such as the register file. It must be saved and restored on every context switch. We can allocate the noncoprocessor save area in the interrupt stack frame.

Coprocessor state is not saved and restored on every context switch. Instead, we use lazy context switching. Figure 9 shows a separate coprocessor save area at the logical base of the user stack which is not part of the interrupt stack frame. This decoupled save area allows saved coprocessor state to persist across any number of exceptions without interference from the normal context switching mechanisms.

Each user task that uses coprocessor state needs a coprocessor save area. For simplicity, an OS might allocate coprocessor save areas for each user task.

8.4 Save and Restore Mechanisms

The Xtensa HAL contains helper routines that operating systems can invoke to save and restore all custom state. (The CHAL also provides assembly-macro versions that avoid function-call overhead.) Because there are two types of custom state, there are two sets of routines.

The first set of routines, `xthal_save_extra` and `xthal_restore_extra`, save and restore noncoprocessor state. The second set of routines, `xthal_save_cpregs` and `xthal_restore_cpregs`, save and restore coprocessor state. The latter routines each take an additional argument that indicates which coprocessor's state is affected.

The HAL also provides non-windowed ABI versions of these routines, allowing inline invocation from context-switching logic or other places where using windowed registers is inconvenient or not allowed.

Of course, all of these routines must be invoked at the right point in the execution of the operating system. The `xthal_save_extra` and `xthal_restore_extra` routines are typically invoked during the normal sequence of context save and restore. In the previous context switching example, the extra state should be saved and restored immediately after saving live window frames to the stack, that is, immediately after the following statements:

```
l32i    a1, a3, 4
call0   xthal_window_spill_nw
```

Invocation of the save and restore mechanisms to save and restore coprocessor state is a little more complex. The `CPENABLE` special register has a bit per coprocessor. When a coprocessor's bit in `CPENABLE` is set to 0 and the coprocessor is referenced, the processor causes an exception. The exception cause is set according to the coprocessor that was referenced.

Recall that the previous context save and restore occurs on interrupt exceptions and is detected with the following code:

```
rsr     a3, EXCCAUSE
beqi    a3, EXCCAUSE_LEVEL1INTERRUPT, handleInt
break   1, 1
```

This code must be changed to detect and dispatch the appropriate handlers for coprocessor exceptions. The EXCCAUSE register contains a value from 32 (for coprocessor 0) through 39 (for coprocessor 7) to reflect the coprocessor that has been referenced. The operating system will then switch the coprocessor context when it has identified the coprocessor that has caused the exception.

This lazy context switching can be done in C or in assembly code. The following example code assumes that the operating system has gone through the effort to set up a C execution environment for exceptions. The following code handles the switching of coprocessor context. It also assumes that interrupts are disabled throughout.

```
TCB *xt_coproc_owner[NUM_CPS];
extern TCB *current_task;
void excSaveCoProc( int excCause )
{
    unsigned key;
    int coproc = excCause - 32;
    TCB *CpOwner;

    CpOwner = xt_coproc_owner[coproc];

    xthal_validate_cp(coproc);

    if (CpOwner != current_task)
    {
        if (CpOwner != NULL)
            xthal_save_cpregs(CP_SAVE_AREA(CpOwner, coproc), coproc);

        xthal_restore_cpregs(CP_SAVE_AREA(current_task, coproc),
                             coproc);
        xt_coproc_owner[coproc] = current_task;
    }
}
```

First, we declare space to maintain coprocessor ownership.

```
TCB *xt_coproc_owner[NUM_CPS];
```

This example assumes the use of an actual task control block (TCB) structure. While our example runtimes so far have managed user tasks by tracking only a stack pointer for each task, systems dealing with coprocessors will typically track more information for each user task in a TCB.

When a user task “owns” a coprocessor, the coprocessor has state because it is a part of that task’s execution state. Ownership specifies which task’s data is currently live in the coprocessor. The xt_coproc_owner array tracks the current owner of each coprocessor. When coprocessor 0 is owned by a particular task, that task’s TCB pointer will

be in xt_coproc_owner[0]. When no task owns it (for example, at startup, or when the last task that owned it was destroyed), we assume that here it will be a NULL pointer.

```
void excSaveCoProc( int excCause )
{
    unsigned key;
    int coproc = excCause - 32;
    TCB *CpOwner;
```

The only argument is the EXCCAUSE that corresponds to the exception that occurred. This routine assumes it is invoked only for CPENABLE coprocessor exceptions. The coproc local variable is given the value of the coprocessor that has been accessed.

Next, we validate the coprocessor.

```
    xthal_validate_cp(coproc);
```

This HAL routine sets CPENABLE such that the coprocessor's registers can be accessed without causing an exception. In this model, the operating system invalidates all coprocessors for access on every switch—even if the task being activated already owns the coprocessors. There are now three cases to consider.

In the first case, the current task already owns the coprocessor. In this case, all of the work is done except for restoring interrupts. The coprocessor has just been marked as safe for use and the data is valid, so the "switch" is over.

In the second case, the task being activated does not currently own the coprocessor, and another owner currently exists.

```
    CpOwner = xt_coproc_owner[coproc];
    if (CpOwner != current_task) {
        if (CpOwner != NULL)
            xthal_save_cpregs(CP_SAVE_AREA(CpOwner, coproc), coproc);
```

Note how the control logic first tests for the first case and skips all further work if the owner and current task are the same. The control logic next tests if the coprocessor currently has a valid owner. If so, then we are in the second case, and we save the current state.

The CP_SAVE_AREA macro converts a TCB pointer and coprocessor number into a memory pointer. This pointer addresses the coprocessor save area within the TCB for the specific coprocessor.

In the third case, the coprocessor has no owner. The differences between case two and case three have already been covered by the control logic, and case two and three are handled identically from this point onward in the code.

```
xthal_restore_cpregs(CP_SAVE_AREA(current_task, coproc), coproc);
xt_coproc_owner[coproc] = current_task;
```

The operations are simple. The coprocessor's state is restored from the current task, and the current task is marked as owning the coprocessor.

9. Memory Management and Protection

Xtensa's Memory Management Unit (MMU) Option provides virtual-to-physical address translation and protection. The translation and protection architecture itself is configurable. This section discusses the features, operations, and programmer implications of the memory management options in the Xtensa architecture, which are listed below.

- With XEA2:
  - MMU with Translation Look Aside Buffer (TLB) and Autorefill
  - Region Protection with Translation
  - Region Protection
- With XEA1:
  - Region Protection (CACHEATTR)

Not all Xtensa processor versions support all memory protection options. Refer to the relevant *Xtensa Microprocessor Data Book* for details.

9.1 Simple Configurations

The last three configurations listed above are relatively simple. The *Xtensa Microprocessor Data Book* describes these configurations adequately for programmers, so we mention here only a simple example of remapping regions in configurations with Region Protection with Translation. See Appendix B for the complete example.

```
            movi    a5, 0xE0000000  // tlb mask, upper 3 bits
            movi    a6, 0f          // PC where ITLB is set
            and     a4, a3, a5      // upper 3 bits of PPN area
            and     a7, a2, a5      // upper 3 bits of VPN area
            and     a6, a6, a5      // upper 3 bits of local PC area
            beq     a7, a6, 1f      // branch if current PC's region

           // Note that in the WITLB section, we don't do any load/stores.
           // It's important in the DTLB case.

            ritlb1  a5, a7          // get current PPN+AM of segment for I
            rdtlb1  a6, a7          // get current PPN+AM of segment for D
            extui   a5, a5, 0, 4    // keep only AM for I
            extui   a6, a6, 0, 4    // keep only AM for D
            add     a2, a4, a5      // combine new PPN with orig AM for I
            add     a3, a4, a6      // combine new PPN with orig AM for D
0:          witlb   a2, a7          // write new tlb mapping for I
            wdtlb   a3, a7          // write new tlb mapping for D
1:          isync
```

On entry, a2 should contain an address within the VPN region, and a3 should contain an address within the PPN region. We begin with a check to verify that it will not remap itself out of existence. In other words, for this example we do not want to remap the region from which the processor is currently fetching instructions. So we branch to avoid the situation altogether.

This code changes only the virtual-to-physical mapping for both the ITLB and the DTLB. The DTLB mapping must follow ITLB mappings so literal values are accessible. To preserve the Access Mode (AM) setting for the region, we first read the current AM value, combine it with the new PPN setting, then write the mapping (for both TLBs). The new mapping takes effect after the `isync` instruction.

9.2 MMU with Translation Lookaside Buffer and Autorefill

This MMU configuration provides all the facilities of a general-purpose MMU, such as demand paging and memory protection, necessary to support operating systems with virtual memory capabilities, such as Linux or WindowsCE.

9.2.1 MMU Versions

Thus far, there have been three versions of the Xtensa MMU. Tensilica T1040 and T1050 processors include version 1. Tensilica RA and RB release processors (LX, LX2, Xtensa 6, Xtensa 7, and Diamond Standard 232L rev.A and rev.B) include version 2, which is identical to version 1 except for access mode (AM field) encodings, enhanced to support non-executable pages. Tensilica RC and later release processors (LX3 thru LX5, Xtensa 8 thru 10, and Diamond Standard 233L Rev.C and later) include MMU version 3.

Portable software using the Xtensa HAL can distinguish MMU versions at compile-time using HAL macros (for example, using `#include <xtensa/config/core.h>`). The presence of an MMU is indicated by `XCHAL_HAVE_PTP_MMU` set to 1 rather than 0. When that is set, MMU version 3 is distinguished by `XCHAL_HAVE_SPANNING_WAY` set to 1, and MMU version 1 is distinguished by `XCHAL_HW_VERSION_MAJOR < 2000`. For more information on the HAL, see the HAL chapter of the *Xtensa System Software Reference Manual*.

9.2.2 TLB Ways

The Xtensa MMU translation hardware is generically called a Translation Lookaside Buffer (TLB). The Xtensa MMU has two independent TLBs, one for instruction memory (ITLB) and one for data memory (DTLB). A TLB is a collection of several *ways* of various types. Figure 10 shows one way with four *entries* that support 4-KB page sizes. Each entry has the following fields:

- The Virtual Page Number (VPN) is a 20-bit field that comes from the high-order bits of the virtual address being translated. It uniquely identifies a 4 KB page in virtual address space. The lower 2 bits of VPN are implicit for each entry, as they index (select) one of the 4 entries. Only the upper 18 bits of VPN are stored in this TLB way.
- The Physical Page Number (PPN) field is a 20-bit field that contains the high-order bits of the physical address to which the VPN is mapped. The PPN uniquely identifies a 4 KB page in physical address space.
- Access Mode (AM) is a 4-bit field that indicates an access mode for the page.
- The Address Space Identifier (ASID) is an 8-bit field that holds the ID of the owning process.
- Shaded portions are unused and not available for software use.

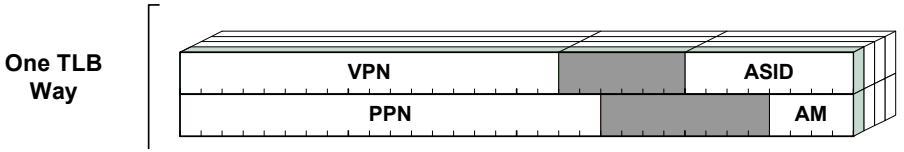

Figure 10. Example TLB Way

9.2.3 TLB Hit, Miss, and MultiHit

Any memory access must match no more than one TLB entry. A TLB miss condition is where no TLB entries match. The case where multiple TLB entries match the address is a multihit condition that results in a MultiHit exception. This condition is not fatal on existing Xtensa processor implementations, but may be fatal in future implementations. Software must avoid this multihit condition.

9.2.4 Address Space Identifier (ASID)

The virtual address input to the TLBs is actually the concatenation of an ASID specified in a processor register with the 32-bit virtual address from the fetch, load, or store address calculation. ASIDs allow software to change the address space that the processor sees (for example, on a context switch) with a simple register write without having to change (that is, invalidate) the entire TLB contents. The TLB stores an ASID with each entry, so it can simultaneously hold translations for multiple address spaces.

The Xtensa MMU reserves two ASID values. The ASID value 0 is reserved to indicate an invalid TLB entry. The ASID value 1 is reserved to indicate kernel or privileged mode. Software may use all other ASID values to identify virtual address spaces (for example, for user tasks).

9.2.5 Protection and Privilege

Software executing with CRING==0 (that is, PS.RING==0 or PS.EXCM==1) is able to execute all Xtensa instructions. Other rings may execute only non-privileged instructions. Since writing to the PS register is a privileged instruction, only software running at CRING==0 (usually an operating system) can effect a ring-to-ring transition.

The 32-bit Ring ASID (RASID) special register contains four 8-bit ASID values, as shown in Figure 11. The RASID fields map an ASID to a ring level, ASID0 corresponding to ring 0, ASID1 corresponding to ring 1, and so forth.

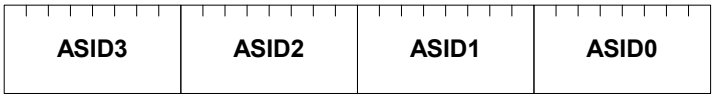

Figure 11. RASID Special Register Format

Each ring has its own ASID. The ASID0 field of the RASID is hardwired to 1, or the kernel ASID. The ASIDs for each ring in RASID must be unique. Each ASID has a single ring level, although software may maintain many ASIDs at the same ring level (except ring 0) to allow sharing. The ring number of a page is not stored in the TLB; only the ASID is stored.

When a TLB is searched for a virtual address match, the ASIDs of all rings specified in RASID are tried. The position of the matching ASID in RASID gives the ring number of the page. If the page's ring number is less than the processor's current ring number (CRING), the access is denied with a Load, Store, or Fetch Prohibited exception.

Software must ensure that all ring fields in the RASID register are unique and valid. Otherwise, a multihit condition is possible. Only one ASID in the RASID is allowed to match a given access.

The ASID0 field in RASID is not writable and is hardwired to the reserved kernel ASID 1. On reset, the other ring fields in RASID are guaranteed to come up with unique ASID values, and they are guaranteed not to equal the invalid ASID value. Software must never write the invalid ASID value to any ring field in RASID. Otherwise, access can match invalid entries, leading to undefined behavior and possible multihit conditions.

9.2.6 ASID Management

ASID management is a software responsibility. In a software environment that consists of only one thread of execution, or with multiple threads sharing a single address space, the RASID register can simply keep its reset value, which consists of valid and unique ASID values. In the presence of multiple *processes* (where each process is assumed able to have its own address space), such as in a protected OS like Unix, the OS con-

text-switches the RASID register along with other process state. It is usually sufficient to context-switch just one of the ASID fields, leaving the others constant; in other words, assigning all processes to the same ring. The additional ASID fields (or rings) allow partitioning a process' accessible memory in up to four address spaces, each used by a different set of processes. When switching between two processes that share an address space (generally at the same ring level), such as shared memory or libraries, the corresponding ASID field need not be context-switched; any TLB entry with that ASID value can be re-used, thus reducing TLB miss overhead.

It is thus up to software (such as an OS kernel) to select which rings (ASID fields) to use, and which possible ASID values within each to use.

With at most 252 processes in a system, each can be assigned a unique ASID value (at one of 3 rings, 1 through 3), leaving value 0 for the invalid ASID, value 1 for kernel ASID at ring 0, and two more values for the remaining two rings (assuming each is unused or used with a single address space).

However, many OS kernels allow for arbitrary numbers of processes. So they assign ASIDs (ASID values) dynamically. One simple method is to allocate an ASID each time a new address space is used. When no more ASIDs are available for a new address space, the kernel flushes the ITLB and DTLB to get rid of any existing uses of assigned ASIDs, and begins assigning ASIDs anew. With 8-bit ASID values, a TLB flush needs to occur at most every 252 context switches, if every context switch is to one new address space. A TLB flush marks the start of a new ASID assignment cycle. To avoid having to mark, at each flush, the 252 affected processes as not having an ASID anymore, the kernel can associate a generation count (or ASID assignment cycle count, incremented every cycle) with each process' assigned ASID in the process control block. When a process tries to use an ASID with an outdated generation count, the kernel knows to assign a new ASID.

9.2.7 Access Modes

The Access Mode field of a TLB entry controls how the processor accesses the page in memory. The Access Mode field itself has subfields:

- bit 0 is a page-executable bit[26].
- bit 1 is a page-writable (dirty) bit; and,
- bits 2 and 3 indicate a cache mode (if both are set, other bits have special meaning)

The relevant *Xtensa Microprocessor Data Book* thoroughly describes all access modes.

26. In MMU versions 2 and 3 only; in MMU version 1, bit 0 is a page-valid bit.

9.2.8 Successful Translation

Virtual to physical address translation consists of searching the TLB for an entry that matches both the virtual address' VPN and one of the ASIDs in the RASID register, and producing both a physical address (using that entry's PPN bits, and the offset bits pass through unchanged), and access-mode bits for the page (using that entry's AM bits).

Recall that there are three possible outcomes to an access: miss, hit, or multihit. Here, we examine the successful translation case. Figure 12 shows one way of a 16-entry, 4-way set-associative TLB to illustrate a successful virtual to physical address translation. The 32-bit virtual address can be broken down as a 20-bit VPN field, with 2 lower bits indexing each way's entries, and a 12-bit offset field (4 KB page size). The VPNs from the entries indexed by the access virtual address are compared against the VPN in the virtual address. If they match, hardware selects the PPN and AM from that entry.

If more than one TLB way matches the virtual address, the processor takes a MultiHit exception. Software must avoid overlapping TLB mappings.

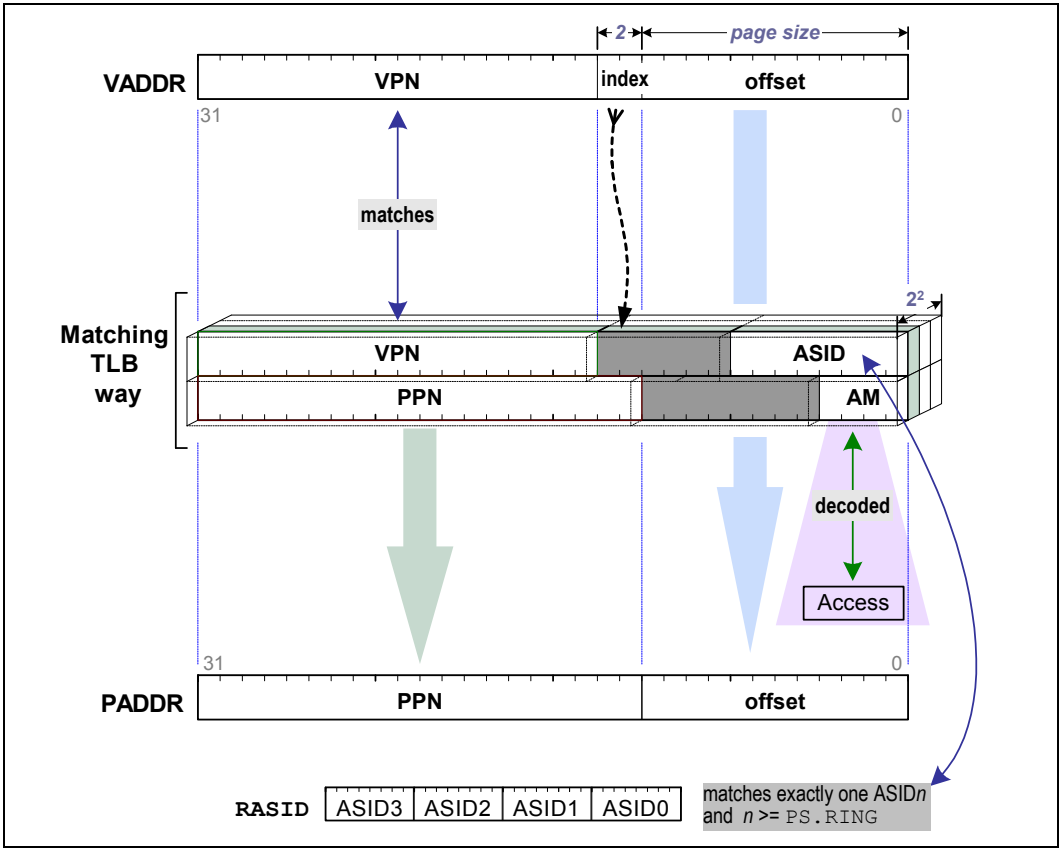

Figure 12. Example MMU Successful Translation

9.2.9 Wired Ways

TLB ways 4 and up of the Xtensa MMU are configured as *wired* ways (sometimes referred to as locked); that is, hardware does not modify them on autorefill operations. It is up to software to write directly into wired ways to specify or change virtual to physical translations.

On reset, entries in all TLB ways other than wired way 6 are marked as invalid (zero ASID field), so that none of them match any memory access.[27]

Wired TLB way 6 is valid on reset, mapping the entire 4 GB virtual space to physical space without translation (similarly to processors configured with region protection).

Wired ways are particularly useful for critical software that cannot afford the overhead or effects of a TLB miss. Operating systems often use wired ways to map page tables, or portions of a page table, as well as exception and interrupt handlers and related data (see Section 9.2.10).

Ways 4 through 6 of both TLBs offer support for large pages, configurable at run time. Ways 7, 8, and 9 are only present in the DTLB, each for a single 4 kB page. See the *Xtensa Microprocessor Data Book* for further descriptions.

9.2.10 About Miss Exceptions and Interrupt and Exception Handlers

The handler for a user, kernel or double exception of any kind (including level-one interrupts) must not itself cause a TLB miss (either ITLB or DTLB) until all critical exception state has been saved, otherwise that miss is unrecoverable. For example, if you get some exception (*e.g.*, `alloca`, `syscall`, etc) and the exception handler itself gets a TLB miss exception without having yet saved `EXCCAUSE` (and `EXCVADDR` if relevant), the first exception's `EXCCAUSE` (and `EXCVADDR`) are lost. Even a “first level” TLB miss handled in hardware (without raising an exception) is to be similarly avoided, as it can modify the `EXCVADDR` register.

So, there is always some critical section of an exception handler (on entry and exit, or possibly the entire handler) that must not be susceptible to TLB miss exceptions. The first thing one typically does in these critical sections is to save some state/registers to memory. Such memory must always be mapped via TLBs, not susceptible to miss exceptions at the time the exception or interrupt can happen. Thus, exception and interrupt handler code sequences, as well as any data area they access (including literals), are normally mapped using wired (or static) TLB ways.

27. MMU version 3 only. In MMU versions 1 and 2, TLB ways 5 and 6 are static; that is, they are always valid, and cannot be modified. Also, wired way 4 is valid on reset if any local memories are configured, in order to map them. Figure 19 on page 161 depicts the resulting memory map.

Note that high-priority interrupts (with level greater than EXCMLEVEL) must also avoid TLB miss exceptions in such critical sections. This is because high-priority interrupts can interrupt the above exception handlers at any point (they are not blocked by PS.EXCM being set), so they must save and restore all exception state around any code that might itself take an exception, such as a TLB miss exception. So the memory used to save data during high-priority interrupt handlers must also always be mapped in the TLB when they occur.

Note also that window overflow and underflow exceptions can safely take a TLB miss exception and be recoverable, because they use neither EXCCAUSE nor EXCVADDR, and always run with PS.EXCM set so any such exception is a double exception (setting DEPC and not the window handler's EPC1).

9.2.11 Autorefill Ways and Page Tables

Ways 0, 1, 2, and 3 of both the ITLB and the DTLB are autorefill ways. Software may write to autorefill ways, but usually lets the processor change the content of autorefill ways instead. If no entry matching an access address is found in a TLB, the processor attempts to load a Page Table Entry (PTE) from memory and write to one of the autorefill ways. This hardware-generated load itself is done in virtual space and requires virtual to physical address translation by the DTLB. If this second translation also misses, the processor takes a TLB miss exception, and software must complete the refill. These exceptions are also generically called *second-level miss* exceptions.

Note that if the hardware-generated load fails for any reason, the processor takes a TLB miss exception. A miss is the most common cause of a TLB miss exception, but is not the only possible cause. Bus errors and privilege violations are two other examples.

If this second translation succeeds, the PTE load proceeds. Figure 13 illustrates the PTE format and how hardware updates a TLB entry. The MMU selects the autorefill way based on a Decay Replacement[28] algorithm. Hardware gathers information from several places and writes it into the selected TLB entry:

- The PTE's RI field (the RING field) is used as an index into the RASID register. The resulting ASID is written into the TLB.
- The PTE's Access Mode and PPN fields are copied directly into the TLB.
- Register EXCVADDR holds the virtual address that caused the exception. The top 18 bits of EXCVADDR form the upper part of the VPN value written into the TLB.
- Bits 12 and 13 form an index into the TLB ways to select an entry from each of the 4 ways. The Decay Replacement algorithm then selects the way to be modified with PTE information. (If a way is configured with 8 entries, bits 12, 13, and 14 form the index.)

28. The Decay Replacement algorithm is a hybrid of a least-recently-used algorithm and a not-most-recently-used algorithm.

Hardware ignores the shaded portion of the PTEs. Software may use those bits to hold page-state information, such as whether the page has been modified, accessed, or is writeable, readable, or present (in the logical address space, as opposed to the current TLB state).

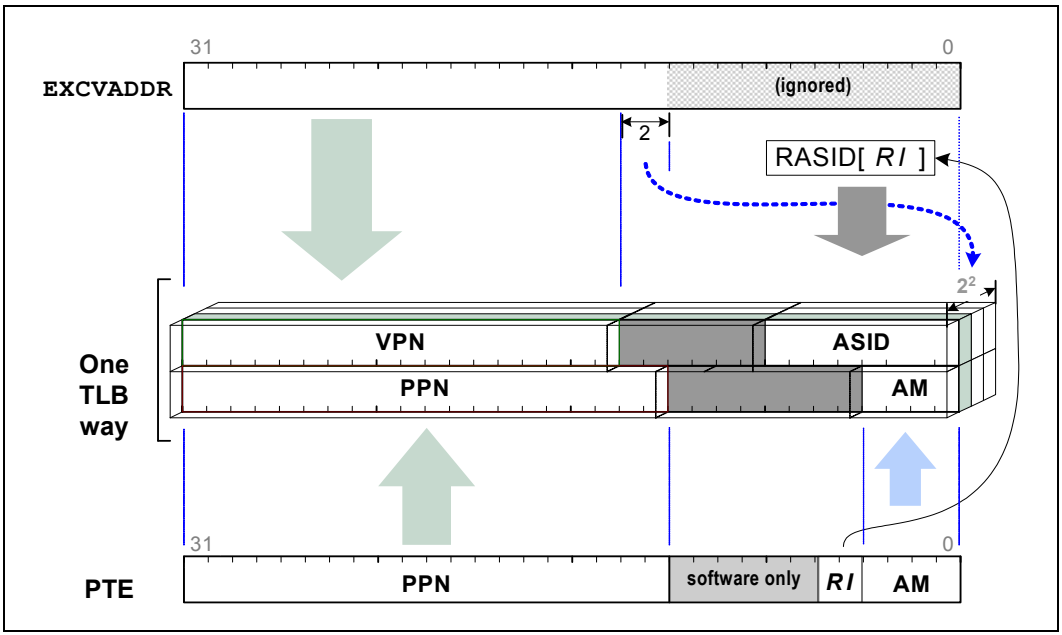

Figure 13. Example Autorefill Operation

Xtensa processors' TLB refill mechanism requires the page table for the current address space to reside in the current virtual address space. Figure 14 illustrates how the MMU hardware forms a virtual address of the PTE to load. The base of the page table is given by the `PTBASE` field of the `PTEVADDR` register. Software sets `PTBASE` when initializing the MMU. On a TLB miss, the processor forms the virtual address of the PTE by catenating the page table base (`PTEVADDR.PTBASE`), the VPN bits of the miss virtual address (`EXCVADDR`), and two zero bits (PTEs are four bytes large).

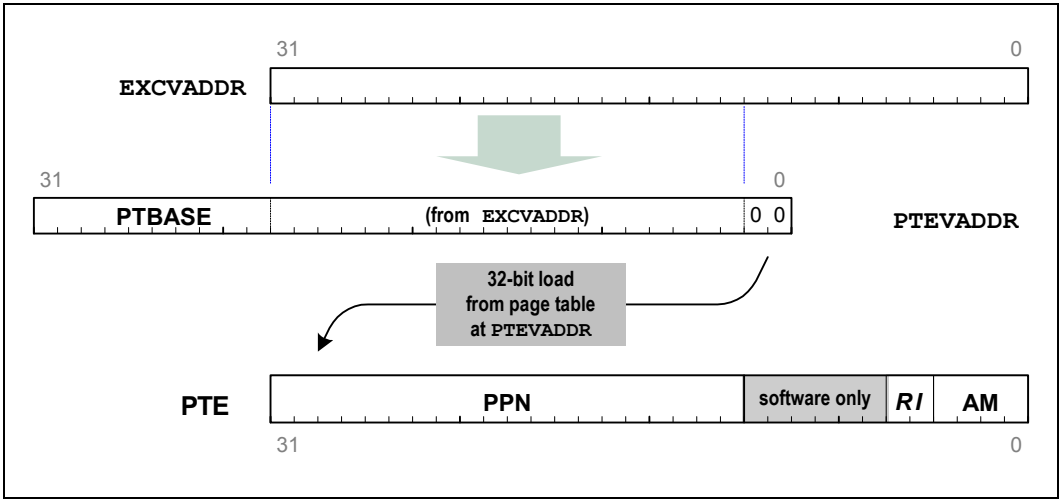

Figure 14. Example PTE-Load Computation

9.2.12 How Large is the Page Table?

Figure 19 on page 165 shows a memory map (possible with all versions of the MMU) with 3.25 GB of dynamically mappable virtual space available to user tasks (addresses 0 through 0xCFFFFFFF are not mapped). The page table for dynamically mapped space is therefore 3.25 GB / 4 KB = 851968 PTEs taking 3.25 MB of virtual address space using 4 bytes per PTE.

Any other area could have been mapped instead (MMU version 3 only), thus the page table can potentially take an entire 4 MB of virtual space. However, as noted in Section 9.2.9, at least some memory must be mapped using wired ways (such as for interrupt and exception handlers), possibly reducing the required size of the page table.

The operating system must decide how to map the page table. At one extreme, it may choose to map the entire 4 MB to avoid the overhead of second-level misses. At the other extreme, it may choose to map only one 4 KB page of the page table at a time to preserve memory.

9.2.13 The Whole TLB

This section contrasts MMU versions 1 and 2 with MMU version 3.

Each TLB is comprised of several ways of different types. Figure 15 and Figure 16 illustrate, respectively, the complete DTLB for MMU versions 1 and 2, and for MMU version 3. The focus of these diagrams is primarily the various TLB-way types that together form the DTLB. The ITLB is identical, except it does not have wired ways 7, 8, and 9.

The *MMU with TLB and Autorefill* option is itself configurable. Autorefill ways 0 through 3 of both the ITLB and the DTLB can be configured to have either 4 or 8 entries per way. Figure 15 and Figure 16 show the largest configuration of 8 entries per autorefill way.

Way 4 of both the ITLB and DTLB supports multiple page sizes. The IPS4 and DPS4 fields of the `ITLBCFG` and `DTLBCFG` special registers, respectively, select the page size for way 4. The reset value for these registers is zero, selecting the 1 MB page size. There are 4 entries, so this can map a span of 4 MB if, for example, mappings are set to be contiguous. Other valid values include 1, 2, and 3, which select page sizes of 4 MB, 16 MB, and 64 MB, respectively. All other values are invalid and result in undefined processor behavior.

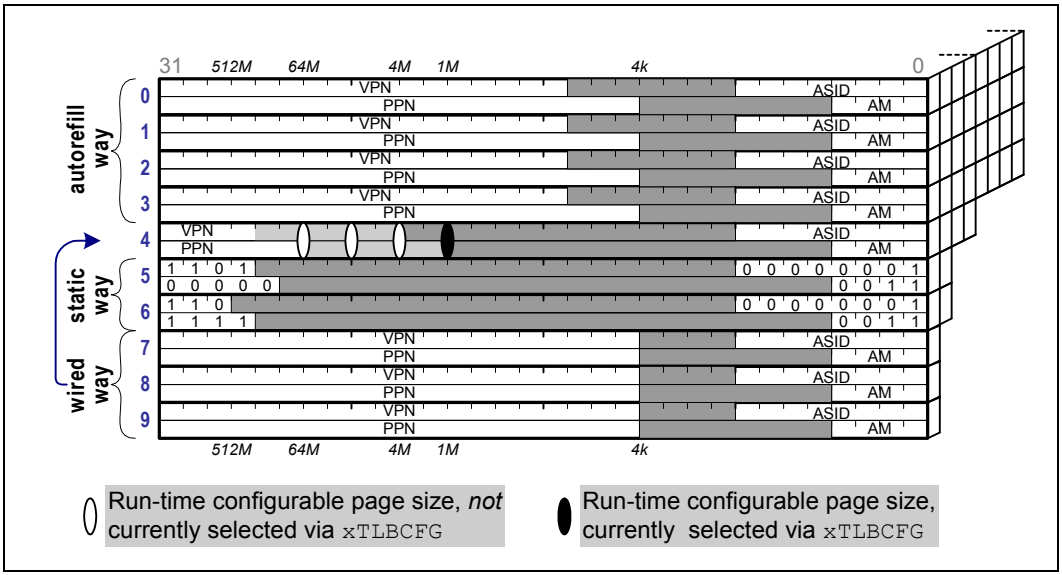

Figure 15. Example MMU Way Configuration for MMU versions 1 and 2

In MMU version 3, ways 5 and 6 are increased to 4 and 8 entries respectively, are wired (writable) instead of static, and support multiple page sizes. Way 5 supports page sizes of 128 MB and 256 MB, selected using values 0 and 1 respectively, of fields `IPS5` of `ITLBCFG` and `DPS5` of `DTLBCFG`. Similarly, way 6 supports page sizes of 512 MB and 256 MB, selected using values 0 and 1 respectively, of fields `IPS5` of `ITLBCFG` and `DPS5` of `DTLBCFG`. At reset, all way 6 entries are marked valid (ASID of 1), mapping the entire 4 GB space in eight 512 MB regions. The reset memory map in MMU version 3 thus maps virtual = physical, which is quite different than with MMU versions 1 and 2.

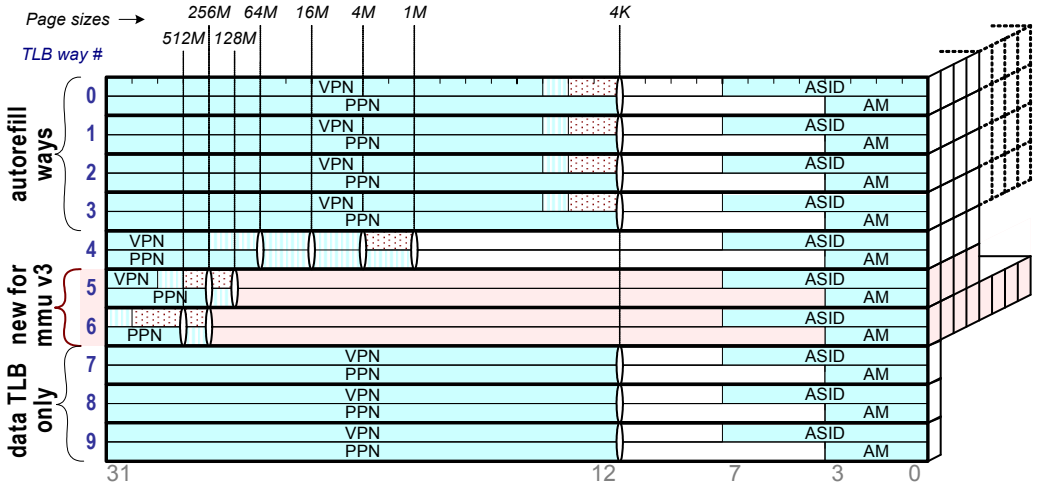

Figure 16. MMU Way Configuration for MMU version 3

Both `ITLBCFG` and `DTLBCFG` are 32-bit special registers with formats illustrated in Figure 17 and Figure 18, respectively. Note that values written to select a page size for way 4 must be written to the appropriate `IPS4` and `DPS4` subfield. In MMU version 3, page sizes for ways 5 and 6 are selected using `IPS5/DPS5` and `IPS6/DPS6` fields respectively. All other subfields are unused and reserved. The result of writing non-zero values to these reserved subfields is undefined.

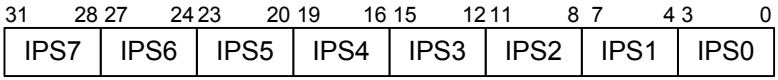

Figure 17. `ITLBCFG` Register Format

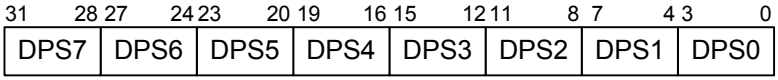

Figure 18. `DTLBCFG` Register Format

9.2.14 Memory Map

Figure 19 shows the virtual and physical address spaces of a version 1 or 2 MMU. It shows static mappings in the MMU configuration, area reserved for kernel space, and cached vs. uncached regions (information useful for mapping peripherals).

MMU version 3 provides much greater flexibility in mapping such large pages using wired ways. See the relevant *Xtensa Microprocessor Data Book* for more details.

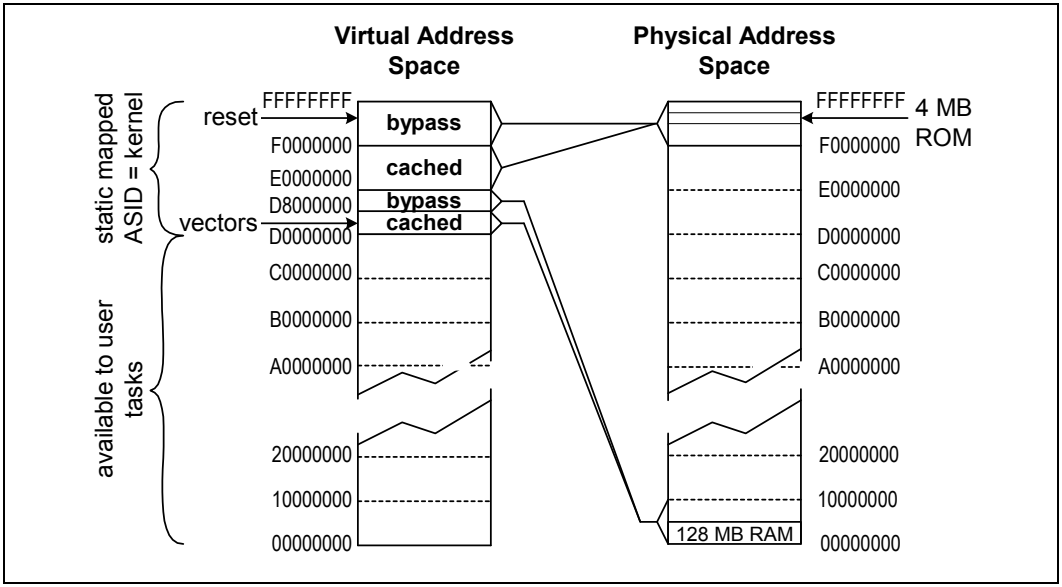

Figure 19. Example MMU Memory Map for MMU versions 1 and 2

9.3 Example TLB Miss Handler

This section describes a highly simplified TLB miss handler for illustrative purposes. More adventurous readers are invited to study the Xtensa architecture portions of the Linux operating system source code for a more in-depth look at how one can implement a virtual memory subsystem using the Xtensa MMU.

Before we begin, note that it is possible to avoid writing a miss handler altogether by simply allocating an aligned 4 MB of memory for the page table (minus any static or wired mappings; for MMU versions 1 and 2 for example, by allocating 3.25 MB aligned to a 4 MB boundary). To ensure the page table is always accessible without taking a second level miss, place it in a *statically* mapped virtual address range (that is, 0xD0000000 through 0xFFFFFFFF for MMU versions 1 and 2) or access it through a sufficiently large *wired* DTLB mapping (such as DTLB way 4; or with MMU version 3, also way 5 or 6). In this case, simply initialize the page table as desired, point `PTEVADDR` to its base address, and you now have paged access in whatever virtual address ranges are not mapped by wired or static TLB entries. When an access misses in the instruction or data TLB, the hardware autorefill mechanism is always able to load the corresponding PTE entry from the page table. The main downside of this approach is that the typical active portion of a page table is much smaller than 4 MB, so a full page table tends to waste memory unnecessarily.

The alternative is to place the page table in unmapped memory (for MMU versions 1 and 2, that's somewhere in virtual address range 0 through 0xCFFFFFFF; with MMU version 3 this could be anywhere). That way we only need to allocate the needed 4 kB pages out of the 4 MB page table range. Each 4 kB of the page table corresponds to 4 MB of virtual memory (1024 PTE entries covering 4 kB each). One of these 4 kB page table pages corresponds to the page table's own 4 MB virtual range. This is the top-level page, which maps the page table itself. It must always be accessible without causing a TLB miss for our example TLB miss handler to work. More on that later. Effectively, this example sets up a two-tier page table by taking advantage of the Xtensa MMU's hardware handling of first-level page table misses.

For this example, let's put the page table at virtual address 0xC0000000. It occupies addresses 0xC0000000 through 0xC033FFFFF, and we initialize `PTEVADDR` to 0xC0000000. The first 4 byte PTE of the page table, at address 0xC0000000, maps the first 4 kB page of virtual memory at address 0; the 2nd PTE at address 0xC0000004 maps the second 4 kB page at virtual address 0x1000, and so on. In general, the PTE corresponding to virtual address *v* is stored at address `PTEVADDR` + (*v* / 4096) * 4.

Similarly, the first 4 kB page of the page table, at 0xC0000xxx, maps the first 4 MB of virtual memory, and so on. The top-level page is located at address 0xC0000xxx + (0xC0000000 / 4096) * 4 = 0xC0300xxx. The top-level page maps the page table itself. To ensure it is always accessible without taking a miss exception, we can setup one of the 4 kB wired ways (way 7, 8, or 9) as follows:

```
#include <xtensa/coreasm.h>

#define TOP_VADDR    0xC0300000        // virt addr of top-level page
#define TOP_PADDR    ...physical address of top-level page...
#define TOP_WAY      7                 // use wired way 7
#define TOP_CA       XCHAL_CA_WRITETHRU   // make it writethru cached
#define TOP_RING     0                 // keep it ring zero (?)

      movi    a2, (TOP_VADDR & 0xFFFFF000) + TOP_WAY
      movi    a3, (TOP_PADDR & 0xFFFFF000) + (TOP_RING*16) + TOP_CA
      wdtlb   a2, a3
      dsync
```

Next, the page table must be initialized. First we initialize the top-level page. We can now write to the top-level page at virtual address `TOP_VADDR`. Typically we only need to initialize a few PTEs at the start of the top-level page, one for each 4 MB of virtual space that we need mapped. This involves allocating a physical page for each top-level PTE. Remaining PTEs are typically setup so that attempting to access the corresponding page raises an exception. There are a few options here, one of which is to initialize them to `XCHAL_CA_ILLEGAL`. If we were virtualizing tasks using protection rings, which is be-

yond the scope of this example, we might initialize empty PTEs with a non-zero ring to get a different exception when a non-privileged task attempts to access an unmapped region of memory.

Now that the top-level page is initialized and provides convenient access to other page table pages, these remaining page table pages can be initialized, much in the same way we initialized the top-level page.

Once the page table is initialized, we're just about ready to access the memory it maps. We just need the TLB miss exception handler. The minimum handler needed is deceptively simple:

```
#include <xtensa/corebits.h>
     ...
_UserExceptionVector:
     wsr     a0, EXCSAVE1        // save a0
     rsr     a0, EXCCAUSE
     beqi    a0, EXCCAUSE_ITLB_MISS, handle_miss
     addi    a0, a0, - EXCCAUSE_DTLB_MISS
     beqz    a0, handle_miss
     ...

handle_miss:
     rsr     a0, PTEVADDR
     l32i    a0, a0, 0           // read the PTE, ignore result
     rsr     a0, EXCSAVE1
     rfe
```

This handler takes advantage of the hardware PTE read. Recall that when a memory access misses in the TLB, hardware first tries to read the corresponding PTE from the page table. If that access also misses, the processor raises a TLB miss exception. In this case, the handler simply reads the PTE again. That will miss, but this time the hardware will look for a corresponding PTE in the top-level page, which never misses. The PTE is now accessible, its address is mapped in the TLB. We just throw away the result, and return to let the original instruction retry its access. It will miss again, but we know that the processor will now succeed in reading the corresponding PTE, which we have already read and discarded. The L32I instruction above is thus used strictly for its side effect of populating the TLB.

Note that in the example code above, accesses to memory not covered by the page table, *i.e.*, not mapped by the top-level page of the page table, result in the L32I instruction taking a double exception. If needed, the double exception handler must be written to anticipate and correctly handle such exceptions. Other alternatives exist, such as pointing unused top-level PTEs all to the same physical page filled with empty PTEs, so that the exception ends up being taken on the original access instead.

A. Full Context Switching Source Code

The following source code is the full set of final code from the examples. Again, recall that good code for pedagogical purposes is not, necessarily, the best code for production systems. In particular, the inclusion of additional comments and the more judicious (and liberal) use of defines is strongly recommended.

This source code (or a slight variant thereof) may be found in the AE Ware section of the Cadence technical support web site: http://support.tensilica.com/ .

A.1 Makefile

```
CC = xt-xcc
CFLAGS = -g -O2
OBJS = main.o userv.o kernv.o reset.o l1h.o intdisp.o windv.o task.o

all: exec

exec: $(OBJS)
     $(CC) -mlsp=nort $^ -o $@ -lhal

main.o: main.c timer.h interrupts.h

reset.o: reset.S
     $(CC) -g -c -mtext-section-literals $<

clean:
     rm -f $(OBJS) exec
```

A.2 interrupts.h

```
#ifndef _XTSTR
# define _XTSTR(x) # x
# define XTSTR(x) _XTSTR(x)
#endif

static __inline__ unsigned int enable_ints(unsigned int mask)
{
    extern unsigned int intMasking[2];
    unsigned int ret;
    unsigned int new_intenable;

    __asm__ __volatile__(
        "rsil    a15, 1                \n\t"
```

```
        "l32i    %0,%3,"XTSTR(INT_ENABLE_MASK)" \n\t"
        "or      %1, %0, %2             \n\t"
        "s32i    %1,%3,"XTSTR(INT_ENABLE_MASK)" \n\t"
        "l32i    %2,%3,"XTSTR(INT_LEVEL_MASK)" \n\t"
        "and     %1,%1,%2               \n\t"
        "wsr     %1,"XTSTR(INTENABLE)"  \n\t"
        "wsr     a15, "XTSTR(PS)"       \n\t"
        "rsync                          \n\t"
        : "=&a" (ret), "=&a" (new_intenable)
        : "a" (mask), "a" (intMasking)
        : "a15"
        );
    return ret;
}

static __inline__ unsigned int read_interrupt()
{
    unsigned int interrupt;

    __asm__ __volatile__ (
        "rsr     %0, "XTSTR(INTERRUPT)
        : "=a" (interrupt)
        );
    return interrupt;
}
```

A.3 intdisp.c

```
#include <xtensa/config/specreg.h>
#include <xtensa/config/core.h>
#include "interrupts.h"
#include "timer.h"
#include "task.h"

int system_ticks;

void timer0( )
{
    unsigned long old_ccompare;
    unsigned long diff;

    do {
        system_ticks++;
        old_ccompare = read_ccompare0();
        set_ccompare0( old_ccompare + TIMER_INTERVAL );
        diff = read_ccount() - old_ccompare;
    } while ( diff > TIMER_INTERVAL );
```

```
    task_tick();
}

void timer1( )
{
    unsigned long old_ccompare;
    unsigned long diff;

    do {
        old_ccompare = read_ccompare1();
        set_ccompare1( old_ccompare + TIMER_INTERVAL );
        diff = read_ccount() - old_ccompare;
    } while ( diff > TIMER_INTERVAL );
}

void intUnhandled()
{
}
```

A.4 kernv.S

```
/*
   KernelExceptionVector

   This implements the kernel exception vector and transfers
   control to the KernelExceptionHandler.
*/
#include <xtensa/config/specreg.h>

        .section .KernelExceptionVector.text, "ax"
        .begin literal_prefix .KernelExceptionVector
        .align 4

_KernelExceptionVector:
        wsr     a3, EXCSAVE_1
        movi    a3, _KernelExceptionHandler
        jx      a3

        .end literal_prefix
```

A.5 l1h.S

```
/*
  LevelOneHandlers.S
```

```
  Handle the user and kernel exceptions.
*/
#include <xtensa/coreasm.h>

        .data
        .global intMasking
        .align  4
intMasking:
        .word   0xFFFFFFFF      // interrupt level mask
        .word   0x00000000      // individual enable mask
#define INT_LEVEL_MASK          0       // offset to the level mask
#define INT_ENABLE_MASK         4       // offset to the enable mask

#define FRAME_SIZE 124        // this gets 16-byte aligned elsewhere;
                              // see references to (FRAME_SIZE + 20)


        .data
        .align 4
.global userStackPtr
userStackPtr:
        .word 0

.global newUserStackPtr
newUserStackPtr:
        .word 0

        .align  16
.global intStackBase
.global intStackEnd
intStackBase:
        .space  8192, 0
intStackEnd:
        .space  16, 0
intStackStart:

.global intHandlers
intHandlers:
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
```

```
        .word   timer0
        .word   timer1
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled
        .word   intUnhandled

        .text
        .align 4
.global _UserExceptionHandler
_UserExceptionHandler:
        rsr     a3, EXCCAUSE
        beqi    a3, EXCCAUSE_LEVEL1INTERRUPT, handleInt
        rsr     a3, EXCSAVE_1
        break   1, 1

handleInt:
        addi    a3, a1, -FRAME_SIZE

saveIntRegs:
        s32i    a0, a3, 0
        s32i    a1, a3, 4
        s32i    a2, a3, 8
        s32i    a4, a3, 16
        s32i    a5, a3, 20
        s32i    a6, a3, 24
        s32i    a7, a3, 28
        s32i    a8, a3, 32
        s32i    a9, a3, 36
        s32i    a10, a3, 40
        s32i    a11, a3, 44
        s32i    a12, a3, 48
        s32i    a13, a3, 52
```

```
        s32i    a14, a3, 56
        s32i    a15, a3, 60
        rsr     a2, EXCSAVE_1
        s32i    a2, a3, 12
        rsr     a2, LBEG
        s32i    a2, a3, 64
        rsr     a2, LEND
        s32i    a2, a3, 68
        rsr     a2, LCOUNT
        s32i    a2, a3, 72
        rsr     a2, SAR
        s32i    a2, a3, 76
        rsr     a2, PS
        s32i    a2, a3, 80
        rsr     a2, EPC_1
        s32i    a2, a3, 84
        movi    a2, 0
        wsr     a2, LCOUNT
        isync

moveToStack:
        movi    a2, userStackPtr
        s32i    a3, a2, 0
        movi    a2, newUserStackPtr
        s32i    a3, a2, 0

        addi    a2, a1, -16
        movi    a1, intStackEnd

        l32i    a0, a2, 0
        s32i    a0, a1, 0
        l32i    a0, a2, 4
        s32i    a0, a1, 4
        l32i    a0, a2, 8
        s32i    a0, a1, 8
        l32i    a0, a2, 12
        s32i    a0, a1, 12

        addi    a1, a1, 16
invokeIntHandler:

        movi    a0, intMasking
        l32i    a2, a0, INT_LEVEL_MASK
        s32i    a2, a3, 88

        rsr     a2, INTERRUPT
        rsr     a3, INTENABLE
        and     a2, a2, a3
        beqz    a2, spurious
```

```
        neg     a3, a2
        and     a2, a3, a2

        //rsil  a3, LOCKOUTLEVEL
        movi    a3, ~XCHAL_INTLEVEL1_MASK
        addi    a2, a2, -1
        or      a3, a3, a2
        addi    a2, a2, 1
        s32i    a3, a0, INT_LEVEL_MASK

        l32i    a0, a0, INT_ENABLE_MASK
        and     a3, a3, a0
        wsr     a3, INTENABLE

        movi    a0, userStackPtr
        l32i    a0, a0, 0
        l32i    a0, a0, 0

        movi    a3, PS_WOE_MASK
        wsr     a3, PS
        rsync

        find_ls_one a4, a2

        movi    a3, intHandlers
        addx4   a4, a4, a3
        l32i    a4, a4, 0

        callx4  a4

returnFromInterrupt:

        movi    a3, userStackPtr
        l32i    a3, a3, 0
        l32i    a4, a3, 80
        wsr     a4, PS
        rsync

        movi    a5, intMasking
        l32i    a2, a3, 88
        //rsil  a4, LOCKOUTLEVEL
        s32i    a2, a5, INT_LEVEL_MASK
        l32i    a5, a5, INT_ENABLE_MASK
        and     a2, a2, a5
        wsr     a2, INTENABLE

        movi    a2, intStackEnd

        l32i    a0, a2, 0
```

```
        s32i    a0, a3, FRAME_SIZE - 16
        l32i    a0, a2, 4
        s32i    a0, a3, FRAME_SIZE - 12
        l32i    a0, a2, 8
        s32i    a0, a3, FRAME_SIZE - 8
        l32i    a0, a2, 12
        s32i    a0, a3, FRAME_SIZE - 4

arRestore:
        l32i    a4, a3, 16
        l32i    a5, a3, 20
        l32i    a6, a3, 24
        l32i    a7, a3, 28
        l32i    a8, a3, 32
        l32i    a9, a3, 36
        l32i    a10, a3, 40
        l32i    a11, a3, 44
        l32i    a12, a3, 48
        l32i    a13, a3, 52
        l32i    a14, a3, 56
        l32i    a15, a3, 60

        movi    a2, userStackPtr
        movi    a1, newUserStackPtr
        l32i    a0, a2, 0
        l32i    a1, a1, 0
        beq     a0, a1, noSwitch
doSwitch:
        s32i    a1, a2, 0

        l32i    a1, a3, 4
        call0   xthal_window_spill_nw

        movi    a3, userStackPtr
        l32i    a3, a3, 0
        j       arRestore

noSwitch:
spurious:

        l32i    a2, a3, 80
        wsr     a2, PS
        rsync

        l32i    a0, a3, 84
        wsr     a0, EPC_1
        l32i    a0, a3, 76
        wsr     a0, SAR
        l32i    a0, a3, 64
```

```
        wsr     a0, LBEG
        l32i    a0, a3, 68
        wsr     a0, LEND
        l32i    a0, a3, 72
        wsr     a0, LCOUNT
        isync

        l32i    a0, a3, 0
        l32i    a1, a3, 4
        l32i    a2, a3, 8
        l32i    a3, a3, 12

        rfe

.global _KernelExceptionHandler
_KernelExceptionHandler:
        rsr     a3, EXCCAUSE
         beqi    a3, EXCCAUSE_LEVEL1INTERRUPT, nestedHandleInt
        break   1, 1

nestedHandleInt:
        addi    a3, a1, -(FRAME_SIZE + 20)

nestedSaveIntRegs:
        s32i    a0, a3, 0 + 20
        s32i    a1, a3, 4 + 20
        s32i    a2, a3, 8 + 20
        s32i    a4, a3, 16 + 20
        s32i    a5, a3, 20 + 20
        s32i    a6, a3, 24 + 20
        s32i    a7, a3, 28 + 20
        s32i    a8, a3, 32 + 20
        s32i    a9, a3, 36 + 20
        s32i    a10, a3, 40 + 20
        s32i    a11, a3, 44 + 20
        s32i    a12, a3, 48 + 20
        s32i    a13, a3, 52 + 20
        s32i    a14, a3, 56 + 20
        s32i    a15, a3, 60 + 20
        rsr     a2, EXCSAVE_1
        s32i    a2, a3, 12 + 20
        rsr     a2, LBEG
        s32i    a2, a3, 64 + 20
        rsr     a2, LEND
        s32i    a2, a3, 68 + 20
        rsr     a2, LCOUNT
        s32i    a2, a3, 72 + 20
        rsr     a2, SAR
        s32i    a2, a3, 76 + 20
```

```
        rsr     a2, PS
        s32i    a2, a3, 80 + 20
        rsr     a2, EPC_1
        s32i    a2, a3, 84 + 20

nestedMoveToStack:
        l32i    a1, a3, (FRAME_SIZE + 20) - 16
        s32i    a1, a3, 0
        l32i    a1, a3, (FRAME_SIZE + 20) - 12
        s32i    a1, a3, 4
        l32i    a1, a3, (FRAME_SIZE + 20) - 8
        s32i    a1, a3, 8
        l32i    a1, a3, (FRAME_SIZE + 20) - 4
        s32i    a1, a3, 12

        addi    a1, a3, 16

nestedInvokeIntHandler:
        movi    a0, intMasking
        l32i    a2, a0, INT_LEVEL_MASK
        s32i    a2, a3, 88 + 20

        rsr     a2, INTERRUPT
        rsr     a3, INTENABLE
        and     a2, a2, a3
        beqz    a2, spurious
        neg     a3, a2
        and     a2, a3, a2

        //rsil  a3, LOCKOUTLEVEL
        movi    a3, ~XCHAL_INTLEVEL1_MASK
        addi    a2, a2, -1
        or      a3, a3, a2
        addi    a2, a2, 1
        s32i    a3, a0, INT_LEVEL_MASK

        l32i    a0, a0, INT_ENABLE_MASK
        and     a3, a3, a0
        wsr     a3, INTENABLE

        l32i    a0, a1, (FRAME_SIZE + 20) - 16 + 0

        movi    a3, PS_WOE_MASK
        wsr     a3, PS
        rsync

        find_ls_one a4, a2

        movi    a3, intHandlers
```

```
        addx4   a4, a4, a3
        l32i    a4, a4, 0

        callx4  a4

nestedReturnFromInterrupt:
        addi    a3, a1, -16
        l32i    a4, a3, 80 + 20
        wsr     a4, PS
        rsync

        l32i    a1, a3, 0
        s32i    a1, a3, (FRAME_SIZE + 20) - 16
        l32i    a1, a3, 4
        s32i    a1, a3, (FRAME_SIZE + 20) - 12
        l32i    a1, a3, 8
        s32i    a1, a3, (FRAME_SIZE + 20) - 8
        l32i    a1, a3, 12
        s32i    a1, a3, (FRAME_SIZE + 20) - 4

        movi    a5, intMasking
        l32i    a2, a3, 88 + 20
        //rsil  a4, LOCKOUTLEVEL
        s32i    a2, a5, INT_LEVEL_MASK
        l32i    a5, a5, INT_ENABLE_MASK
        and     a2, a2, a5
        wsr     a2, INTENABLE

nestedSpurious:
        l32i    a0, a3, 84 + 20
        wsr     a0, EPC_1
        l32i    a0, a3, 76 + 20
        wsr     a0, SAR
        l32i    a0, a3, 64 + 20
        wsr     a0, LBEG
        l32i    a0, a3, 68 + 20
        wsr     a0, LEND
        l32i    a0, a3, 72 + 20
        wsr     a0, LCOUNT
        isync

        l32i    a0, a3, 0 + 20
        l32i    a1, a3, 4 + 20
        l32i    a2, a3, 8 + 20
        l32i    a4, a3, 16 + 20
        l32i    a5, a3, 20 + 20
        l32i    a6, a3, 24 + 20
        l32i    a7, a3, 28 + 20
        l32i    a8, a3, 32 + 20
```

```
        l32i    a9, a3, 36 + 20
        l32i    a10, a3, 40 + 20
        l32i    a11, a3, 44 + 20
        l32i    a12, a3, 48 + 20
        l32i    a13, a3, 52 + 20
        l32i    a14, a3, 56 + 20
        l32i    a15, a3, 60 + 20
        l32i    a3, a3, 12 + 20

        rfe
```

A.6 main.c

```
#include <xtensa/config/specreg.h>
#include "timer.h"
#include "interrupts.h"
#include "task.h"

#define STACK_SIZE 0x2000

int global_fool_the_optimizer = 0;
int failed = 0;

void deep_call( int depth )
{
    int a = depth;
    int b = depth + 1;
    int c = depth + 2;
    int d = depth + 3;

    if( global_fool_the_optimizer )
    {
        a = b = c = d = 10;
    }

    if( depth > 0 )
        deep_call( depth - 1 );

    if( a != depth ||
        b != depth + 1 ||
        c != depth + 2 ||
        d != depth + 3)
    {
        failed++;
    }
}
```

```
unsigned char task1_stack[STACK_SIZE];
void task1( void *arg )
{
    while( 1 )
        deep_call( (int) arg );
}

unsigned char task2_stack[STACK_SIZE];
void task2( void *arg )
{
    while( 1 )
        deep_call( (int) arg );
}

main()
{
    add_task( task1, task1_stack, STACK_SIZE, (void *)10 );
    add_task( task2, task2_stack, STACK_SIZE, (void *)20 );

    set_ccompare0( read_ccount() + TIMER_INTERVAL );
    enable_ints( TIMER_INT_MASK );
    wait_func();
}

wait_func()
{
        while(1);
}
```

A.7 reset.S

```
#include <xtensa/coreasm.h>
#include <xtensa/simcall.h>

        .global _ResetVector

        .section        .ResetVector.text, "ax"
        .align  4
_ResetVector:
        j       Reset           // jump around literal pool


        .literal_position  // tells assembler place literals here

        .align  4
Reset:
        movi    a0, 0
```

```
        wsr     a0, INTENABLE

        wsr     a0, DBREAKC_0
        wsr     a0, DBREAKC_1

        wsr     a0, ICOUNTLEVEL
        isync

        rsil    a1, XCHAL_DEBUGLEVEL - 1

        wsr     a0, CCOUNT
        wsr     a0, WINDOWBASE
        rsync

        movi    a0, 16
arloop:
        addi    a4, a0, -1
        movi    a0, 0
        movi    a1, 0
        movi    a2, 0
        movi    a3, 0
        rotw    1
        bnez    a0, arloop

        movi    a1, 1
        wsr     a1, WINDOWSTART

        ssai    0

        wsr     a0, EXCSAVE_1
        wsr     a0, EPC_1
        wsr     a0, EXCCAUSE

        wsr     a0, EPC_2
        wsr     a0, EPS_2
        wsr     a0, EXCSAVE_2

        wsr     a0, EPC_3
        wsr     a0, EPS_3
        wsr     a0, EXCSAVE_3

        wsr     a0, EPC_4
        wsr     a0, EPS_4
        wsr     a0, EXCSAVE_4

        wsr     a0, CCOMPARE_0
        wsr     a0, CCOMPARE_1
        wsr     a0, CCOMPARE_2
```

```
        movi    a2, XCHAL_INTTYPE_MASK_EXTERN_EDGE |
XCHAL_INTTYPE_MASK_SOFTWARE
        wsr     a2, INTCLEAR

        wsr     a0, BR
        wsr     a0, CPENABLE

        movi    a2, XCHAL_DEBUGLEVEL - 1
        wsr     a2, PS
        rsync

        movi    a5, 0xE0000000
        movi    a4, 2
        movi    a3, 0
        j       3f
2:      sub     a3, a3, a5
        sub     a4, a4, a5
3:      wdtlb   a4, a3
        witlb   a4, a3
        bne     a3, a5, 2b
        isync

        movi    a2, 128
        movi    a3, 0
        loop    a2, .L0
        iii     a3, 0
        iii     a3, 16
        iii     a3, 32
        iii     a3, 48
        addi    a3, a3, 64
.L0:
        isync

        movi    a2, 128
        movi    a3, 0
        loop    a2, .L1
        diu     a3, 0
        diu     a3, 16
        diu     a3, 32
        diu     a3, 48
        addi    a3, a3, 64
.L1:

        movi    a2, 64
        movi    a3, 0
        loop    a2, .L2
        dii     a3, 0
        dii     a3, 16
        dii     a3, 32
```

```
        dii     a3, 48
        addi    a3, a3, 64
.L2:
        dsync

        .macro  set_access_mode am
        rdtlb1  a4, a3
        ritlb1  a5, a3
        srli    a4, a4, 4
        slli    a4, a4, 4
        srli    a5, a5, 4
        slli    a5, a5, 4
        addi    a4, a4, \am
        addi    a5, a5, \am
        wdtlb   a4, a3
        witlb   a5, a3
        .endm

        movi    a2, 0x20000000  // region address increment
        movi    a3, 0           // start at region 0

        // Turn caches on for regions 0x40000000 and 0x80000000
        // Disable all accesses for 0, 0xC0000000, and 0xE0000000
        // Leave all other regions unchanged (i.e., uncached).

        set_access_mode 15
        add     a3, a3, a2      // a3 <-- 0x20000000
        add     a3, a3, a2      // a3 <-- 0x40000000
        set_access_mode 1
        add     a3, a3, a2      // a3 <-- 0x60000000
        add     a3, a3, a2      // a3 <-- 0x80000000
        set_access_mode 1
        add     a3, a3, a2      // a3 <-- 0xA0000000
        add     a3, a3, a2      // a3 <-- 0xC0000000
        set_access_mode 15
        add     a3, a3, a2      // a3 <-- 0xE0000000
        set_access_mode 15


        movi    a2, _rom_store_table
        beqz    a2, unpackdone

unpack:
        l32i    a3, a2, 0       // start vaddr
        l32i    a4, a2, 4       // end vaddr
        l32i    a5, a2, 8       // store vaddr
        bltu    a3, a4, unpack1
        bnez    a3, unpacknext
        bnez    a5, unpacknext
```

```
        j       unpackdone

unpack1:
        l32i    a6, a5, 0
        addi    a5, a5, 4
        s32i    a6, a3, 0
        addi    a3, a3, 4
        bltu    a3, a4, unpack1

unpacknext:
        addi    a2, a2, 12
        j       unpack

unpackdone:
        movi    sp, __stack

        movi    a2, PS_WOE_MASK | PS_PROGSTACK_MASK
        wsr     a2, PS
        rsync

#ifdef BSS_SLOW
        movi    a8, _bss_start
        movi    a10, _bss_end
        sub     a11, a10, a8
        srli    a11, a11, 2

        movi    a9, 0
        loopnez a11, zerodone
        s32i    a9, a8, 0
        addi    a8, a8, 4
zerodone:
#else
        movi    a8, _bss_start
        movi    a10, _bss_end
        sub     a11, a10, a8
        slli    a12, a11, 28
        srli    a12, a12, 30

        movi    a9, 0
        loopnez a12, zerodone_fourbyte
        s32i    a9, a8, 0
        addi    a8, a8, 4
zerodone_fourbyte:

        srli    a11, a11, 4
        loopnez a11, zerodone_16byte
        s32i    a9, a8, 0
        s32i    a9, a8, 4
        s32i    a9, a8, 8
```

```
        s32i    a9, a8, 12
        addi    a8, a8, 16
zerodone_16byte:

#endif
callmain:
        movi    a0, 0
        movi    a6, 0   // clear argc
        movi    a7, 0   // clear argv
        movi    a8, 0   // clear envp
        movi    a4, main
        callx4  a4

reset_exit:
        movi    a2, SYS_exit
        simcall
```

A.8 task.c

```
#include <xtensa/config/specreg.h>
#include "task.h"

unsigned int *task_stacks[MAXTASKS];
unsigned int allocated_tasks = 0;
int current_task = -1;

void add_task( TaskFunc *entry,
               void *stack,
               unsigned int stack_size,
               void *arg )
{
    unsigned int *sp;
    unsigned int stack_words;
    int i;

    sp = (unsigned int *)((int)(stack + stack_size) & 0xfffffff0);
    sp = sp - FRAME_SIZE / 4;

    for( i = 0 ; i < FRAME_SIZE / 4 ; i++ )
        sp[i] = 0;

    sp[ FRAME_PC/4 ] = (unsigned int)entry;
    sp[ FRAME_PS/4 ] = PS_WOE_MASK  | PS_UM_MASK |
                       PS_EXCM_MASK | (1 << PS_CALLINC_SHIFT);
    sp[ FRAME_AR(6)/4 ] = (int)arg;
    sp[ FRAME_AR(1)/4 ] = (int)(sp + FRAME_SIZE / 4);
```

```
    task_stacks[allocated_tasks++] = sp;
}

extern int *newUserStackPtr;
extern int *userStackPtr;
void task_tick( void )
{
    if( current_task >= 0 )
        task_stacks[current_task] = userStackPtr;

    if( ++current_task == allocated_tasks )
        current_task = 0;
    newUserStackPtr = task_stacks[current_task];
}
```

A.9 task.h

```
#define MAXTASKS 10

extern unsigned int *task_stacks[];
extern unsigned int allocated_tasks;

typedef void (TaskFunc)(void *arg);

void add_task( TaskFunc *entry, void *stack, unsigned int stack_size,
               void *arg );

void task_tick( void );

/* frame defines */
#define FRAME_SIZE 124        // this gets 16-byte aligned elsewhere;
                              // see references to (FRAME_SIZE + 20)
#define FRAME_AR(x) (x * 4)
#define FRAME_LBEG 64
#define FRAME_LEND 68
#define FRAME_LCOUNT 72
#define FRAME_SAR 76
#define FRAME_PS 80
#define FRAME_PC 84
```

A.10 timer.h

```
#include <xtensa/config/core.h>

#define TIMER_INTERVAL 0x1000
```

```
#define TIMER_INT_MASK  (1 << XCHAL_TIMER0_INTERRUPT)
#define TIMER2_INT_MASK (1 << XCHAL_TIMER1_INTERRUPT)
#define TWO_TIMERS_INT_MASK ( TIMER_INT_MASK + TIMER2_INT_MASK )

#define _XTSTR(x) # x
#define XTSTR(x) _XTSTR(x)

static __inline__ int read_ccount()
{
    unsigned int ccount;

    __asm__ __volatile__ (
        "rsr     %0, "XTSTR(CCOUNT)
        : "=a" (ccount)
        );
    return ccount;
}

static __inline__ int read_ccompare0()
{
    unsigned int ccompare0;

    __asm__ __volatile__ (
        "rsr     %0, "XTSTR(CCOMPARE_0)
        : "=a" (ccompare0)
        );
    return ccompare0;
}

static __inline__ int read_ccompare1()
{
    unsigned int ccompare1;

    __asm__ __volatile__ (
        "rsr     %0, "XTSTR(CCOMPARE_1)
        : "=a" (ccompare1)
        );
    return ccompare1;
}

static __inline__ unsigned int read_intenable()
{
    unsigned int intenable;

    __asm__ __volatile__ (
        "rsr     %0, "XTSTR(INTENABLE)
        : "=a" (intenable)
        );
    return intenable;
```

```
}

static __inline__ void set_ccompare1(int val)
{
    __asm__ __volatile__ (
        "wsr     %0, "XTSTR(CCOMPARE_1)"\n\t"
        "isync\n\t"
        :
        : "a" (val)
        );
}

static __inline__ void set_ccompare0(int val)
{
    __asm__ __volatile__ (
        "wsr     %0, "XTSTR(CCOMPARE_0)"\n\t"
        "isync\n\t"
        :
        : "a" (val)
        );
}
```

A.11 userv.S

```
/*
   UserExceptionVector

   This implements the user exception vector and transfers
   control to the UserExceptionHandler.
*/
#include <xtensa/config/specreg.h>

        .section .UserExceptionVector.text, "ax"
        .begin literal_prefix .UserExceptionVector
        .align 4

_UserExceptionVector:
        wsr     a3, EXCSAVE_1
        movi    a3, _UserExceptionHandler
        jx      a3

        .end literal_prefix
```

A.12 windv.S

```
// Register Window Overflow/Underflow Handlers

// Copyright 1998-2004 Tensilica Inc.
// These coded instructions, statements, and computer programs are
// Confidential Proprietary Information of Tensilica Inc. and may not be
// disclosed to third parties or copied in any form, in whole or in part,
// without the prior written consent of Tensilica Inc.

        .section .WindowVectors.text, "ax"
        .begin literal_prefix .WindowVectors
        .align 4



// Exports
.global _WindowOverflow4
.global _WindowUnderflow4
.global _WindowOverflow8
.global _WindowUnderflow8
.global _WindowOverflow12
.global _WindowUnderflow12

// Imports
//   None

// NOTE: The underflow handler for returning from call[i+1] to call[i] must
// preserve all the registers from call[i+1]'s window.  a0 and a1 must be
// preserved because the retw instruction will be reexecuted (and may even
// underflow if an intervening exception has flushed call[i]'s registers).
// Registers a2 and up may contain return values.

// inside call[i] referencing a register that
// contains data from call[j]
// On entry here: window rotated to call[j] start point; the
// registers to be saved are a0-a3; a4-a15 must be preserved
// a5 is call[j+1]'s stack pointer
```

```
        .align 64
_WindowOverflow4:
        addi    a5, a5, -16 // to make store offsets positive
        s32i    a0, a5,   0 // save a0 to call[j+1]'s stack frame
        s32i    a1, a5,   4 // save a1 to call[j+1]'s stack frame
        s32i    a2, a5,   8 // save a2 to call[j+1]'s stack frame
        s32i    a3, a5,  12 // save a3 to call[j+1]'s stack frame
        addi    a5, a5,  16 // restore a5
        rfwo                // rotates back to call[i] position

// returning from call[i+1] to call[i] where
// call[i]'s registers must be reloaded
// On entry here: a0-a3 will be reloaded with
// call[i].reg[0..3] but initially contain garbage.
// a4-a15 are call[i+1].reg[0..11],
// (in particular, a5 is call[i+1]'s stack pointer)
// and must be preserved

        .align 64
_WindowUnderflow4:
        addi    a3, a5, -16 // to make load offsets positive
        l32i    a0, a3,   0 // restore a0 from call[i+1]'s stack frame
        l32i    a1, a3,   4 // restore a1 from call[i+1]'s stack frame
        l32i    a2, a3,   8 // restore a2 from call[i+1]'s stack frame
        l32i    a3, a3,  12 // restore a3 from call[i+1]'s stack frame
        rfwu

// On entry here: window rotated to call[j]; the registers to be
// saved are a0-a7; a8-a15 must be preserved
// a9 is call[j+1]'s stack pointer

        .align 64
_WindowOverflow8:
        addi    a9, a9, -16 // to make store offsets positive
        s32i    a0, a9,   0 // save a0 to call[j+1]'s stack frame
        s32i    a1, a9,   4 // save a1 to call[j+1]'s stack frame
```

```
        s32i    a2, a9,   8 // save a2 to call[j+1]'s stack frame
        s32i    a3, a9,  12 // save a3 to call[j+1]'s stack frame
        addi    a9, a9,  16 // restore a9
        addi    a0, a1, -16 // a0 <- call[j-1]'s sp
        l32i    a0, a0,   4 // (used to find end of call[j]'s frame)
        addi    a0, a0, -32 // to make load offsets positive
        s32i    a4, a0,   0 // save a4 to call[j]'s stack frame
        s32i    a5, a0,   4 // save a5 to call[j]'s stack frame
        s32i    a6, a0,   8 // save a6 to call[j]'s stack frame
        s32i    a7, a0,  12 // save a7 to call[j]'s stack frame
        rfwo                // rotates back to call[i] position

// On entry here: a0-a7 are call[i].reg[0..7] and initially
// contain garbage, a8-a15 are call[i+1].reg[0..7],
// (in particular, a9 is call[i+1]'s stack pointer)
// and must be preserved

        .align 64
_WindowUnderflow8:
        addi    a9, a9, -16 // to make load offsets positive
        l32i    a0, a9,   0 // restore a0 from call[i+1]'s stack frame
        l32i    a1, a9,   4 // restore a1 from call[i+1]'s stack frame
        l32i    a2, a9,   8 // restore a2 from call[i+1]'s stack frame
        // don't restore a3 yet because we need it as a temporary
        addi    a3, a1, -16 // a3 <- call[i-1]'s sp
        l32i    a3, a3,   4 // (used to find end of call[i]'s frame)
        addi    a3, a3, -32 // to make load offsets positive
        l32i    a4, a3,   0 // restore a4 from call[i]'s stack frame
        l32i    a5, a3,   4 // restore a5 from call[i]'s stack frame
        l32i    a6, a3,   8 // restore a6 from call[i]'s stack frame
        l32i    a7, a3,  12 // restore a7 from call[i]'s stack frame
        // we're done with a3 and can restore it now
        l32i    a3, a9,  12 // restore a3 from call[i+1]'s stack frame
        addi    a9, a9,  16 // restore a9
        rfwu
```

```
// On entry here: window rotated to call[j]; the registers to be
// saved are a0-a11 ; a12-a15 must be preserved
// a13 is call[j+1]'s stack pointer

        .align 64
_WindowOverflow12:
        addi    a13, a13, -16 // to make store offsets positive
        s32i    a0,  a13,   0 // save a0 to call[j+1]'s stack frame
        s32i    a1,  a13,   4 // save a1 to call[j+1]'s stack frame
        s32i    a2,  a13,   8 // save a2 to call[j+1]'s stack frame
        s32i    a3,  a13,  12 // save a3 to call[j+1]'s stack frame
        addi    a13, a13,  16 // restore a13
        addi    a0,  a1,  -16 // a0 <- call[j-1]'s sp
        l32i    a0,  a0,    4 // (used to find end of call[j]'s frame)
        addi    a0,  a0,  -48 // to make load offsets positive
        s32i    a4,  a0,    0 // save a4 to end of call[j]'s stack frame
        s32i    a5,  a0,    4 // save a5 to end of call[j]'s stack frame
        s32i    a6,  a0,    8 // save a6 to end of call[j]'s stack frame
        s32i    a7,  a0,   12 // save a7 to end of call[j]'s stack frame
        s32i    a8,  a0,   16 // save a8 to end of call[j]'s stack frame
        s32i    a9,  a0,   20 // save a9 to end of call[j]'s stack frame
        s32i    a10, a0,   24 // save a10 to end of call[j]'s stack frame
        s32i    a11, a0,   28 // save a11 to end of call[j]'s stack frame
        rfwo                  // rotates back to call[i] position

// On entry here:
// a0-a11 are call[i].reg[0..11] and initially contain garbage
// a12-a15 are call[i+1].reg[0..3], (in particular, a13 is call[i+1]'s stack pointer)
// and must be preserved

        .align 64
_WindowUnderflow12:
        addi    a13, a13, -16 // to make load offsets positive
        l32i    a0,  a13,   0 // restore a0 from call[i+1]'s stack frame
        l32i    a1,  a13,   4 // restore a1 from call[i+1]'s stack frame
        l32i    a2,  a13,   8 // restore a2 from call[i+1]'s stack frame
```

```
// don't restore a3 yet because we need it as a temporary
addi    a3,  a1,  -16 // a3 <- call[i-1]'s sp
l32i    a3,  a3,    4 // (used to find end of call[i]'s frame)
addi    a3,  a3,  -48 // to make load offsets positive
l32i    a4,  a3,    0 // restore a4 from end of call[i]'s stack frame
l32i    a5,  a3,    4 // restore a5 from end of call[i]'s stack frame
l32i    a6,  a3,    8 // restore a6 from end of call[i]'s stack frame
l32i    a7,  a3,   12 // restore a7 from end of call[i]'s stack frame
l32i    a8,  a3,   16 // restore a8 from end of call[i]'s stack frame
l32i    a9,  a3,   20 // restore a9 from end of call[i]'s stack frame
l32i    a10, a3,   24 // restore a10 from end of call[i]'s stack frame
l32i    a11, a3,   28 // restore a11 from end of call[i]'s stack frame
// we're done with a3 and can restore it now
l32i    a3,  a13,  12 // restore a3 from call[i+1]'s stack frame
addi    a13, a13,  16 // restore a13
rfwu

.align 64

.end literal_prefix
```

B. MMU Code Examples

The following source code is the full set of final code from the examples. This code is for pedagogical purposes only. It will not compile, as it largely consists of code excerpts from an early version of the Xtensa architecture portions of the Linux operating system.

B.1 simple.S

```
/*
 * extern void map_region (unsigned vpn, unsigned ppn);
 *
 * Maps the selected virtual 512MB segment to the specified physical
 * 512MB segment.  (Lower 29 bits of both parameters are ignored.)
 * VPN in current segment case is not handled
 *
 * Entry:
 *      a2              VPN/TLB entry to set
 *      a3              PPN to target
 * Exit:
 *      None
 */


        .text
        .global map_region
        .type   map_region,@function
        .align  4
map_region:
        entry   sp, 64

# if XCHAL_HAVE_XLT_CACHEATTR

        movi    a5, 0xE0000000  // tlb mask, upper 3 bits
        movi    a6, 0f          // PC where ITLB is set
        and     a4, a3, a5      // upper 3 bits of PPN area
        and     a7, a2, a5      // upper 3 bits of VPN area
        and     a6, a6, a5      // upper 3 bits of local PC area
```

```
        beq     a7, a6, 1f      // branch if current PC's region

        // Note that in the WITLB section, we don't do any load/stores.
        // May not be an issue, but it's important in the DTLB case.

        ritlb1  a5, a7          // get current PPN+AM of segment for I
        rdtlb1  a6, a7          // get current PPN+AM of segment for D
        extui   a5, a5, 0, 4    // keep only AM for I
        extui   a6, a6, 0, 4    // keep only AM for D
        add     a2, a4, a5      // combine new PPN with orig AM for I
        add     a3, a4, a6      // combine new PPN with orig AM for D
0:      witlb   a2, a7          // write new tlb mapping for I
        wdtlb   a3, a7          // write new tlb mapping for D
        isync
        j       2f:
1:
        // Handle if VPN is current segment
2:

# endif
        retw
```

B.2 mmu_context.h

```
/*
 * include/asm-xtensa/mmu_context.h
 *
 * Switch an MMU context.
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License.  See the file "COPYING" in the main directory of
 * this archive for more details.
 *
 * Copyright (C) 2001 Tensilica Inc.
 */
```

```
#include <linux/config.h>
#include <asm/pgalloc.h>
#include <asm/pgtable.h>


#define WIRED_WAY_FOR_PAGE_TABLE  7

extern inline void set_rasid_register (unsigned long val)
{
        __asm__ __volatile__ (" wsr   %0, "XTSTR(RASID)"\n\t"
                                       " isync\n"
                                       : : "a" (val));
}

extern inline unsigned long get_rasid_register (void)
{
        unsigned long tmp;
        __asm__ __volatile__ (" rsr   %0, "XTSTR(RASID)"\n\t"
                                       " isync\n"
                                    : "=a" (tmp));
        return tmp;
}

extern inline unsigned long itlb_probe(unsigned long addr)
{
        unsigned long tmp;
        __asm__ __volatile__("pitlb  %0, %1\n\t"
                                       : "=a" (tmp)
                                       : "a" (addr));
        return tmp;
}

extern inline unsigned long dtlb_probe(unsigned long addr)
{
        unsigned long tmp;
        __asm__ __volatile__("pdtlb  %0, %1\n\t"
```

```
                                        : "=a" (tmp)
                                        : "a" (addr));
        return tmp;
}

extern inline void invalidate_itlb_entry (unsigned long probe)
{
        __asm__ __volatile__("iitlb  %0\n\t"
                                        "isync\n\t"
                                        : : "a" (probe));
}

extern inline void invalidate_dtlb_entry (unsigned long probe)
{
        __asm__ __volatile__("idtlb  %0\n\t"
                                        "dsync\n\t"
                                        : : "a" (probe));
}

extern inline void set_itlbcfg_register (unsigned long val)
{
        __asm__ __volatile__("wsr  %0, "XTSTR(ITLBCFG)"\n\t"
                                        "isync\n\t"
                                        : : "a" (val));
}

extern inline void set_dtlbcfg_register (unsigned long val)
{
        __asm__ __volatile__("wsr  %0, "XTSTR(DTLBCFG)"\n\t"
                                        "dsync\n\t"
                                        : : "a" (val));
}

extern inline void set_ptevaddr_register (unsigned long val)
{
        __asm__ __volatile__(" wsr  %0, "XTSTR(PTEVADDR)"\n\t"
```

```
                                        " isync\n"
                                        : : "a" (val));
}

extern inline unsigned long read_ptevaddr_register (void)
{
        unsigned long tmp;
        __asm__ __volatile__("rsr  %0, "XTSTR(PTEVADDR)"\n\t"
                                        : "=a" (tmp));
        return tmp;
}

extern inline void write_dtlb_entry (pte_t entry, int way)
{
        __asm__ __volatile__("wdtlb  %1, %0\n\t"
                                    "dsync\n\t"
                                        : : "r" (way), "r" (entry) );
}

extern inline void write_itlb_entry (pte_t entry, int way)
{
        __asm__ __volatile__("witlb  %1, %0\n\t"
                                        "isync\n\t"
                                        : : "r" (way), "r" (entry) );
}
```

B.3 fault.c

```
/*
 * arch/xtensa/mm/fault.c
 *
 * Derived from MIPS.
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file "COPYING" in the main directory of this archive
 * for more details.
```

```
 *
 * Copyright (C) 1995 Linus Torvalds
 * Copyright (C) 2004 Tensilica Inc.
 */

#include <linux/signal.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/types.h>
#include <linux/ptrace.h>
#include <linux/mman.h>
#include <linux/mm.h>
#include <linux/smp.h>
#include <linux/smp_lock.h>
#include <linux/version.h>

#include <asm/hardirq.h>
#include <asm/pgalloc.h>
#include <asm/mmu_context.h>
#include <asm/softirq.h>
#include <asm/system.h>
#include <asm/uaccess.h>

#define development_version (LINUX_VERSION_CODE & 0x100)

unsigned long asid_cache = ASID_FIRST_VERSION;



/*
 *  do_page_fault() could handle:
 *
 *      [i,d]priv_error
 *      [i,d]sr_error  (size restrictions)
```

```
 *       fetchAttrError
 *       loadAttrError
 *       storeAttrError
 *
 *  but not:
 *
 *       miss_error (2nd-level miss)
 */


/*
 * Macro for exception fixup code to access integer registers.
 */
#define dpf_reg(r) (regs->regs[r])

/*
 * This routine handles page faults.  It determines the address,
 * and the problem, and then passes it off to one of the appropriate
 * routines.
 */
asmlinkage void do_page_fault(struct pt_regs *regs)
{
        struct vm_area_struct * vma;
        struct task_struct *tsk = current;
        struct mm_struct *mm = tsk->mm;
        unsigned long fixup, tlb_entry;
        unsigned address = regs->excvaddr;
        siginfo_t info;
        pgd_t *pgd;
        pmd_t *pmd;
        pte_t *pte;
        unsigned write = 0;

        pgd = pgd_offset (mm, address); /* or, 'current_pgd' ? */
        pmd = pmd_offset (pgd, address);
        if ( ! (pmd_present(*pmd)) )
```

```
        goto out_of_memory; /* should never happen! */
pte = pte_offset (pmd, address);

switch( regs->exccause ) {
case XCHAL_EXCCAUSE_ITLB_PRIVILEGE:

        /* Check for uninitialized PTEs here.  If we find one,
         * invalidate the ITLB entry.  Please refer to pte_alloc() and
         * get_pte_slow() for more details. */

        if (pte_none(*pte)) {
                if ((tlb_entry = itlb_probe (address)) & ITLB_PROBE_SUCCESS)
                        invalidate_itlb_entry (tlb_entry);
        }

        write = 2;/* i-fetch */
        break;

case XCHAL_EXCCAUSE_FETCH_CACHE_ATTRIBUTE:

        if (pte_present(*pte) && pte_read(*pte)) {
                pte_val(*pte) |= (_PAGE_ACCESSED | _PAGE_VALID);
                /* Invalidate the dtlb entry so the MMU will
                  reload the new pte. */
                if ((tlb_entry = itlb_probe (address)) & ITLB_PROBE_SUCCESS)
                        invalidate_itlb_entry (tlb_entry);
                return;
        }
        write = 2;/* i-fetch */
        break;

case XCHAL_EXCCAUSE_LOAD_CACHE_ATTRIBUTE:

        if (pte_present(*pte) && pte_read(*pte)) {
                pte_val(*pte) |= (_PAGE_ACCESSED | _PAGE_VALID);
                /* Invalidate the dtlb entry so the MMU will
```

```
                          reload the new pte. */
                        if ((tlb_entry = dtlb_probe (address)) & DTLB_PROBE_SUCCESS)
                                invalidate_dtlb_entry (tlb_entry);
                        return;
                }
                write = 0;
                break;

        case XCHAL_EXCCAUSE_STORE_CACHE_ATTRIBUTE:

                if (pte_present(*pte) && pte_write(*pte)) {
                        pte_val(*pte) |= (_PAGE_ACCESSED | _PAGE_MODIFIED | _PAGE_VALID |
_PAGE_DIRTY);
                        /* Invalidate the dtlb entry so the MMU will
                          reload the new pte. */
                        if ((tlb_entry = dtlb_probe (address)) & DTLB_PROBE_SUCCESS)
                                invalidate_dtlb_entry (tlb_entry);
                        return;
                }
                write = 1;
                break;

        case XCHAL_EXCCAUSE_DTLB_PRIVILEGE:

                /* Check for uninitialized PTEs here.  If we find one,
                 * invalidate the DTLB entry.  Please refer to pte_alloc() and
                 * get_pte_slow() for more details. */

                if (pte_none(*pte)) {
                        if ((tlb_entry = dtlb_probe (address)) & DTLB_PROBE_SUCCESS)
                                invalidate_dtlb_entry (tlb_entry);
                }

                write = 3;
                break;
        default:
                printk ("Unhandled exccause in do_page_fault()\n");
```

```
                xt_panic();
                break;
        }

        info.si_code = SEGV_MAPERR;
        /*
         * If we're in an interrupt or have no user
         * context, we must not take the fault..
         */
        if ( in_interrupt() || !mm )
                goto no_context;
#if 0
        printk("[%s:%d:%08x:%d:%08x]\n", tsk->comm, tsk->pid,
               address, regs->exccause, regs->pc);
#endif
        down_read(&mm->mmap_sem);
        vma = find_vma(mm, address);
        if (!vma)
                goto bad_area;
        if (vma->vm_start <= address)
                goto good_area;
        if (!(vma->vm_flags & VM_GROWSDOWN))
                goto bad_area;
        if (expand_stack(vma, address))
                goto bad_area;
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
 */
good_area:
        info.si_code = SEGV_ACCERR;

        if (write == 1) {/* store */
                if (!(vma->vm_flags & VM_WRITE))
                        goto bad_area;
        } else if(write == 2) {/* i-fetch */
```

```
                if (!(vma->vm_flags & VM_EXEC))
                        goto bad_area;
        } else {                /* load */
                if (!(vma->vm_flags & VM_READ))
                        goto bad_area;
        }

        /*
         * If for any reason at all we couldn't handle the fault,
         * make sure we exit gracefully rather than endlessly redo
         * the fault.
         */
        switch (handle_mm_fault(mm, vma, address, write)) {
        case 1:
                tsk->min_flt++;
                break;
        case 2:
                tsk->maj_flt++;
                break;
        case 0:
                goto do_sigbus;
        default:
                goto out_of_memory;
        }

        up_read(&mm->mmap_sem);
        return;

/*
 * Something tried to access memory that isn't in our memory map..
 * Fix it, but check if it's kernel or user first..
 */
bad_area:
        up_read(&mm->mmap_sem);

        if (user_mode(regs)) {
```

```
                tsk->thread.bad_vaddr = address;
                tsk->thread.error_code = write;
#if 1
                printk("do_page_fault() #2: sending SIGSEGV to %s for illegal %s\n"
                       "%08x (pc == %08x, ra == %08x, write == %d)\n",
                       tsk->comm,
                       write ? "write access to" : "read access from",
                       address,
                       regs->pc,
                       (regs->aregs[0] & 0x3FFFFFFF) | (regs->pc & 0xC0000000),
                       write);
#endif
                info.si_signo = SIGSEGV;
                info.si_errno = 0;
                /* info.si_code has been set above */
                info.si_addr = (void *) address;
                force_sig_info(SIGSEGV, &info, tsk);
                return;
        }

no_context:
        /* Are we prepared to handle this kernel fault?  */
        fixup = search_exception_table(regs->pc);
        if (fixup) {
                long new_epc;

                tsk->thread.bad_uaddr = address;
                new_epc = fixup_exception(dpf_reg, fixup, regs->pc);
                if (development_version)
                        printk(KERN_DEBUG "%s: Exception at [<%x>] (%lx)\n",
                               tsk->comm, regs->pc, new_epc);
                regs->pc = new_epc;
                return;
        }

        /*
```

```
         * Oops. The kernel tried to access some bad page. We'll have to
         * terminate things with extreme prejudice.
         */
        printk(KERN_ALERT "Unable to handle kernel paging request at virtual "
               "address %08x, epc == %08x, ra == %08x\n",
               address, regs->pc, regs->aregs[0]);
        die("Oops", regs);
        do_exit(SIGKILL);

/*
 * We ran out of memory, or some other thing happened to us that made
 * us unable to handle the page fault gracefully.
 */
out_of_memory:
        up_read(&mm->mmap_sem);
        printk("VM: killing process %s\n", tsk->comm);
        if (user_mode(regs))
                do_exit(SIGKILL);
        goto no_context;

do_sigbus:
        up_read(&mm->mmap_sem);

        /*
         * Send a sigbus, regardless of whether we were in kernel
         * or user vector mode.
         */
        tsk->thread.bad_vaddr = address;
        info.si_code = SIGBUS;
        info.si_errno = 0;
        info.si_code = BUS_ADRERR;
        info.si_addr = (void *) address;
        force_sig_info(SIGBUS, &info, tsk);

        /* Kernel vector mode? Handle exceptions or die */
        if (!user_mode(regs))
```

```
                goto no_context;
}
```

B.4 mmu.c

```
/*
 * arch/xtensa/mm/mmu.c
 *
 * Logic that manipulates the Xtensa MMU.
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file "COPYING" in the main directory of this archive
 * for more details.
 *
 * Copyright (C) 2004 Tensilica Inc.
 */

#include <xtensa/config/core.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <asm/mmu_context.h>
#include <asm/xtutil.h>
#include <asm/pgtable.h>
#include <asm/pgalloc.h>


void flush_tlb_all (void)
{
        unsigned long flags;
        save_and_cli (flags);
        __asm__ __volatile__ ("iitlba\n\t"
                                        "idtlba\n\t"
                                        "isync\n\t"); /* isync includes dsync */
        restore_flags (flags);
}
```

```
void init_mmu (void)
{

        /* Writing zeros to the <t>TLBCFG special registers ensure
         * that valid values exist in the register.  For existing
         * PGSZID<w> fields, zero selects the first element of the
         * page-size array.  For nonexistant PGSZID<w> fields, zero is
         * the best value to write.  Also, when changing PGSZID<w>
         * fields, the corresponding TLB must be flushed. */

        set_itlbcfg_register (0);
        set_dtlbcfg_register (0);
        flush_tlb_all ();

        /* Set rasid register to a known value. */

        set_rasid_register (ASID_ALL_RESERVED);

        /*
         *  Set PTEVADDR special register to the start of the page table,
         *  which is in kernel mappable space (i.e., not statically mapped).
         *  This register's value is undefined on reset.
         */

        set_ptevaddr_register (PGTABLE_START);

}


void handle_2nd_level_miss (struct pt_regs *regs)
{
        struct task_struct *tsk = current;
        unsigned long vpnval;
        pgd_t *pgd;
        pmd_t *pmd;
        pmd_t pmdval;
```

```
        pte_t pteval;

        /*  We need to map the page of PTEs for the user task.  Find
         *  the pointer to that page.  */

        pgd = pgd_offset (tsk->mm, regs->excvaddr);
        pmd = pmd_offset (pgd, regs->excvaddr);

        /* We want to map the page of PTEs into the Page Table, but if
         * the task doesn't yet have a mapping for the region, just
         * map the exception_pte_table for the region.  Note that we
         * do not modify the mappings for the task (tsk->mm->pgd) itself.
         *
         * exception_pte_table contains PTEs that will generate access
         * faults, and the exception will end up in do_page_fault()
         * for further handling. */

        pmdval = *pmd;
        if (pmd_none(pmdval))
                pmdval = __pmd((unsigned long)exception_pte_table);

        /* read ptevaddr and convert to top of page-table page */
        vpnval = read_ptevaddr_register() & PAGE_MASK;
        vpnval += WIRED_WAY_FOR_PAGE_TABLE; /* add way number for 'wdtlb' insn */
        pteval = mk_pte (virt_to_page(pmd_val(pmdval)), PAGE_KERNEL);
        /* How pteval is really computed:
         * pteval = ((pmdval - PAGE_OFFSET) & PAGE_MASK) | PAGE_KERNEL; */
        write_dtlb_entry (pteval, vpnval);
}
```

C. Xtensa Exception Architecture Version 1

This appendix describes Xtensa Exception Architecture 1 (XEA1). You should use XEA2 in all new designs as it is more flexible and robust than XEA1. XEA1 is the original exception architecture and is not supported on Xtensa LX processors. For backward compatibility, Xtensa Tools Version 7.0 continues to support XEA1, where applicable.

The *Xtensa Instruction Set Architecture (ISA) Reference Manual* defines all details of both variants.

C.1 XEA1 Versus XEA2

This section highlights differences between the two exception architectures. Note that it does not attempt to re-explain all the details of various features. Rather, it offers a list of features to consider carefully when reading the respective descriptions. This section is useful for programmers migrating from one exception architecture to another, or considering how to configure an Xtensa processor.

The following elements are unique to XEA2:

- `PS.EXCM`
- `PS.RING`
- `DoubleExceptionVector`
- `DEPC` special register
- `EXCVADDR` special register
- `rfde` instruction
- Use of `rfe` to return from a user exception
- Independent memory-access settings for instruction and data memory regions

The following elements are unique to XEA1:

- `rfue` instruction
- `CACHEATTR` special register
- `PS.INTLEVEL` must be zero when running windowed code

XEA2 introduces the concept of “effective” or “current” values of processor states that differ from corresponding `PS` fields. For example, `PS.INTLEVEL` may be zero, but if `PS.EXCM` is set, the current interrupt level (`CINTLEVEL`) is `EXCMLEVEL`. Other processor states include `CRING` and `CWOE`. The processor state in XEA1 is always reflected in the `PS` fields; there is no “effective” or “current” value that differs from the `PS` fields.

Xtensa IV (T1040) and Xtensa V (T1050) support both exception architecture variants.

XEA1 and XEA2 have significant differences in exception semantics. Refer to Appendix C.3.2 and Section 3.1.2 on page 38 for comparisons.

C.2 Differences in Register Window Exception Vectors

The `l32e` and `s32e` instructions were introduced to the Xtensa ISA in the T1040 release, at the same time as the XEA2 option was introduced. These instructions are available in both XEA1 and XEA2. In addition to addressing important issues for XEA2, such as memory-access privileges for configurations with a full MMU, these instructions permit a slightly more efficient implementation of window exception handlers. Previous implementations were limited to `l32i` and `s32i` instructions, which support only positive offsets. `l32e` and `s32e` instructions support only small negative offsets, which work more naturally with window exception handlers.

The `l32e` and `s32e` instructions are sometimes associated with XEA2. This is mostly because the XTOS window exception handlers use them only for processors configured with XEA2 for historical reasons of simplicity.

C.3 Xtensa Exception Architecture 1 Description

This section highlights the key points of XEA1 that affect programmers and assumes familiarity with the ISA.

C.3.1 The PS Register for XEA1

Understanding the Miscellaneous Program State Register (`PS`) is necessary before considering other exception elements. `PS` contains miscellaneous fields that are grouped together so they can be saved and restored easily for interrupts and context switching. Figure 20 shows the layout, and Table 10 describes the fields. Note that the layout is the same as for XEA2, without the `RING` and `EXCM` fields. There are significant semantic differences when compared with XEA2.

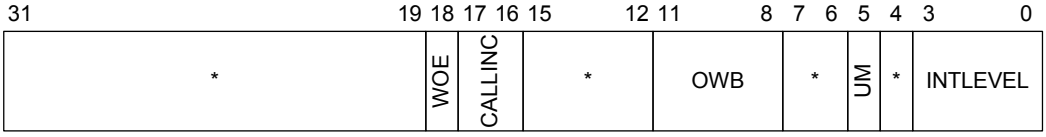

Figure 20. `PS` Register Format—Xtensa Exception Architecture 1

Table 10. PS Register Fields—Xtensa Exception Architecture 1

Field	Width (bits)	Definition
INTLEVEL	4	Interrupt-Level Disable This field is the current interrupt level of the processor. Interrupts of level N or less are disabled when INTLEVEL is N. To enable an interrupt of level N, INTLEVEL must be less than N, and the INTENABLE bit for that interrupt must be set to 1. The processor changes this field to 1 when entering an exception handler for the purpose of disabling level-one interrupts. When returning from an exception, the processor sets this field to 0 (see instructions RFUE, RFE, RFWO, and RFWU). At processor reset, the value of this field is 15.
UM	1	User Vector Mode 0 → kernel vector mode 1 → user vector mode Conventional use of the user vector mode bit is for controlling stack switching on exceptions. Kernel vector mode indicates the processor is using the kernel exception stack. User vector mode indicates that a different stack is in use, and exceptions will need to switch stacks. Note that hardware does not enforce conventional use, but provides this bit as a convenience to operating systems. Operating-system implementations are free to redefine the meaning of this bit. They may even ignore it entirely, provided they capture exceptions at the appropriate vector. An alternate view of the UM bit is that it is solely a vector selector bit. If zero, exceptions are transferred to the KernelExceptionVector. If one, exceptions are transferred to the UserExceptionVector. The processor sets UM to zero on reset and user vector exceptions. Programmers should not associate memory protection or privileges with either user or kernel vector mode. Notions of protection and privileged instructions do not exist. Again, the UM bit is simply a vector-selector bit. At processor reset, the value of this field is 0.
OWB	NAREG/4	Old Window Base The value of WINDOWBASE before window overflows and underflows. At processor reset, the value of this field is 0.
CALLINC	2	Call Increment CALLn and CALLXn instructions set CALLINC to the window increment amount. ENTRY instructions use CALLINC to rotate the logical window. At processor reset, the value of this field is 0.

Table 10. PS Register Fields—Xtensa Exception Architecture 1 (continued)

Field	Width (bits)	Definition
WOE	1	Window Overflow-Detection Enable 0 → overflow detection disabled 1 → overflow detection enabled At processor reset, the value of this field is 0.
*		Reserved for future use. Writing anything other than a zero value or the corresponding value read from PS results in undefined processor behavior. (Note: In XEA1, these bits are ignored on write and read as zero. XEA1 is no longer supported in newer processors, so there is no future use per se.)

C.3.2 Exception Semantics

The following pseudo code describes the exception semantics. Of particular interest is that the processor raises PS.INTLEVEL to 1 and clears PS.UM to 0.

```
EPC1 ← PC
PS.INTLEVEL ← 1
if PS.UM
        EXCCAUSE ← cause
        nextPC ← UserExceptionVector
        PS.UM ← 0
        PS.WOE ← 0
elseif Window Overflow
        process Window Overflow
else
        EXCCAUSE ← cause
        nextPC ← KernelExceptionVector
        -- note PS.WOE is left unchanged
        -- note PS.UM is already 0
endif
```

C.3.3 Exception Vectors for XEA1

XEA1 provides two exception vectors, UserExceptionVector and KernelExceptionVector. When an exception condition occurs, the processor uses PS.UM to select the appropriate vector at which to transfer control. If PS.UM is 1, the processor jumps to the UserExceptionVector; otherwise, it jumps to KernelExceptionVector.

C.3.4 Program Code for XEA1

On exceptions, the processor raises PS.INTLEVEL to 1 specifically to mask level-one interrupts. All return-from-exception instructions (RFUE, RFE, RFWI, and RFWU) set PS.INTLEVEL to zero. This functionality requires that all windowed code execute at interrupt level zero. This typically affects application software and any exception or level-one interrupt handlers that are written in C.

Software that tries to violate this requirement will find the processor running at interrupt-level zero on return from the first window overflow.

C.3.5 Windowed Code and Xtensa Processor State for XEA1

Any code that invokes the windowed-register mechanisms according to the Windowed ABI is called *Windowed Code* in the following. This section describes the processor state required to execute windowed code.

When the processor transfers control to a vector (for example, UserExceptionVector), the processor state is insufficient for executing windowed code. Some introductory code, written in assembly, is required to modify the processor state to make it suitable for windowed code. The following conditions must be met.

PS.WOE must be 1. PS.WOE enables the window-overflow exceptions in the processor. On Xtensa IV (T1040) processors or later, executing a windowed-register instruction (for example, CALLn, CALLXn, ENTRY, or RETW) when PS.WOE==0 will result in an Illegal Instruction exception.

The WINDOWSTART and WINDOWBASE special registers must be coherent. That is, the expression ((1 << WINDOWBASE) & WINDOWSTART) must be nonzero, and WINDOWSTART must reflect contiguous window frames of appropriate size. (It is often easiest to set only one bit in WINDOWSTART and show only the current window frame.) These registers are undefined on processor reset, so reset handlers must set them. On Xtensa IV (T1040) processors or later, executing a windowed-register instruction with an incoherent WINDOWSTART and WINDOWBASE may result in an Illegal Instruction exception. See Chapter 5 for more details on these special registers.

General-purpose register a1 must contain a valid stack pointer. The ABI and Software Conventions for Xtensa processors designate a1 as the stack pointer. While executing windowed code, the possibility of window-overflow and window-underflow exceptions exists. The window-exception handlers assume a valid stack pointer. Additionally, valid stack frames must exist in the stack space in memory. Section 5.2 on page 76 details the Xtensa stack layout. Debuggers for Xtensa processors consider a return value of zero (in a0) as an indicator of the last valid stack frame in a trace back of function calls.

Windowed code must run at interrupt level zero (PS.INTLEVEL == 0) because window exceptions clear PS.INTLEVEL on return. See Section C.3.4.

If interrupts must be disabled while running windowed code, then to satisfy the PS.INTLEVEL==0 requirement, the introductory assembly code should mask all relevant interrupts via INTENABLE and lower PS.INTLEVEL to zero. Consider an example of level-one interrupts. Even though the processor is then running at interrupt level zero, the effective interrupt level is really one because all level-one interrupts are disabled.

Unlike reset handlers, exception handlers must consider the context from which control arrived. Exception handlers are likely to have meaningful state in general-purpose and special registers, that must be saved before and restored after executing windowed code. The underlying kernel, runtime, or low-level software design, particularly stack usage, usually has the greatest impact on deciding what state needs preservation. For example, if all software uses only a single stack, exception handlers likely need to preserve only the current logical window and various special registers before calling windowed code. If an operating system and user tasks do not share stack space and their stacks cannot be linked somehow, exception handlers likely must save the entire physical register file to memory and save various special registers before calling windowed code.

C.3.6 XEA1 Interrupts and Window Overflows

For XEA1, assembly code that is running at processor interrupt levels higher than zero should carefully avoid causing a window overflow because returning from a window overflow will cause the processor to return to interrupt level zero. A safe recommendation is that the sequence of instructions between the rsil instruction and the wsr/rsync instructions should not reference a register number higher than any register number used in the rsil instruction.

C.3.7 Controlling the Caches for Processors with XEA1

Xtensa processors with XEA1 have a CACHEATTR register. CACHEATTR controls memory access modes, including whether or not a particular memory region is cached and accessible. The reset value of CACHEATTR is 0x22222222, selecting cache-bypass mode for all 512 MB regions. This guarantees that the code at the reset vector is in a region that is executable and not cached, so that it can be fetched before initializing the cache.

The Xtensa ISA does not include a mechanism for modifying the CACHEATTR bits of the address that it is currently executing. This means that there is no micro architecture-independent way to change the caching properties on the currently executing code.

However, for all implementations of Xtensa processors with XEA1, the following sequence will allow changing of caching properties of the memory region from which we are currently executing code.

```
        movi    a2, 0x22222222          // this value disables caches
        j       1f
        .align  16
1:      wsr     a2, CACHEATTR
        isync
        nop
        nop
        nop
        nop
```

Turning on the caches is handled in the same way as turning off the caches, and with essentially the same code sequence.

```
        movi    a2, 0xFF21212F    // some system-specific value
        j       1f
        .align  16
1:      wsr     a2, CACHEATTR
        isync
        nop
        nop
        nop
        nop
```

The value loaded into the `CACHEATTR` is different because the value is turning on the caches for various sections of the memory. Each of the eight nibbles in the `CACHEATTR` register corresponds to a 512 MB region within the 4 GB memory space. The Xtensa ISA fully specifies all the valid values (cache attributes) of each nibble.

In this case, we are using three different values:

- A nibble of value ‘2’ specifies a bypass-cache (cache disabled) region.
- A nibble of value ‘1’ specifies a region whose accesses are cached.
- A nibble of value ‘0xF’ specifies a memory range as invalid. Memory references to addresses that are in an invalid range will cause an exception.
- Note that there is no way to separately control the enabling of instruction and data caches for the same memory range. Either both caches are on or both caches are off.

It is important that a cache miss (and a subsequent refill of a cache line) not occur during the execution of this code sequence. Therefore, the `wsr` instruction must be aligned to the minimum cache-line size, which is 16. This code sequence fits into 16 bytes. If this code sequence is aligned to a byte boundary greater than 16, the linker may corrupt your reset vector by inserting gaps.

For more details on changing cache attributes, see the cache sections of the HAL chapter in the *Xtensa System Software Reference Manual*.

Index

Symbols

__asm__ statement 87
.align directive 9
.comm directive 95
.global directive 9
.literal section 23, 27
.section directive 25, 48
.space directive 95
.text directive 9
.type directive 9

A

ABI 12, 40, 65, 76, 95, 97, 127, 215
  non-windowed 148
Access Mode 52, 155, 157
  and processor reset 56
  cache attribute 157
  dirty bit 157
  valid bit 157
add_task 125, 126
Adding interrupt dispatching mechanism 116
Address mappings on reset 51
Address space identifier (see ASID) 155
allocated_tasks 126
AM (see Access Mode) 52
Ancestor
  stack frames 97
  window frames 81
Architectural overview of register windows 65
Argument passing (see parameter passing) 12
ASID 155
  assignment cycle 157
  dynamic assignment of 157
  kernel 156
  management of 156
  reserved values 155
Assembler
  GNU 33
  instruction scheduler 30
  relaxations (and literals) 22
Assembly code
  assembler relaxations 22
  auto-scheduling instructions 30
  compiler-generated 6
  disassembly 24, 71
  efficient branches 28
  example 5
  file name suffixes 20
  hand-scheduling instructions 29
  more readable version 7
  portability 32
  preprocessing 20
  readability 32
  reasons to write 5
  rules of thumb 28
  tips and tricks 31
  to read CCOUNT 85
  useful macros 32
  writing efficient 28
Atomic 91
atomic_add 21
Autorefill
  decay Replacement algorithm 160
  example operation 161
  hardware-generated load 160
  PTE-load computation 161
  second-level miss 160
  way 160

B

Base processor core 141
Base save area 76, 97, 131
  moving 99, 103
BBCI instruction 32
BBCI.L instruction 32
BBSI instruction 31
BBSI.L instruction 31
BGEUI instruction 10
Branch instructions 28
break 63, 96
Break in func1 73
Breakpoints 68, 121
bss section cleared 61
Building the ResetVector 63

C

C
  code won't run 58, 61
  execution environment 149
  interrupt handler 101
Cache

attribute 157
changing properties 216
initialization 50
line locking 55
turn on 216
turn on data cache 55
turn on instruction cache 53
CACHEATTR register 211, 217
cacheattrasm.h 51, 58
Call Increment 37
Call instructions 17, 66
CALLINC 37, 127
Calling
the C handler 101
the first C function 61
callmain 68
C-callable Assembly to read CCOUNT 85
CCOMPARE register 85, 106
reading 88
setting 89
first timer value 90
updating 85
CCOUNT register 49, 85
reading with C-callable assembly 85
reading with inline assembly 86
wrapping 106
CHAL 2, 118, 148
CINTLEVEL 36
CLOOPENABLE 36
Code
locating at fixed locations 24
size 64
Code density 65
option 10
Compiled code
code density 65
performance 65
Configurability 2, 141
abstracting 32
Context switching 125
and coprocessors 146
lazy 142
preemptive 131
testing 135
Coprocessor option 141, 142
Coprocessor save area 145
Coprocessor state 141, 145, 146, 147
Coprocessor, owned by user task 149
Core
processor state 146
register state 146
coreasm.h 32, 118
CP_SAVE_AREA 150
CPENABLE register 142, 148, 150
CRING 36, 37, 156
current_task 126
Currently pending processor interrupt 105
Custom state 141
policy decisions 141
Custom state save area 145
CWOE 36, 37
Cycle count 49

D

Data cache, initializing 55
DBREAKA register, and processor reset 49
DBREAKC register, and processor reset 49
Debug Option and processor reset 49
Debugging 40
with GNU 49
Decay Replacement algorithm 160
deep_call 136
Default Xtensa run time 25
Definitions
notations xi
terms xi
Demand paging 154
DEPC register 211
Dirty bit 157
Double exception vector 38, 211
DTLBCFG register
and processor reset 163
format 164
undefined processor behavior 163
Dummy stack frame 99
Dynamic memory allocation 145

E

ENTRY instruction 9, 19, 127
Entry point of user task 127
EPC1 register 96, 104
EXCCAUSE register 96, 149, 150
Exception
semantics and XEA1 214
semantics and XEA2 38
Exception mode 103
and masking interrupts 38
EXCM36, 44, 103, 104, 118, 127, 133, 156, 211
and exception handling 36

overriding PS fields 36
EXCSAVE_1 register 94, 95
EXCVADDR register 160, 161, 211
Exit code 63
Extensible 141
Extra save area 76, 127

F

find_ls_one 118
First C function returns 63
First instruction 48
First timer value 90
Full context switching source code 169
Function
alignment requirements 19
invocation and return 17
invocation example 18
nested 97
parameters 10

G

global_fool_the_optimizer 136
GNU assembler 33

H

HAL 2
Heartbeat 85
High-priority interrupts 41
handlers in C 41
Housekeeping 85

I

IBREAKENABLE register
and processor reset 45
ICOUNT register
and processor reset 45
ICOUNTLEVEL register
and processor reset 45
III instruction 54
Instruction cache, initializing 53
Instruction Set Architecture (ISA) 1
intdisp.c 170, 208
intDispatch 102
INTENABLE register
enabling interrupts 91
modification 91
nested interrupt handling 108
processor reset 49
INTERRUPT register 105
Interrupts
and processor reset 49
between callN and entry 98
complications 106
determining which to process 106
dispatch table 116, 120
dispatching mechanism 116
enabling 91
handling 85, 93
latency 93
memory efficiency 93
nested 107, 119
allowing 112
examples 120
NMI 47
restoring processor state 103
return overview 102
returning from 104
saving processor state 96
speed 93
stack 93, 95
custom frame 147
layout 100
user frame 97
using 109
timer 85
timer handler 105
tracking pending 105
interrupts.h 169, 199
INTLEVEL register 36, 112, 211
ISA 1
ITLBCFG register
and processor reset 163
register format 164
undefined processor behavior 163

K

Kernel exception
handler 5
handling 119
vector 38
literals within 26
Kernel mode 155
Kernel vector mode 37
kernv.S 171

L

L16SI instruction 15
L1H.S instruction 171
L32R instruction 23
Large page support 159
Large values as function parameters 16
Lazy context switching 142

LBEGIN register 96
LCOUNT register 96
  and processor reset 45
LEND register 96
Level-one interrupts 40
LHAL 2
Linker script 24, 59
  memories 26
Linking stack frame 93
Linux 154, 165
literal_prefix 27
Literals, locating in .text 48
Local memory
  and ROM unpacking 58
Locating code at fixed locations 24
LOCKLEVEL 21
Logical registers 65
Logical window 18, 30, 65, 68
Loops
  body 10
  setup 10
Low-level interrupts 40

M

main 68
main.c 180
Makefile 169
Mapping large pages 159
Medium-level interrupts 41
Memory
  dynamic allocation 145
  map 164
  physical 26
  protection 153, 156
Memory management options
  MMU with TLB and Autorefill 153
  Region Protection
    with XEA1 153
    with XEA2 153
  Region Protection with Translation 153
  simple translation example 153
Memory management unit (See *MMU*)
MMU
  address translation 153
  configuration 154
  example memory map 164
  example way configuration 163
  general purpose 154
  successful translation 158
  translation hardware 154
  undefined behavior 156
MOVI instruction 22
Moving the base save area 103
--mtext-section-literals directive 63
Multiple page sizes 163

N

Nested functions 97
Nested interrupts
  allowing 112
  handling 107
newUserStackPtr 129, 131, 132
Nibble 57, 217
NMI
  and processor reset 47
  interrupts 47
Noncoprocessor save area 145
Noncoprocessor state 141, 145, 146, 147
Non-windowed ABI 148
Nort linker support packages 25
nort-rom 58

O

Offset bits 158
Offsets, vector 78
Old Window Base 37
OS compatibility 2
Overlapping TLB mappings 158
OWB 37, 43

P

Page table 160
  mapping of 159
Page table entry (See *pte*)
Parameter passing 12
  in registers 12
  larger than 32 bits 16
  on the stack 12
  smaller than 32 bits 15
  structures in registers 17
PC register
  and processor reset 43
Performance 58, 65
  of compiled code 65
Physical page number 52
Physical register file 30, 67, 68, 93, 99, 131
Pipeline bubbles 29
PPN (Physical Page Number) 52, 155
Preemptive time slice 128
Preface xi
Privilege level 37

Privilege mode 155, 156
Processor
  base core 141
  configurability 141
  extending 141
  resetting 43
  restoring state 103
  resume execution of 102
Program counter, obtaining 31
PS register 43, 96, 127, 133
  and calling C code 61, 101
  and processor reset 44
  and XEA1 212
  and XEA2 35
PS.CALLINC 43
PS.OWB 43
PS.WOE 44
PTBASE register field 161
pte
  autorefill ways 160–161
PTE-load computation 161
PTEVADDR register 161

R

RASID
  and processor reset 156
  register 156, 158
  unique field values 156
RDTLB1 instruction 56
read_ccount 85
REF instruction 211
Region Protection (see Memory management options) 153
Region Protection with Translation (see Memory management options) 153
Register
  defined values 43
  files 141
  state at callmain (table) 69
  state at func (table) 72
  state at func1 (table) 74
  state at main (table) 70
  state following overflow (table) 75
  use lowest-numbered 30
Reset
  address mappings 51
  handler 5, 24
  handler pitfalls 53
  processor 43
  state 43
  vector 24, 43, 48, 63
    making space for literals 48
reset.S 181
Restoring the processor state 103
Returning from an interrupt 104
retw.n (and retw) 10
RFDE instruction 211
RFE instruction 36, 133
RFUE instruction 211
RFWO instruction 36
RFWU instruction 36
RING 37, 156, 211
Ring 156
  ASID register 156
  current value 156
  level 156
  ring-to-ring transition 156
RingCount 37
RISC 1
RITLB1 instruction 56
ROM unpacking 58
Round robin scheduling 125, 129
Run time, Xtensa default 25

S

SAR register 96
Saving user stack pointer 99
Scheduling, round robin 125, 129
set_access_mode 56
Shutdown the simulator 63
SIMCALL instruction 63
Sleep 128, 132
Software conventions 12, 40, 76, 215
Source code, full context switching 169
Stack
  alignment requirements 95
  ancestor frames 97
  frame
    dummy 99
    last valid 40
    per function requirement 19
    size of 19
  initialization 126
  interrupt 93
  layout
    for call frames 76
    for interrupts with custom state 147
    for user interrupts 96
    of interrupt stack 100
  linking frame 93

pointer
alignment requirements 127
finding adjacent frame's 80
initial computation of 126
initialize 61
locating 80
must be valid 40, 133
register designation 40
registration 126
switching stacks 100
user 99
window exception handlers 40
size of 13
traceback 40
user interrupt frame 97
State manipulation routines 142
Switching the context 131
System heartbeat 85
System tick 85

T

Task control block 126, 145, 149
Task creation 125
task_stacks 126
task.c 186
task.h 187
TCB (see Task control block) 126
Tensilica Instruction Extension (See TIE) 141
Tick (see Timers) 85
TIE
lazy context switching 142
save and restore mechanisms 148
stack impacts of
bad user stack layout 146
user stack layout 147
Time slice, preemptive 128
TIMER_INTERVAL 106
timer.h 87, 187
Timers 120
interrupt handler 105, 128
interrupt handling 128
interrupts 85
number of 107
setting 85
TLB
4-way, set-associative 158
flush 157
for data memory 154
for instruction memory 154
hit 155
invalid entry 155
miss 155
miss handler example 165
multihit 155, 156, 158
multiple page sizes 163
overlapping mappings 158
ways 154
Translation lookaside buffer (see TLB) 154
Troubleshooting
C code cannot run 61
error in developing run-time 96
first C function returns 63
True cycle count 49

U

UM 94, 119, 127
Unpacking, ROM 58
User exception handler 5, 24, 94
prolog 95
start 95
User exception vector 24, 38, 93, 94, 96
User stack
layout with TIE 147
pointer, saving 99
User state 141
User task, entry point of 127
User vector mode 37
userStackPtr 95, 99, 129, 131, 132
userv.S 189

V

Valid bit 157
Vector offsets 78
Virtual memory support 154
Virtual page number (see VPN) 155
Virtual to physical address translation 153
VPN 52, 155
offset bits 158

W

wait_func 120
WDTLB instruction 52, 56
Window
ancestor frames 81
exception handlers 77
overflow exception 67, 73
Window exception handlers 5, 65, 77
more efficient implementation 212
Window frame 65, 68, 99, 103, 114, 135
live 93, 99
Window increment 17

Window overflow....44, 76
avoiding....30
trigger....73, 92
triggers....114
Window Overflow Enable....37
Window pane....65
intermittent corruption....99
Window save area....19, 76
Window vector base....78
WINDOWBASE register....67, 68, 71, 76
Windowed Code....39
Windowed register option....65
WindowsCE....154
WINDOWSTART register....67, 68, 71, 99
coherency with WINDOWBASE....40
WindowVectors.S....65
windv.S....189
Wired way
and processor reset....159
mapping large pages....159
mapping page tables....159
WITLB instruction....52, 56
WOE....37, 44, 127

X

XEA1....211
XEA1 versus XEA2....211
XEA2....3, 35, 211
xt-as vs. xt-gcc....20
Xtensa ABI....12, 40, 65, 76, 95, 97, 127, 215
Xtensa CHAL....118
Xtensa Exception Architecture 1 (see XEA1)..3
Xtensa Exception Architecture 2 (see XEA2)..3
Xtensa HAL....2, 86, 142, 148
Xtensa Instruction Set Architecture (ISA)....1
Xtensa Processor Generator....2
Xtensa software conventions....12, 40, 76, 215
Xtensa Xplorer....141
xt-gdb....49, 71
xthal_restore_cpregs....148
xthal_restore_extra....148
xthal_save_cpregs....148
xthal_save_extra....148

Z

Zero-overhead loop....32